Progetto GPU Computing:
Parallel Differential Distinguisher

Matteo Onger

Luglio 2024

1 Introduzione

La crittanalisi differenziale è una forma di crittanalisi che si concentra sullo studio delle differenze nei dati forniti in ingresso a una funzione crittografica, e nei dati restituiti in output da quest'ultima. Un attacco differenziale punta a sfruttare a proprio vantaggio eventuali proprietà non casuali degli output delle primitive crittografiche in analisi, a cui sono forniti in ingresso input con differenze note e spesso accuratamente scelte.

Si consideri, per esempio, un cifrario a blocchi f: siano P_1 e P_2 una coppia di testi in chiaro di n bit scelti casualmente, ne consegue che la differenza in input ad essi associata è pari a:

$$\Delta_{in} = P_1 \oplus P_2.$$

Mentre, indicando con C_1 e con C_2 i rispettivi testi cifrati prodotti da f, la differenza in output può essere espressa come:

$$\Delta_{out} = C_1 \oplus C_2 = f(P_1) \oplus f(P_2).$$

Per garantire la massima sicurezza, una funzione crittografica dovrebbe essere del tutto indistinguibile da una fonte casuale di bit, quindi, per ogni differenza in input, ogni possibile differenza in output dovrebbe avere la stessa probabilità di essere prodotta.

$$\Delta_{out} \sim U(0, 2^n - 1) \qquad \forall\, \Delta_{in} \in [0, 2^n - 1]$$

Se le differenze in output non seguono una distribuzione uniforme, un attaccante può sfruttare questa vulnerabilità a proprio vantaggio, per esempio, per tentare di recuperare la chiave di cifratura.

Un distinguisher differenziale è un classificatore che accetta in input d coppie indipendenti di testi. Ogni coppia può essere stata generata casualmente o essere stata prodotta dalla funzione crittografica in analisi a partire da una coppia di testi in chiaro; l'obiettivo del distinguisher è proprio stabilirne l'origine.

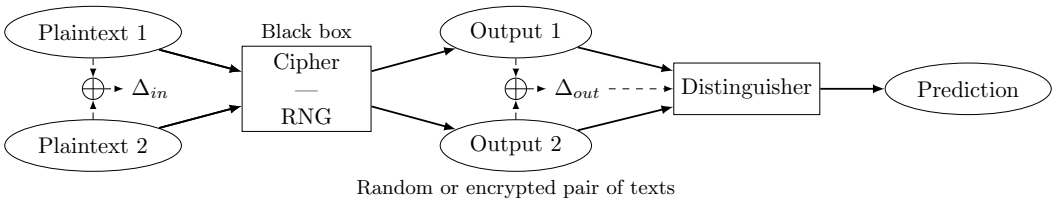

Figure 1: Schema generale del processo. I nodi ellittici rappresentano i dati in input/output.

Affinché il distinguisher abbia un basso tasso di errore, la distribuzione delle differenze in output deve differire il più possibile da una distribuzione uniforme. Quindi, visto che gli attacchi di crittoanalisi differenziale sono tipicamente attacchi con testo in chiaro a scelta, conviene cercare una differenza in input Δ_{in} che garantisca ciò, e utilizzarla per generare le coppie di testi in chiaro a partire dalle quali saranno prodotti i dati forniti in input al distinguisher.

Per una descrizione più dettagliata degli argomenti sopra accennati, o per vedere come un distinguisher possa essere utilizzato all'interno di un attacco differenziale, si consiglia di consultare fonti terze come [1], [3], [4] e [6].

2 Scopo del progetto

Lo scopo del progetto è contribuire allo sviluppo di un framework che integri diversi algoritmi per la ricerca delle migliori differenze in input, diversi criteri per il calcolo della fitness delle differenze trovate e diverse reti neurali che implementino il distinguisher vero e proprio, ottenendo così uno strumento altamente personalizzabile da poter applicare a diverse funzioni crittografiche.

In particolare, ci si è concentrati sulla componente software, nota come *optimizer*, che precede il distinguisher e che si occupa di trovare "buone" differenze in input. La componente principale dell'algoritmo in questione, infatti, è tipicamente un algoritmo evolutivo che può essere ampiamente velocizzato e/o migliorato grazie all'uso di GPU. L'elevato grado di parallelismo introdotto dalle GPU non consente semplicemente di eseguire i medesimi calcoli in meno tempo, ma permette, per esempio, di considerare una popolazione più ampia di differenze in input e un campione più ampio di differenze in output, portando quindi a stime più accurate delle fitness degli individui della popolazione.

