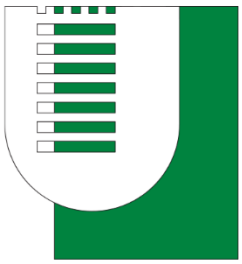
PROGETTO TEORIA DELLA SICUREZZA E CRITTOGRAFIA

A.A 2019-2020

Prof. M.Talamo

Prof. F.Arcieri



TOR VERGATA

UNIVERSITÀ DEGLI STUDI DI ROMA

"VALUTAZIONE COSTI BENEFICI DEI PRINCIPALI ALGORITMI DI CRITTOGRAFIA"

M.POLITI – S.SALMAN – A. SILVESTRI

INTRODUZIONE PROGETTO

Durante il progetto realizzeremo un'analisi costi benefici dei principali algoritmi di crittografia, in particolare:

- RSA
- ECC (*Elliptic Curve Cryptography*)
- AES (*Advanced Encryption Standard*)
- BLOWFISH

Confronteremo gli algoritmi a due a due, ovvero i due simmetrici e i due asimmetrici, confronteremo quindi:

1. **RSA** con **ECC**
2. **AES** con **BLOWFISH**

1 PARAMETRI DI VALUTAZIONE

Elenchiamo i confronti che effettueremo per derivare le statistiche necessarie alla successiva valutazione:

- **Entropia**
    - ❖ Chiavi sia pubbliche sia private: generiamo un campione esaustivo di chiavi e le concateniamo. Attraverso lo strumento OpenSSL utilizziamo il formato di uscita der(binario) e misuriamo l'entropia della concatenazione di n chiavi (*questo ci permetterà di approssimare l'entropia nella generazione*).
    - ❖ Messaggi: i messaggi devono essere cifrati.
        - ➢ Usiamo generatore numeri casuali Butterfly (meno prevedibile di Strong) su una stringa Stringa base di 100Mb
        - ➢ Creazione messaggio (multiplo di 2048 per RSA e multiplo di 320 per ECC)
    - ❖ Test1:

- ➢ Misuriamo entropia & Die-Hard della stringa generata (100MB)
- ➢ Cifriamo con RSA / ECC /AES /BLOWFISH, verifichiamo se l'Entropia aumenta o diminuisce(sia senza padding sia con padding)
- ➢ Test2: comportamento con una stringa uniforme(*1111111111*).

· **Lunghezza del messaggio (input o output)**

- Misuriamo la variazione di entropia e la velocità di cifratura e decifratura in base alla lunghezza del messaggio e creiamo dei report statistici che indicano il comportamento asintotico e i vari dati ricavati nei Test.

- **Velocità di cifratura e decifratura:**
  - Misuriamo la velocità di cifratura e decifratura dei file per i vari algoritmi(*nel caso di file sufficientemente grandi e.g. Stringa derivante da Butterfly 100MB*).
  - Velocità di generazione delle chiavi
  - Misuriamo il tempo di generazione delle chiavi(*Nel caso in cui la quantità sia notevole*).

- **Tecnica di padding**
  - Decifrare il messaggio
  - Struttura padding con standard PKCS PKCS1
  - Verifichiamo che riduca l'entropia (poichè introduce una componente strutturata nel messaggio)

- **Valutazioni:**
  - Quale algoritmo scegliamo e le motivazioni per cui lo scegliamo
  - Report statistici sullo studio delle caratteristiche
  - La nostra misura di Entropia rispetto ai Test Die-Hard (osservare se la valutazione Die-Hard segue lo stesso andamento della nostra Entropia)
  - Variazione di Entropia nei vari algoritmi

2 STRUMENTI DI VALUTAZIONE

Utilizzeremo i nostri script Python per la generazione delle chiavi nei vari algoritmi, il calcolo dei tempi necessari, l'entropia e tutto ciò che riguarda i dati statistici rilevanti.

Abbiamo integrato l'utilizzo di OpenSSL all'interno degli script per la generazione delle chiavi nei diversi algoritmi. Tutte le chiavi generate sono disponibili in ipotesi di un futuro approfondimento sugli algoritmi e sono archiviate all'interno della cartella github del progetto.

Per la generazione della stringa di 100Mb utilizzeremo Butterfly all'interno dell'ambiente Guile studiato durante il corso.

Successivamente effettueremo i test di DieHard e li confronteremo con i dati da noi calcolati.

3 OBIETTIVO

Una volta effettuate tutte le misure e derivati i vari dati statistici, faremo una nostra scelta in uno scenario aziendale in cui bisogna decidere l'algoritmo da adottare in base alle necessità. Indicheremo quindi quali algoritmi si adattano a quali richieste e motiveremo le nostre scelte sulla base delle informazioni ottenute durante lo studio degli algoritmi.

ALGORITMI DI CIFRATURA SIMMETRICA

1 INTRODUZIONE

L'obiettivo della crittografia simmetrica è quella di costruire sistemi in cui due parti possano scambiarsi dati segretamente, essendosi però già scambiati (o accordati) qualche informazione precedentemente. Questa informazione sulla quale le due parti devono essere già a conoscenza prende il nome di *private-key.*
La *private-key* sarà usata sia per convertire il plaintext in *ciphertext*, che per riottenere il testo in chiaro da quello criptato.

Uno schema crittografico è composto da 3 algoritmi:

- **Algoritmo di generazione della chiave GEN:**
  E' un algoritmo probabilistico che torna in output una chiave in accordo con qualche distribuzione di probabilità.

- **Algoritmo di criptazione ENC:**
  Prende in input una chiave *k*, un messaggio *m* e torna in output un chipertext *c.*
  Piu formalmente :
  $$Enc_k(m) = c$$

- **Algoritmo di decriptazione DEC:**
  Prende in input una chiave *k* e un ciphertext *c* e torna in output un plaintext *m.*
  Piu formalmente:
  $$Dec_k\ (c) = m$$

La correttezza richiesta è che quindi sia verificata la seguente:

$$Dec_k\ \ (Enc_k(m) = c) = m$$

Un importante principio alla base della crittografia può essere riassunto:

Non deve essere necessario che lo schema di crittografia delle informazioni sia segreto, questo deve poter cadere nelle mani di chiunque e garantire lo stesso un canale di comunicazione sicuro tra due parti.

2 AES

In crittografia, l'**Advanced Encryption Standard** (**AES**),è un algoritmo di cifratura a blocchi. AES è veloce sia se sviluppato in software sia se sviluppato in hardware; è relativamente semplice da implementare, richiede poca memoria ed offre un buon livello di protezione/sicurezza, motivi che complessivamente l'hanno preferito agli altri algoritmi proposti.
Nell'AES il blocco è invece di dimensione fissa (128 bit) e la chiave può essere di 128, 192 o 256 bit

La AES è basato su una **rete a sostituzione e permutazione**, il che significa che una serie di operazioni matematiche crea dati altamente modificati . L'input è un testo semplice e la chiave usata per pilotare le operazioni. Queste possono essere semplici come una rotazione a livello di singoli bit (bitwise rotation), XOR (exclusive OR) o piu complesse. Siccome un singolo passaggio dovrebbe essere semplice da decifrare, tutte le moderne tecnologie di codifica **lavorano a più riprese**. I cicli AES si ripetono per 10, 12 o 14 volte per AES-128, AES-192 e AES-256. Le chiavi AES sopportano anche lo stesso processo dei dati utente, che vengono rovesciati in una chiave mutevole.

Molti dibattiti accademici nel settore della sicurezza ruotano attorno alle cosiddette violazioni: ci riferisce a metodi di violazione diversi dalla ricerca "brute-force" per la ricerca della chiave di accesso. Si è parlato e si parla molto di attacchi XSL e legati alla chiave, ma il successo non è ancora arrivato. L'unica via sfruttabile per far breccia nella codifica AES è un attacco di tipo "side channel".
Questo richiede che l'attacco avvenga sullo stesso sistema sul quale la codifica AES è eseguita, e bisogna trovare un modo per ottenere le informazioni di cache timing. In questo caso, è possibile tracciare il numero di cicli macchina fino a che il processo di crittografia è completato.

3 BLOWFISH

In crittografia, Blowfish è un algoritmo a chiave simmetrica a blocchi, ideato nel 1993 da Bruce Schneier. Sebbene tutt'oggi non sia reperibile una crittanalisi di Blowfish, questo

algoritmo sta suscitando nuovamente interesse se implementato con una maggior dimensione dei blocchi, come nel caso di AES o Twofish.

Schneier progettò blowfish per essere un algoritmo di utilizzo generale, utile a rimpiazzare l'allora decadente Data Encryption Standard (DES) e libero da problemi caratteristici di altri algoritmi. All'epoca molti altri sistemi di cifratura erano proprietari, coperti da brevetto o da segreti governativi.

Due delle caratteristiche di rilievo di blowfish sono S-Box dipendenti dalla chiave, e una lista di chiavi estremamente complessa.

La maggiore velocità è ottenuta attraverso la rimozione di ogni permutazione (operazione che non introduce alcun ritardo se realizzata via hardware, ma abbastanza dispendiosa se realizzata via software).

L'algoritmo è composto da 16 round, in ognuno dei quali si utilizza una sottochiave diversa ki, ed un round finale in cui si prevede l'utilizzo di due sottochiavi k17 e k18. Viene utilizzata una funzione F per ciascun round, risultante dalla combinazione delle uscite di 4 S-Box.

Le operazioni utilizzate in Blowfish sono unicamente: addizioni modulo 2^{32} e XOR.

ALGORITMI DI CIFRATURA ASIMMETRICA

1 INTRODUZIONE

La **crittografia asimmetrica**, conosciuta anche come **crittografia a chiave pubblica/privata** , è un sistema crittografico in cui, come si deduce dal nome, ad ogni attore coinvolto nella comunicazione è associata una coppia di chiavi:

- La chiave pubblica, che deve essere condivisa;
- La chiave privata, appunto personale e segreta;

Si può così evitare il collo di bottiglia della crittografia simmetrica, cioè la condivisione tra le due parti dell'unica chiave utile alla cifratura/decifratura. Il meccanismo si basa sul fatto che, se con una delle due chiavi sì cifra un messaggio, allora quest'ultimo sarà decifrato solo con l'altra.

2 RSA

L'RSA è il sistema crittografico piu usato al giorno d' oggi.
Per ottenere una discreta sicurezza è necessario utilizzare chiavi binarie di almeno 2048 bit. Quelle a 512 bit sono ricavabili in poche ore. Le chiavi a 1024 bit, ancora oggi largamente utilizzate, non sono più consigliabili.La fattorizzazione di interi grandi, infatti, è progredita rapidamente mediante l'utilizzo di hardware dedicati, al punto che potrebbe essere possibile fattorizzare un intero di 1024 bit in un solo anno di tempo, al costo di un milione di dollari (un costo sostenibile per qualunque grande organizzazione, agenzia o intelligence).

RSA è basato sull' elevata complessità computazionale della fattorizzazione in numeri primi. Il suo funzionamento base è il seguente:

1. Si scelgono a caso due numeri primi p e q abbastanza grandi da garantire la sicurezza dell'algoritmo
2. Si calcola il loro prodotto $n = pq$, chiamato *modulo* , e il prodotto $\varphi(n) = (p-1)(q-1)$
3. Si considera che la fattorizzazione di n è segreta e solo chi sceglie i due numeri primi, p e q la conosce.
4. Si sceglie poi un numero e (esponente pubblico), *coprimo* con $\varphi(n)$ e più piccolo di $\varphi(n)$.
5. Si calcola il numero d (esponente privato) tale che il suo prodotto con e sia congruo a 1 modulo $\varphi(n)$ ovvero $ed = 1 \pmod{\varphi(n)}$.

La chiave pubblica è (n,e) mentre la chiave privata (n,d).
La forza dell'algoritmo sta nel fatto che per calcolare d da e (o viceversa) non basta la conoscenza di n ma serve il numero $\varphi(n)=(p-1)(q-1)$, e che il suo calcolo richiede tempi molto elevati , infatti *fattorizzare* in numeri primi è un'operazione computazionalmente molto costosa.

Un messaggio *m* viene cifrato attraverso l'operazione $m^c (mod\ n)$ trasformandolo nel messaggio cifrato *c.* Una volta trasmesso, *c* viene decifrato con $c^d (mod\ n) = m$

La **firma digitale** non è altro che l'inverso: il messaggio viene crittografato con la chiave privata, in modo che chiunque possa, utilizzando la chiave pubblica conosciuta da tutti, decifrarlo e, oltre a poterlo leggere in chiaro, essere certo che il messaggio è stato mandato dal possessore della chiave privata corrispondente a quella pubblica utilizzata per leggerlo. Per motivi di efficienza e comodità, normalmente viene inviato il messaggio in chiaro con allegata la firma digitale di un hash del messaggio stesso; in questo modo il ricevente può direttamente leggere il messaggio (che è in chiaro), utilizzare la chiave pubblica per estrarre l'hash dalla firma e verificare che questo sia uguale a quello calcolato localmente sul messaggio ricevuto. Se l'hash utilizzato è crittograficamente sicuro, la corrispondenza dei due valori conferma che il messaggio ricevuto è identico a quello originariamente firmato e trasmesso.

3 ECC (Elliptic Curve Cryptography)

L'algoritmo ECC risale al 1976 ed è quindi uno dei più antichi algoritmi a chiave pubblica. L'algoritmo è particolarmente adatto alla generazione di una chiave segreta tra due corrispondenti che comunicano attraverso un canale non sicuro (pubblico). La sua sicurezza si basa sulla complessità computazionale del calcolo del logaritmo discreto.

Supponiamo di avere i soliti Alice e Bob che vogliono scambiarsi una chiave segreta in modo sicuro.