Di seguito si riporta l'algoritmo, ad alto livello, dell'*optimizer*.

Algorithm 1 Optimizer

Require: $scenario$, $cipher$, $evoalg$, $fitness$ ▷ *parametri di configurazione*
  $deltas \leftarrow \emptyset$ ▷ *insieme differenze trovate*
  $round \leftarrow 0$ ▷ *numero di round di cifratura eseguiti*
  repeat
    $round \leftarrow round + 1$
    ▷ *l'algo evolutivo restituisce le differenze trovate con la rispettiva fitness* ◁
    new_deltas, $scores \leftarrow evoalg(scenario, round, cipher, fitness)$
    $deltas \leftarrow deltas \cup new\_deltas$
    $best\_score \leftarrow \max(scores)$
  until $round >= MAX\_ROUND$ or $best\_score < THRESHOLD$
  ▷ *rivaluta tutte le differenze* ◁
  $scores \leftarrow fitness(deltas)$
  output $deltas$, $scores$ ▷ *migliori differenze in input trovate*

Nel capitolo successivo verranno approfonditi i dettagli implementativi dell'algoritmo 1, mentre maggiori informazioni in merito agli algoritmi evolutivi sono riportate in [2].

3 Dettagli implementativi

La struttura più naturale per rappresentare i testi, in chiaro o cifrati che siano, e quindi le differenze, che, come detto, sono semplicemente lo XOR di una coppia di testi, è un vettore. Per ridurre i tempi di computazione e la memoria utilizzata, altra risorsa fondamentale data la necessità di tenere in memoria più differenze possibili per poterle processare in parallelo, i vettori in questione sono array di interi senza segno su b bit, dove b dipende della funzione crittografica considerata.

Inoltre, sempre in quest'ottica, si è tentato di ridurre al minimo lo spostamento di dati tra l'host e il device, e di rendere il più allineato, coalescente e privo di collisioni possibile ogni accesso in memoria.

La libreria *Cupy* è stata ampiamente utilizzata per eseguire efficacemente alcune manipolazioni di base sui vettori rappresentanti i testi, le differenze in input o le differenze in output, mentre per eseguire alcune operazioni più articolate, anche per ragioni di efficienza, è stato necessario ricorre a kernel appositamente implementati grazie a *Numba*.

Il compito dell'*optimizer* è collezionare le migliori differenze in input trovate dall'algoritmo evolutivo per ogni round di cifratura e accodarle alle differenze già note. Questo avviene in tempo costante grazie ad un apposito kernel. Prima di restituire l'elenco completo delle differenze trovate, esse vengono rivalutate. Di seguito si analizzano brevemente le principali componenti invocate, direttamente o indirettamente, dall'*optimizer*.

Filtro di Bloom

Per evitare di memorizzare e processare differenze in input già note, sprecando tempo e memoria, sia l'*optimizer* che l'algoritmo evolutivo vero e proprio usano un filtro di Bloom [5]. I filtri in questione vengono memorizzati nella memoria globale della GPU e un apposito kernel è stato sviluppato per rendere il più veloce possibile la ricerca e l'inserimento. In particolare:

- la ricerca, ipotizzando di avere d thread per ogni elemento da cercare, dove d coincide col numero di valori di hash calcolati per elemento, può avvenire in tempo costante $\mathcal{O}(1)$;
- mentre l'inserimento di un nuovo elemento nel filtro costituisce una sezione critica, che viene garantita grazie a un mutex implementato sfruttando le operazioni atomiche disponibili. Quindi se su m elementi, n sono nuovi, il loro inserimento deve avvenire sequenzialmente al fine di evitare inconsistenze ed errori. La complessità temporale sarà quindi $\mathcal{O}(n)$.

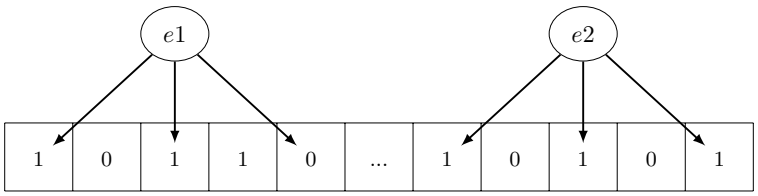

Figure 2: La figura mostra un semplice esempio in cui, per ogni elemento, vengono calcolati tre valori di hash. L'elemento $e1$ viene etichettato come nuovo, mentre l'elemento $e2$ come già visto.