1. Alice genera e comunica pubblicamente un numero primi *N* molto elevato (p.es. 1024 bit, circa 300 cifre decimali) e un generatore **g** (che esiste sicuramente essendo N primo); si potrebbero anche usare due numeri *N*,*g* pubblicati da un qualche ente pubblico.
2. Alice genera un numero casuale *a* < *N* e calcola $A = g^a\ (mod\ n)$, ed invia A a Bob. A questo punto *N,g,A,* sono pubblici *a*, è noto solo ad Alice
3. Bob genera un numero casuale *b* < *N* e calcola $B = g^a\ (mod$ N) , ed invia *B* ad Alice. A questo punto *N,g,A,B* sono pubblici, *a* è noto solo ad Alice , *b* solo a Bob.
4. Alice calcola $k = B^a\ (mod\ n)$
5. Alce calcola $k = A^b\ (mod\ n)$

Inutile aggiungere che i due numeri k sono uguali! Infatti entrambe valgono $g^{ab}\ (mod\ n)$. A questo punto Alice e Bob possono usare *k* come chiave per comunicare con un cifrario simmetrico.

La cosa importante per la sicurezza è che un terzo che intercettasse i quattro numeri *N,g,A,B* non sarebbe in grado di ottenere $k = g^{ab} (mod\ n)$non conoscendo né *a* né

b. In realtà basterebbe calcolarli $a = log_g A$ e $b = log_g B$ ma il calcolo del logaritmo discreto è computazionalmente proibitivo per numeri molto grandi (1024 bit e oltre).

ENTROPIA

1 DEFINIZIONE

Nella teoria dell'informazione l'entropia di una sorgente di messaggi è l'informazione media contenuta in ogni messaggio emesso. L'entropia nella crittografia viene utilizzata come misura di riferimento per la robustezza degli algoritmi di cifratura. L'informazione contenuta in un messaggio è tanto più grande quanto meno probabile era. Un messaggio scontato, che ha un'alta probabilità di essere emesso dalla sorgente contiene poca informazione, mentre un messaggio inaspettato, poco probabile contiene una grande quantità di informazione. L'entropia di una sorgente risponde a domande come: qual è il numero minimo di bit che servono per memorizzare in media un messaggio della sorgente? Quanto sono prevedibili i messaggi emessi dalla sorgente?
Una sequenza di lettere come aaaaaaaa possiede meno entropia di una parola come alfabeto la quale a sua volta possiede ancora meno entropia di una stringa completamente casuale come j3s0vek3. L'entropia può essere vista come la casualità contenuta in una stringa, ed è strettamente collegata al numero minimo di bit necessari per rappresentarla senza errori.
Sia x una variabile aleatoria con distribuzione di probabilità p1,...,pn si ha:

$$\boldsymbol{H(x) = -\sum_{i=1}^{n} p(i)\, log(p(i))}$$

CONFRONTO TRA GLI ALGORITMI DI CIFRATURA

INTRODUZIONE

Per verificare l'efficienza dei nostri algoritmi in termine di tempi di generazione delle chiavi , cifratura, decifratura, ed entropia dei messaggi risultanti, ci siamo forniti di file di testo di varie dimensioni i quali sono stati poi criptati e decriptati utilizzando gli stessi.
I file di testo dati in input sono stati generati utilizzando il generatore butterfly nell'ambiente **GUILE** come segue:

```
(define p (make-butterfly "aaa" (mtfa-num-to-bv (mtfa-strong-random 32000))))
(mtfa - bv - to - num(p 'get 8))
Samir Salman, [07.02.20 11:46]
(define (write-to-file path string-list)
(call -
with-output - file path
( lambda (output-port)
(write output-port))))



(define p (make-butterfly "aaa" (mtfa-num-to-bv (mtfa-strong-random 32000))))


(call-with-output-file "file.txt"
( lambda (output-port)
(display  (mtfa-bv-to-num (p 'get 100000000)) output-port)))
```

Le dimensioni dei file generati sono di 10,50 e 100 MB.
Inoltre per vedere le differenze rispetto ad altri file non creati utilizzando lo stesso random generator abbiamo provato a cifrare anche un file uniforme (sequenze di "1").

1 CONFRONTO AES 256 VS BLOWFISH

Abbiamo analizzato il comportamento dei due algoritmi al variare dei diversi fattori, tra cui ad esempio: lunghezza del messaggio, numero di bits delle chiavi ecc..

IL COMPORTAMENTO DI AES

Per generare chiavi simmetriche AES e per cifrare e decifrare dei file di testo usando le stesse, abbiamo utilizzato il software OpenSSL.

In particolare il comando di cui abbiamo fatto utilizzo per la generazione di chiavi è:

```
os.system("openssl enc -aes-256-cbc -nosalt -pbkdf2 -k randomString -P > keys/key.pem")
```

Mentre i successivi due comandi sono stati utilizzati rispettivamente per criptare e decriptare file passati in input:

1. ```
   os.system("openssl enc -nosalt -aes-256-cbc -in plainText -out crypter_files_time/encrypted.txt -K key ")
   ```

2. ```
   subprocess.check_output("openssl enc -aes-256-cbc -d -in encryptedFile  -K key ", shell=True)
   ```

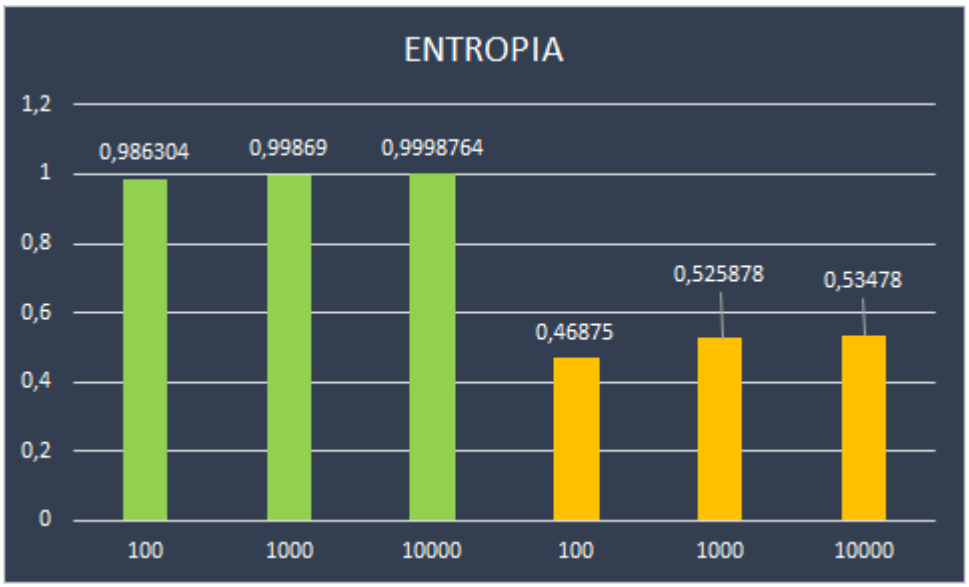
Abbiamo effettuato diverse prove sull'algoritmo di cifratura AES, di seguito elenchiamo la tabella dei risultati ottenuti.