In tutto questo, si sta volutamente trascurando il tempo necessario per calcolare l'hash di un elemento, dato che:

- in generale, le funzioni di hash, sopratutto se non-crittografiche, hanno bassi tempi di computazione;
- inoltre, gli elementi sono qui vettori di interi senza segno che, visto ciò che rappresentano, sono sempre piuttosto corti, lunghi al massimo qualche parola.

Algoritmo evolutivo

L'algoritmo evolutivo, chiamato dall'*optimizer*, tenta di trovare le migliori differenze in input per il round considerato della funzione crittografica in analisi. Ogni volta, la prima generazione di individui viene generata casualmente, mentre le generazioni successive sono prodotte ogni volta a partire dalla generazione precedente. Le generazioni, per definizione, sono strettamente sequenziali e non si prestano quindi ad essere parallelizzate; al contrario, gli individui che costituiscono la prole sono del tutto indipendenti tra di loro, e possono quindi essere facilmente generati in parallelo. La generazione di quest'ultimi è infatti affidata ad un apposito kernel che viene invocato lanciando un thread per ogni individuo della prole da produrre. La complessità computazionale del kernel in questione è lineare al numero di parole che costituiscono un individuo ed è per cui solitamente trascurabile, anche qualora si volessero applicare mutazioni e crossover più sofisticati.

Fitness

Il calcolo della fitness delle differenze in input è forse il principale collo di bottiglia dell'intero processo, perché per ogni differenza in input da valutare è necessario raccogliere quante più differenze in output possibili al fine di produrre una stima che sia il più accurata possibile.

Nell'implementazione realizzata, la fitness coincide col *bias score* che, per ognuna delle l differenze in input Δ_{in}, dato un insieme S di m differenze in output Δ_{out} di lunghezza n bit, viene calcolato come:

$$\forall\ \Delta_{in},\ S = \{\Delta_{out} \mid \Delta_{out} \in \{0,1\}^n\} \implies fitness(\Delta_{in}) = \frac{\sum_{i=0}^{n-1} |0.5 - \mathbb{P}(\Delta_{out}[i] = 1)|}{n} \in [0, 0.5]$$

dove $\mathbb{P}(\Delta_{out}[i] = 1)$ è la probabilità che l'i-esimo bit di una differenza in output sia pari a uno, ed è quindi stimata come sotto riportato.

$$\mathbb{P}(\Delta_{out}[i] = 1) = \frac{\sum_{\Delta_{out} \in S} \Delta_{out}[i]}{m}$$

Un approccio puramente sequenziale avrebbe tempi di computazione troppo lunghi, nell'ordine di $\mathcal{O}(l \cdot m \cdot n + l \cdot m)$. Mentre, grazie al parallelismo, se non vi fossero limiti alle risorse hardware disponibili, i tempi di computazione potrebbero essere ridotti a $\mathcal{O}(\log_2 m + \log_2 n)$.

Un kernel puramente parallelo è stato sviluppato e si è mostrato effettivamente efficace per valutare con precisione un numero estremamente ridotto di differenze in input. Ma se, come nel caso qui considerato, si ha la necessità di valutare un numero considerevole di differenze, un kernel puramente parallelo necessita di troppi thread e di troppa memoria, globale e condivisa, affinché, con l'hardware a disposizione, si possa effettivamente eseguire in parallelo, e risulta quindi avere prestazioni inferiori rispetto ad un kernel che prediliga un approccio intermedio.

Il kernel utilizzato, infatti, prevede di utilizzare solo $l \cdot n$ thread e ricorre alla riduzione in parallelo solo per calcolare la somma delle probabilità, mentre stima le probabilità iterando sulle m differenze in output. Per ridurre al minimo i tempi di computazione, in teoria pari a $\mathcal{O}(m + \log_2 n)$, si è tentato di ottimizzare al massimo il corpo del ciclo, ricorrendo anche al *loop unrolling*. Nel capitolo 4 è possibile osservare i tempi di computazione dei due kernel a confronto.

4 Risultati sperimentali

In quest'ultima sezione prima delle conclusioni, vengono riportati e analizzati i risultati di alcuni semplici test condotti al fine di verificare i tempi di computazione e la scalabilità dell'*optimizer* sviluppato, per poi confrontarlo con l'implementazione originale che non prevedeva l'uso di GPU. La tabella 1 riporta l'elenco dei test condotti.

Test eseguiti						
ID	#Round	#Generazioni	*card*(Individui)	*card*(Prole)	#Batch	$\#\Delta_{out}$ per batch
Test_01	2	10	32	128	1	16.384
Test_02	2	10	32	128	1	65.536
Test_03	2	10	32	128	1	262.144
Test_04	2	10	32	128	1	1.048.576
Test_05	2	10	32	128	1	2.097.152
Test_06	2	10	32	128	5	2.097.152
Test_07	2	10	32	128	50	2.097.152
Test_08	2	10	32	1.024	1	16.384
Test_09	2	10	32	4.096	1	16.384
Test_10	2	100	32	128	1	1.048.576
Test_11	8	10	32	128	1	1.048.576

Table 1: Caratteristiche dei test condotti.

Innanzitutto, dai dati riportati nella tabella 2, si vede chiaramente come non vi sia paragone tra l'*optimizer* che sfrutta la GPU e la versione originale, nonostante anche quest'ultima, essendo basata principalmente su *Numpy*, ricorresse in un certo senso al parallelismo. I primi 5 test effettuati mostrano chiaramente che, finché ad aumentare è il numero di differenze in output per batch, i tempi di computazione della versione basata su GPU restano quasi invariati, nonostante una crescita esponenziale dei campioni considerati.

Tempi di computazione (sec)		
ID	AutoND CPU	AutoND2 GPU-T4
Test_01	44,20	6,78
Test_02	207,50	8,49
Test_03	—,–	9,39
Test_04	—,–	17,22
Test_05	—,–	29,52
Test_06	—,–	120,78
Test_07	—,–	1139,24
Test_08	149,21	8,31
Test_09	245,05	12,38
Test_08	—,–	122,21
Test_09	—,–	159,32

Table 2: Tempi di computazione per test e *optimizer* in secondi.

Al contrario, come prevedibile, se ad aumentare è il numero di batch (Test_06 e Test_07), i tempi di computazione crescono linearmente. Questo è mostrato chiaramente anche dal grafico 3 sotto riportato.

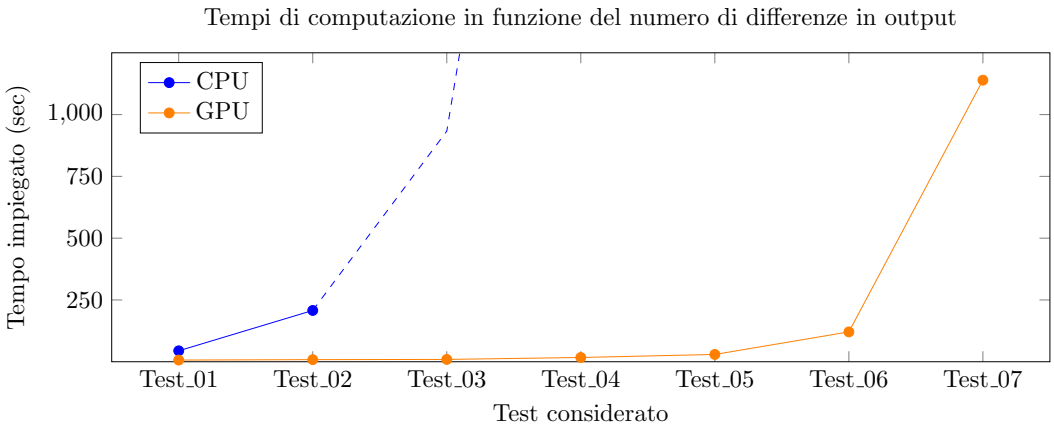

Figure 3: Grafico dei tempi di computazione dei primi 7 test effettuati.

I test Test_08 e Test_09 mostrano invece come i tempi di computazione scalino bene anche se ad aumentare è il numero di figli considerati per generazione, caratteristica potenzialmente importante per esplorare un più ampio numero di differenze in input.

I dati raccolti mostrano quindi chiaramente come, almeno finché le risorse hardware lo consentono, i tempi di computazione scalino bene al crescere del volume dei dati trattati. Si ha una crescita lineare solo se ad aumentare è il numero di generazioni (Test_10), il numero di round (Test_11) o il numero di batch. Di questi tre parametri, in realtà, come già detto, solo le generazioni sono, per loro stessa natura, strettamente sequenziali, mentre in teoria gli altri due casi sarebbero facilmente parallelizzabili, se lo si ritenesse vantaggioso.