VALORE	RISULTATO	RISULTATO	RISULTATO	RISULTATO	RISULTATO	RISULTATO
NUMERO BIT	256	256	256	256	256	256
NUMERO CHIAVI GENERATE	100	1000	10000	100	1000	10000
TEMPO TOTALE (Sec.)	6,539	57,157590	695,84624000	5,338739	54,129300	680,129600
AVG TEMPO (Sec.)	0,06539	0,05715759	0,069584	0,05338739	0,0541293	0,068012962
LUNGHEZZA CHIAVE CONCATENATA (Caratteri)	3200	32000	320000	3200	32000	320000
ENTROPIA	0,986304	0,99869	0,9998764	0,46875	0,525878	0,53478
GENERATORE	/DEV	/DEV	/DEV	UNI	UNI	UNI

Nella precedente tabella abbiamo quindi i vari valori di:

1. **NUMERO BIT:** Numero di bit delle chiavi generate
2. **NUMERO CHIAVI GENERATE:** Numero di chiavi generate
3. **TEMPO TOTALE:** Tempo per la generazione di N chiave simmetriche
4. **AVG TEMPO**: Media del tempo
5. **LUNGHEZZA CHIAVE CONCATENATA**: Lunghezza in caratteri della chiave risultante dalla concatenazione di tutte le N chiavi generate
6. **ENTROPIA**: Entropia calcolata sulla chiavi concatenate
7. **GENERATORE**: Il generatore scelto per la generazione della passkey
    a. /DEV è il generatore del sistema operativo Linux
    b. UNI passkey generata da una stringa uniforme

Possiamo osservare quindi che il tempo medio di generazione di ogni chiave è circa di 0,06s. Per quanto riguarda il generatore **/DEV** l'entropia aumenta all'aumentare del numero di chiavi, su 10.000 chiavi il valore dell'entropia della chiave concatenata è di 0,9998764. Tutti i dati sono derivanti dall'algoritmo AES a 256 bit. Per il generatore **UNI** il valore dell'entropia rimane molto basso, cresce lievemente all'aumentare delle chiavi, questo a favore del tempo medio di generazione delle chiavi che si aggira intorno ai 0.04 s.
Di seguito dei grafici che analizzano il comportamento dell'algoritmo AES.



In questo grafico osserviamo la variazione di entropia in base al numero delle chiavi generate, utilizzando il generatore /DEV (in verde) e la stringa uniforme (in arancione)

In questo grafico osserviamo il tempo totale per la generazione delle chiavi nel caso del generatore /DEV in blu e con la stringa uniforme in arancione

IL COMPORTAMENTO DI BLOWFISH

Per generare chiavi simmetriche Blowfish, e per cifrare e decifrare dei file di testo usando le stesse, abbiamo utilizzato il software OpenSSL.
In particolare il comando di cui abbiamo fatto utilizzo per la generazione di chiavi è:

```
os.system("openssl enc -bf -salt  -K key -iv iv " + randomString + " -base64 -P  > keys/key.pem")
```

Mentre i successivi due comandi sono stati utilizzati rispettivamente per criptare e decriptare file passati in input:

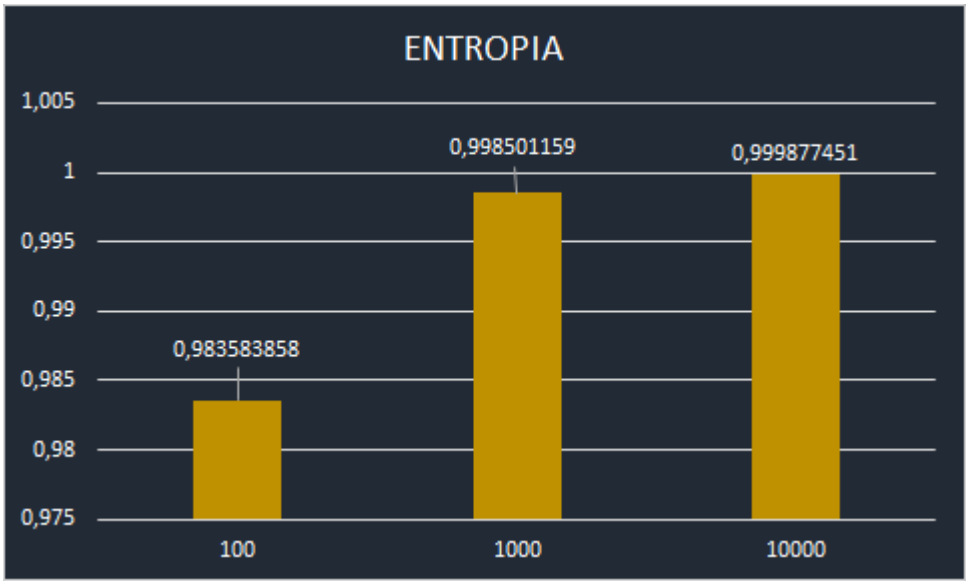
1. `os.system("openssl enc -salt -bf -in plainText -out crypter_files_time/encrypted.txt -k key ")`

2. `subprocess.check_output("openssl enc -bf -d -in ecryptedFile  -K key -iv iv ", shell=True)`

Abbiamo effettuato diverse prove sull'algoritmo di cifratura BLOWFISH, di seguito elenchiamo la tabella dei risultati ottenuti.

VALORE	RISULTATO	RISULTATO	RISULTATO
NUMERO BIT	1024	1024	1024
NUMERO CHIAVI GENERATE	100	1000	10000
TEMPO TOTALE (Sec.)	6,848962	58,658835	625,07080000
AVG TEMPO (Sec.)	0,06848962	0,058658835	0,06250708
LUNGHEZZA CHIAVE CONCATENATA (Caratteri)	3200	32000	320000
ENTROPIA	0,983583858	0,998501159	0,999877451

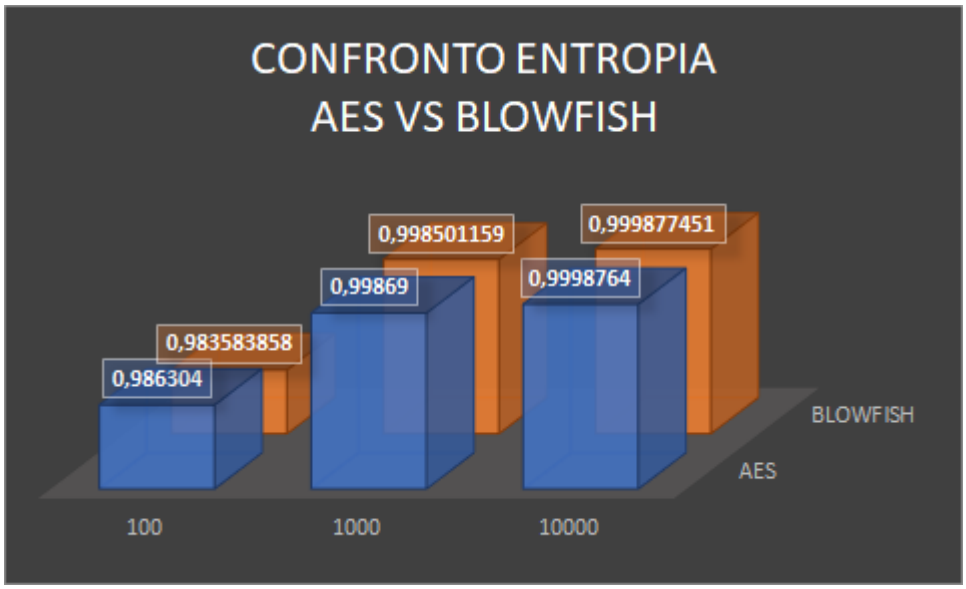
Possiamo osservare quindi che il tempo medio di generazione di ogni chiave è circa di 0,06s. L'entropia delle chiavi aumenta all'aumentare del numero di chiavi, su 10.000 chiavi il valore dell'entropia della chiave concatenata è di 0,999877451. Tutti i dati sono derivanti dall'algoritmo BLOWFISH a 1024 bit.



In questo grafico osserviamo il comportamento dell'entropia delle chiavi generate tramite l'algoritmo BLOWFISH, all'aumentare del numero di chiavi generate.

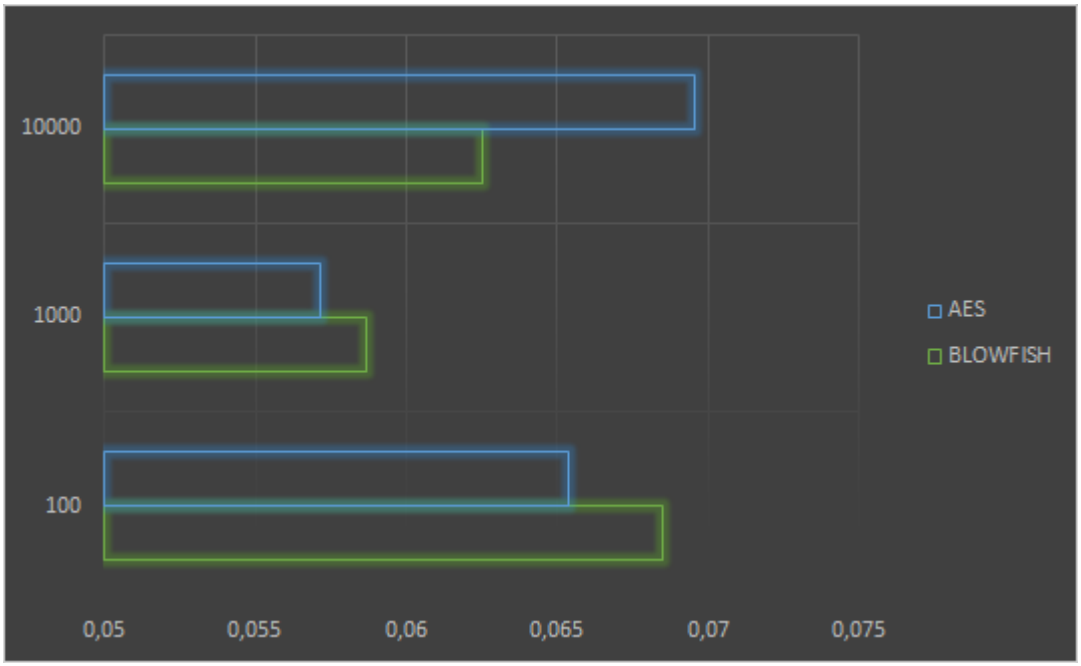
In questo grafico osserviamo il tempo medio impiegato per generare le chiavi tramite l'algoritmo BLOWFISH, all'aumentare del numero di chiavi generate.

CONFRONTO DATI TRA AES E BLOWFISH



In questo grafico confrontiamo il comportamento dell'entropia derivata dagli algoritmi BLOWFISH (rappresentata in arancione) e AES (in blu), all'aumentare del numero di chiavi generate.,

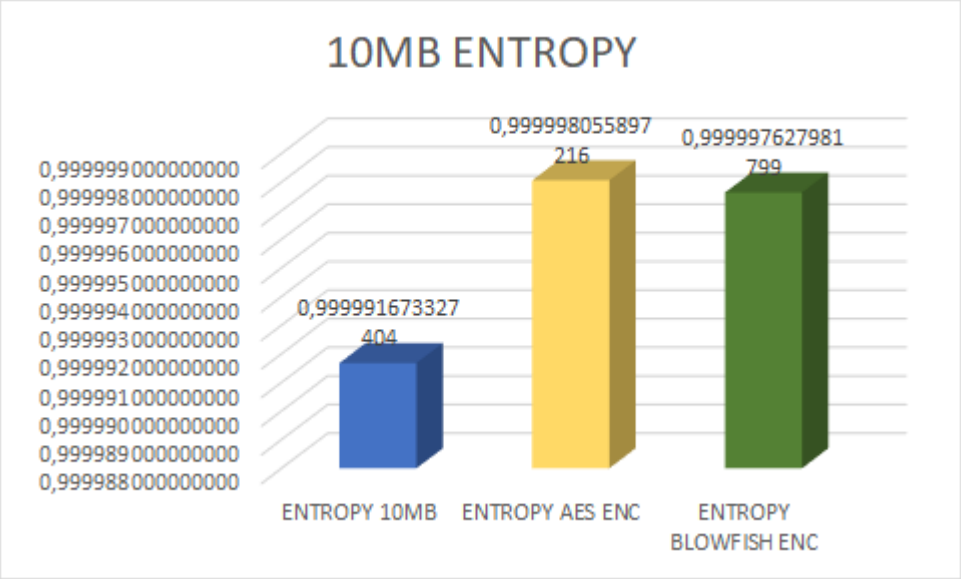
Come osserviamo dal grafico, l'entropia della concatenazione delle chiavi generata con Blowfish è lievemente più alta di quella delle chiavi generate con AES. È davvero interessante però osservare la differenza minima tra i due algoritmi, che indicano una forte validità di entrambi. **Il collo di bottiglia degli algoritmi a chiave simmetrica resta comunque lo scambio delle chiavi tra i due endpoint.**



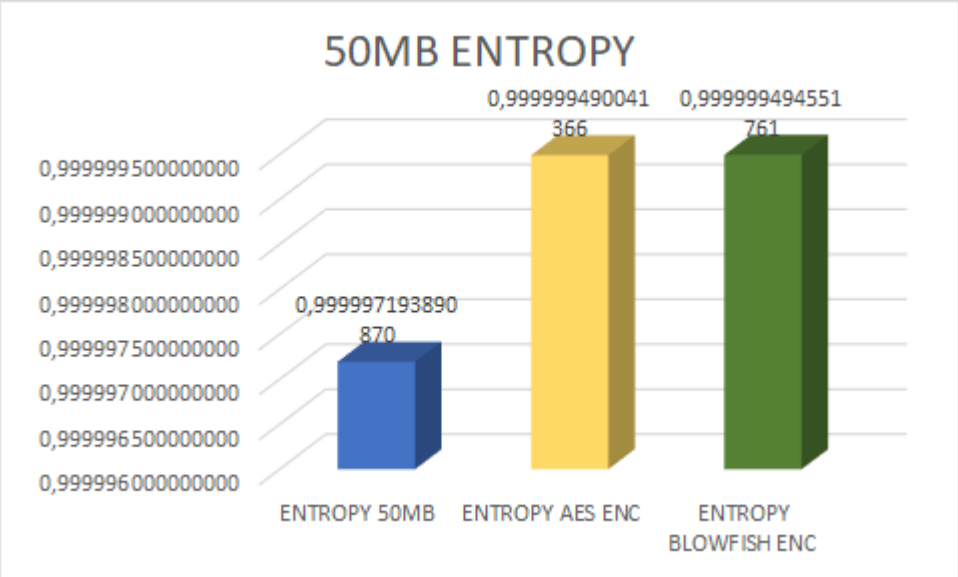
In questo grafico confrontiamo il tempo medio impiegato dagli algoritmi AES e BLOWFISH per la generazione rispettivamente di 100, 1000 e 10.000 chiavi.