Un analisi estremamente più dettagliata può essere fatta consultando il report prodotto da NVIDIA Nsight Systems.

Confronto tra kernel

In aggiunta, nella tabella 3 e nel grafico 4 sono mostrati i tempi di computazione dei due kernel sviluppati per il calcolo della fitness: il Kernel_01, che massimizza il grado di parallelismo, e il Kernel_02, che è stato poi preferito proprio in virtù dei dati raccolti e dello specifico caso d'uso. Per maggiori dettagli in merito si rimanda al codice.

Test eseguiti (sec)				
ID	$\#\Delta_{in}$	$\#\Delta_{out}$	Kernel_01	Kernel_02
Test_01	1	1.048.576	0,01	0,06
Test_02	1	1.073.741.824	4,42	25,20
Test_03	16	67.108.864	4,59	1,55
Test_04	128	8.388.608	5,06	0,88
Test_05	1.024	1.048.576	4,37	0,17

Table 3: Comparazione dei tempi di computazione dei due kernel al variare delle differenze, in input e in output, considerate.

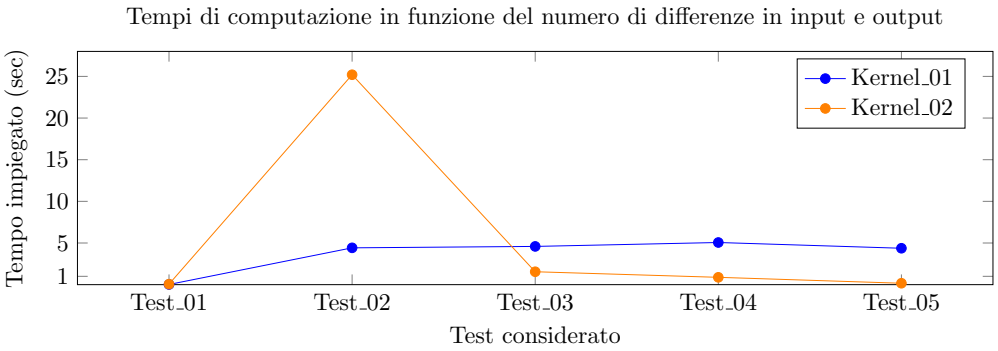

Figure 4: Grafico dei tempi di computazione.

5 Conclusioni

I risultati sperimentali, in perfetto accordo con l'analisi teorica, dimostrano chiaramente i vantaggi ottenuti grazie all'uso della GPU. Si noti, inoltre, che l'algoritmo evolutivo potrebbe ovviamente essere migliorato, ricorrendo a mutazioni e crossover più sofisticati, introducendo nuovi parametri e/o operazioni, come il *simulated annealing* per esempio, senza impattare significativamente sulle performance. Ovviamente, delle migliorie sono sempre possibili: per esempio, gli stream potrebbero essere introdotti per garantire ulteriore scalabilità e far sì che eventuali trasferimenti di dati possano avvenire contemporaneamente all'esecuzione dei kernel, così come il codice potrebbe essere adattato e ottimizzato per lavorare sfruttando più GPU. Questi esempi possono essere considerati possibili sviluppi futuri.

References

[1] Itai Dinur. Improved differential cryptanalysis of round-reduced speck. In *Selected Areas in Cryptography–SAC 2014: 21st International Conference, Montreal, QC, Canada, August 14-15, 2014, Revised Selected Papers 21*, pages 147–164. Springer, 2014.

[2] Agoston E Eiben and James E Smith. *Introduction to evolutionary computing*. Springer, 2015.

[3] Aron Gohr. Improving attacks on round-reduced speck32/64 using deep learning. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part II 39*, pages 150–179. Springer, 2019.

[4] Howard M Heys. A tutorial on linear and differential cryptanalysis. *Cryptologia*, 26(3):189–221, 2002.

[5] Adam Kirsch and Michael D Mitzenmacher. Building a better bloom filter. *Harvard*, 2005.

[6] Tarun Yadav and Manoj Kumar. Differential-ml distinguisher: Machine learning based generic extension for differential cryptanalysis. In *International Conference on Cryptology and Information Security in Latin America*, pages 191–212. Springer, 2021.