Il tempo medio per la generazione delle chiavi restituisce invece dei risultati curiosi. Mentre AES come prevedibile aumenta leggermente il tempo medio per la generazione delle chiavi al crescere del numero di chiavi generate, Blowfish tende a diminuire il tempo medio per la generazione delle chiavi all'aumentare del numero di chiavi generate.

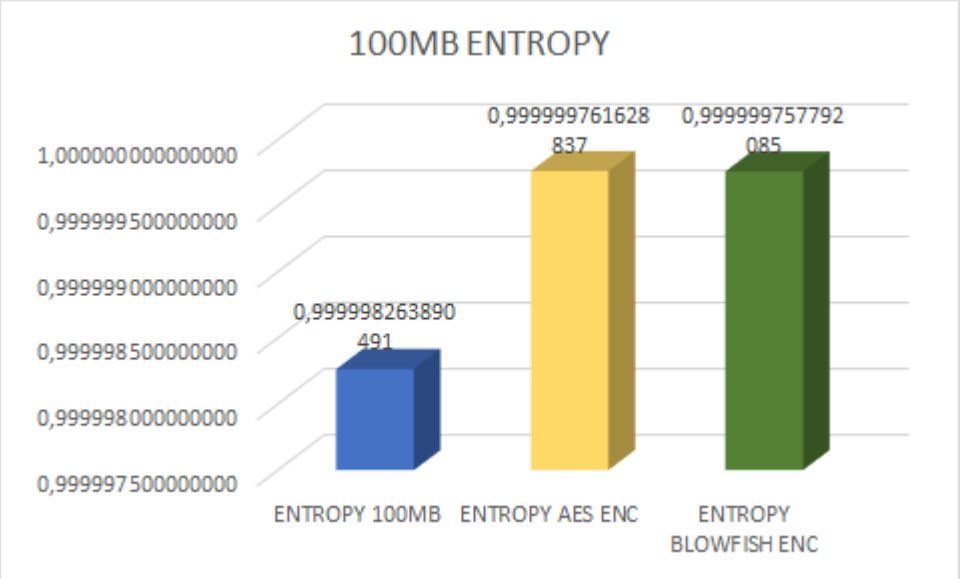
Da questi confronti deriviamo che entrambi gli algoritmi garantiscono una forte entropia sulle chiavi generate, Blowfish si distingue da AES per quanto riguarda il tempo medio di generazione delle chiavi. Osserviamo ora il comportamento nella cifratura di file di diverse dimensioni **(10,50,100 MB)**.



Osserviamo che il file da 10MB aumenta entropia, seppur già alta, dopo la cifratura attraverso entrambi gli algoritmi. Sottolineiamo il fatto che il file cifrato mediante AES ha un'entropia leggermente maggiore rispetto a quello cifrato con Blowfish. Proseguiamo con le nostre osservazioni, aumentando la dimensione del file a 50MB.

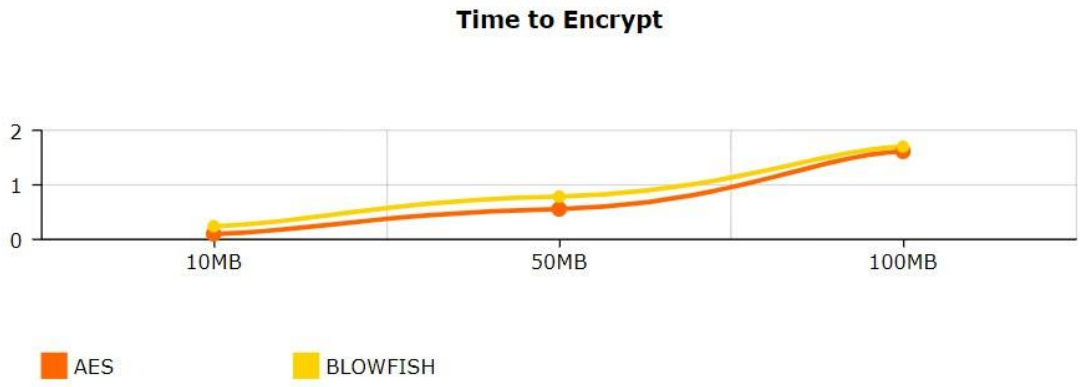


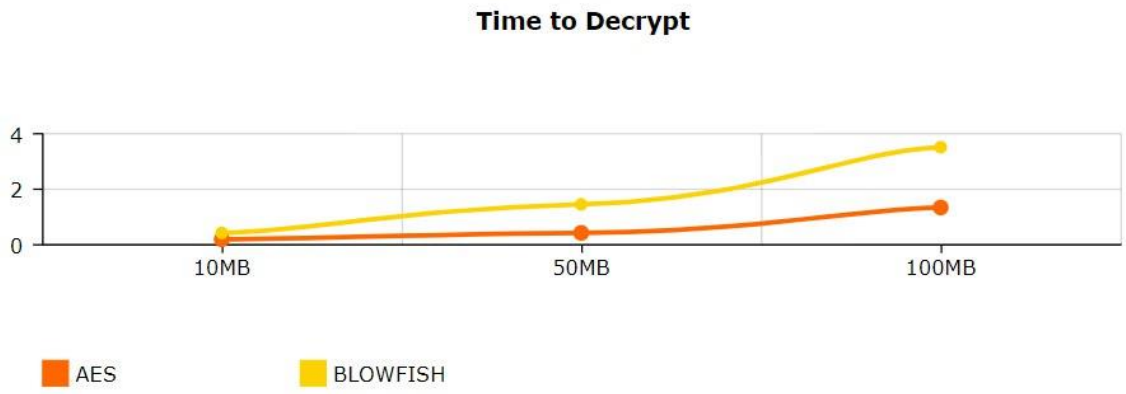
Nel caso del file da 50MB l'entropia del file cifrato con Blowfish è leggermente è più alta di quella del file cifrato con AES. Osserviamo che la differenza tra le due entropie è sempre dell'ordine di 10^{-7}.



In ultima analisi analizziamo i valori delle entropie sugli algoritmi simmetrici riguardo ad un file criptato di 100MB che riteniamo maggiormente valido ai fini del confronto.
Notiamo che l'entropia del file in chiaro era già molto elevata poiché il file era stato generato con un random generator molto buono (*butterfly*).
In seguito alla cifratura del file l'entropia aumenta sia con l'algoritmo AES che con il BLOWFISH.
Sebbene molto simili possiamo notare una leggera differenza all'ottava cifra decimale, e siamo quindi in grado di affermare che l'entropia del file criptato con AES è superiore, sebbene di poco di quella del file criptato con BLOWFISH.

Osserviamo ora i dati relativi ai tempi di cifratura e decifratura dei files.





Mentre per la generazione delle chiavi l'algoritmo Blowfish era in modo rilevante più veloce di AES, vediamo dai grafici che nei tempi di cifratura e decifratura vince sempre AES, molto più veloce, soprattutto nel caso della decifratura. Questo è un fattore molto importante, tenendo conto dell'equilibrio nel caso dell'entropia.

CONCLUSIONI

Tenendo conto dell'equilibrio per quanto riguarda l'entropia nei file criptati, spostiamo l'attenzione sui tempi per la cifratura e decifratura dei file. Supponendo di lavorare con file di dimensione maggiore, i tempi sono un fattore molto importante, che fa la differenza visti i risultati. Concludiamo quindi sottolineando il forte equilibrio nelle entropie, sia per quanto riguarda la generazione delle chiavi, che per la cifratura dei file, ma il profondo distacco nei tempi. Valuteremo successivamente quale tra i due algoritmi riteniamo più valido e in quali scenari.

2 CONFRONTO RSA 2048 VS ECC 320

Abbiamo analizzato il comportamento dei due algoritmi al variare dei diversi fattori, tra cui ad esempio: lunghezza del messaggio, numero di bits delle chiavi ecc..

IL COMPORTAMENTO DI RSA

Per generare chiavi asimmetriche RSA e per cifrare e decifrare dei file di testo usando le stesse, abbiamo utilizzato il software OpenSSL.
In particolare il comando di cui abbiamo fatto utilizzo per la generazione di chiavi private è:

```
os.system('openssl genrsa -out privateKey 1024')
```

Mentre il comando utilizzato per la generazione di chiavi pubbliche è:

```
os.system('openssl rsa -in privKey -pubout -out pubKey')
```

Vediamo ora rispettivamente i comandi di cui abbiamo ausilio per la cifratura e la decifratura dei messaggi:

1. ```
   subprocess.Popen("dd count=1 skip=" + str(index) + " if=" + large_file_path + " bs=245 | openssl rsautl -encrypt -inkey " + pub_key + " -pubin", shell=True, stdout=subprocess.PIPE).stdout
   ```

2. ```
   os.popen("dd count=1 skip=" + str(index) + " if=" + crypt_file_path + " bs=256 | openssl rsautl -decrypt -inkey " + pvt_key)
   ```

Vogliamo qui ricordare che quando ci occupiamo di cifrare e decifrare dei messaggi, abbiamo dei limiti, in quanto il messaggio che intendiamo criptare non deve essere più lungo della lunghezza della chiave utilizzata.
Quindi ad esempio per una coppia di chiavi RSA a 1024 bit, il messaggio deve essere lungo al più 1024 bit.
Per poter analizzare ed estrarre statistiche riguardo l'utilizzo degli algoritmi di cifratura su file di dimensione maggiore (10,50,100 MB) ci siamo serviti di un artificio.
Il messaggio in chiaro è stato suddiviso a blocchi di *n* bit, dove *n* è il numero di bit della coppia di chiavi RSA, e così abbiamo potuto cifrare ogni blocco singolarmente, e alla fine concatenare tutti i blocchi così cifrati per ottenere il risultato voluto.
Stesso meccanismo vale per la decifratura del file, ovviamente con questo metodo il tempo di cifratura e decifratura aumenta notevolmente, ma siamo così in grado di utilizzare questo algoritmo che garantisce alta entropia anche su file di grande dimensione.

Il metodo appena presentato è stato molto utile al fine di poter confrontare gli algoritmi RSA e ECL.
Dobbiamo però dire che nell'uso comune RSA è usato in modo ibrido con la crittografia simmetrica di AES, in questo modo si è in grado di cifrare messaggi lunghi e garantire un elevata entropia e velocità di cifratura.

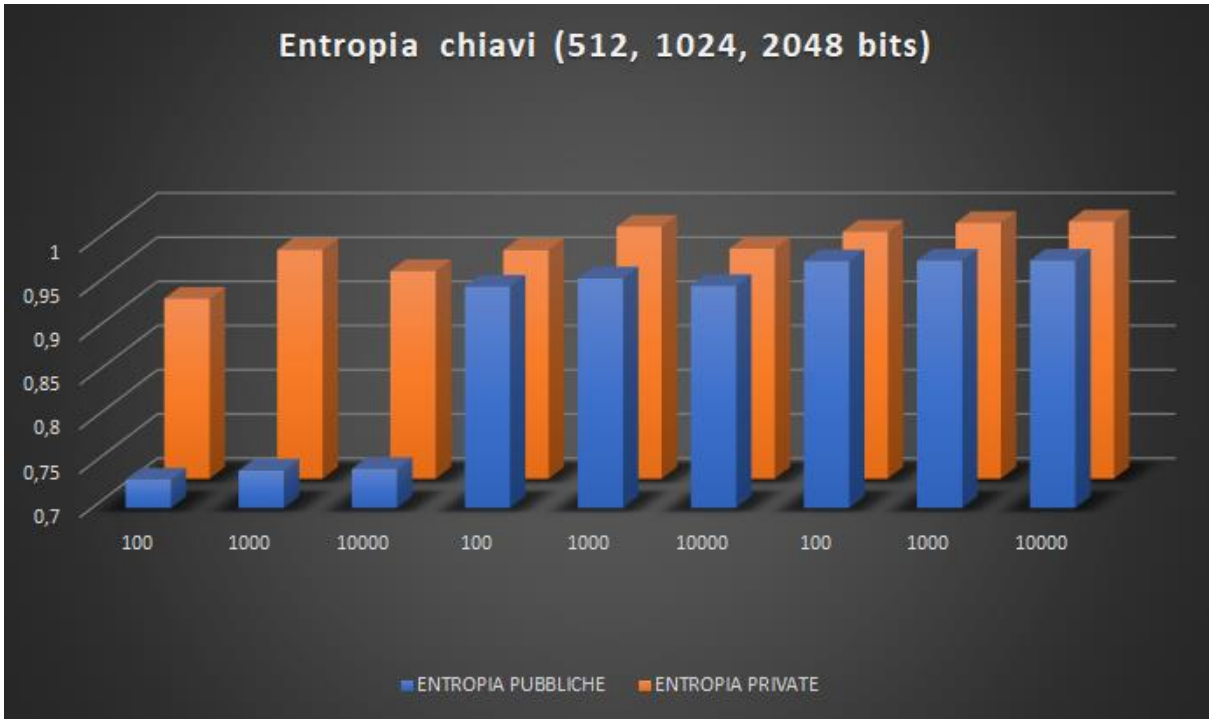
Abbiamo effettuato diverse prove sull'algoritmo di cifratura RSA, di seguito elenchiamo la tabella dei risultati ottenuti.

VALORE	RISULTATO	RISULTATO	RISULTATO
NUMERO BIT	512	512	512
NUMERO CHIAVI GENERATE	100	1000	10000
TEMPO TOTALE PRIVATE (Sec.)	7,42014	65,182070	658,86139000
TEMPO TOTALE PUBBLICHE (Sec.)	6,52972	56,704000	571,439650
AVG TEMPO PRIVATE (Sec.)	0,0742	0,065182	0,065886
AVG TEMPO PUBBLICHE (Sec.)	0,065297	0,056704	0,0571439
LUNGHEZZA CHIAVE CONCATENATA PUB (Caratteri)	12800	128000	14400000
LUNGHEZZA CHIAVE CONCATENATA PVT (Caratteri)	42632	426320	13200000
ENTROPIA PUBBLICHE	0,73224	0,742363	0,7440188
ENTROPIA PRIVATE	0,903494	0,9587122	0,9346333

VALORE	RISULTATO	RISULTATO	RISULTATO
NUMERO BIT	1024	1024	1024
NUMERO CHIAVI GENERATE	100	1000	10000
TEMPO TOTALE PRIVATE (Sec.)	9,5446	78,061447	847,066920
TEMPO TOTALE PUBBLICHE (Sec.)	6,81089	56,458631	602,083239
AVG TEMPO PRIVATE (Sec.)	0,09544	0,07806	0,084706692
AVG TEMPO PUBBLICHE (Sec.)	0,068108	0,056458	0,060208324
LUNGHEZZA CHIAVE CONCATENATA PUB (Caratteri)	81276	812760	8127600
LUNGHEZZA CHIAVE CONCATENATA PVT (Caratteri)	21600	216000	2160000
ENTROPIA PUBBLICHE	0,9504156	0,959782	0,95113753
ENTROPIA PRIVATE	0,9584752	0,984976	0,960092744

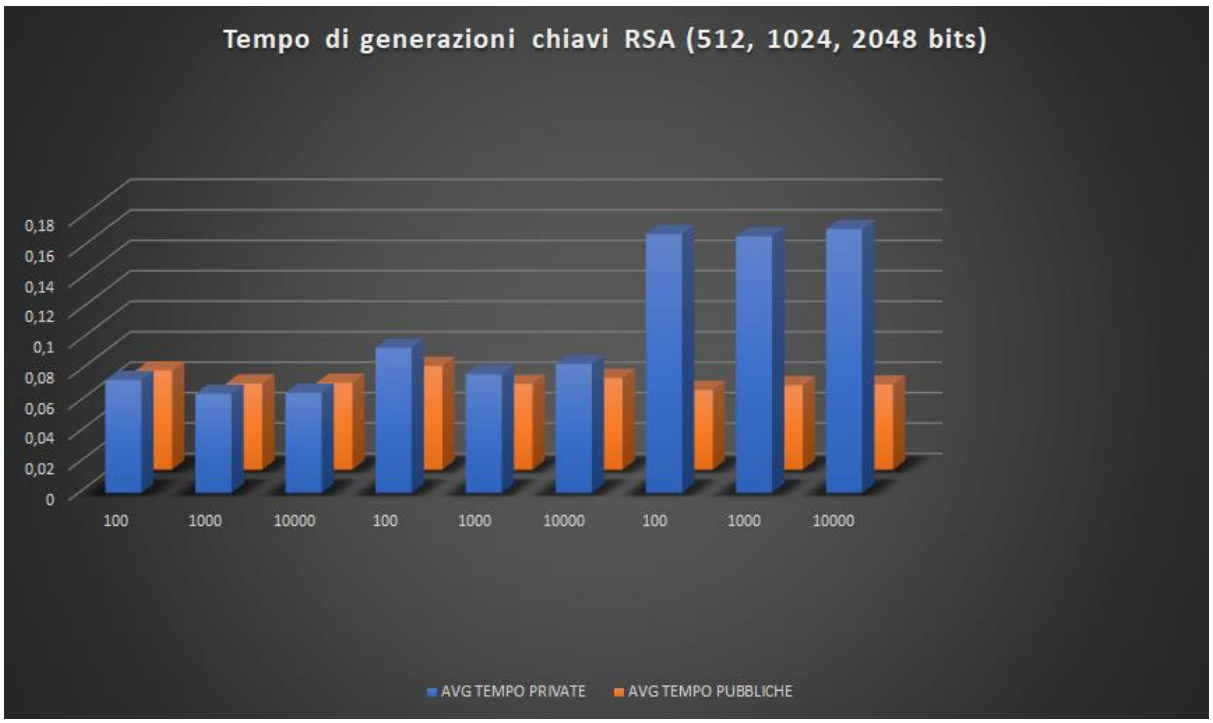
VALORE	RISULTATO	RISULTATO	RISULTATO
NUMERO BIT	2048	2048	2048
NUMERO CHIAVI GENERATE	100	1000	10000
TEMPO TOTALE PRIVATE (Sec.)	17,00781	168,539710	1.733,553900
TEMPO TOTALE PUBBLICHE (Sec.)	5,23311	55,484750	559,928600
AVG TEMPO PRIVATE (Sec.)	0,17007	0,16853	0,17335
AVG TEMPO PUBBLICHE (Sec.)	0,05233	0,055484	0,055992
LUNGHEZZA CHIAVE CONCATENATA PUB (Caratteri)	39200	392000	14400000
LUNGHEZZA CHIAVE CONCATENATA PVT (Caratteri)	159020	1960000	19600000
ENTROPIA PUBBLICHE	0,97913	0,97985	0,979866
ENTROPIA PRIVATE	0,979134	0,989	0,990746

Dalle tabelle osserviamo che il tempo medio per la generazione delle chiavi private si aggira intorno agli 0,08 s in media, tranne nel caso di RSA 2048 bits, in cui i tempi medi per le chiavi private sono in media 0.17s. Per quanto riguarda le chiavi pubbliche invece i tempi rimangono in un intorno di 0.06s. Osserviamo che i tempi di generazione delle chiavi sono molto simili a quelli degli algoritmi simmetrici, aspetto molto positivo.
Vediamo ora i grafici che analizzano il comportamento dell'entropia nelle chiavi di RSA rispettivamente per chiavi a 512,1024 e 2048 bits.



In questo grafico osserviamo il comportamento dell'entropia delle chiavi generate tramite l'algoritmo RSA, all'aumentare del numero di chiavi generate e in base al numero di bit della chiave.

Notiamo il vertiginoso aumento di entropia delle chiavi pubbliche all'aumentare del numero di bits, questo sottolinea che un RSA con chiavi a 512 è di molto differente ad un RSA con chiavi di dimensioni >= 1024 bits. Sulle 10.000 chiavi è evidente l'aumento dell'entropia sulle chiavi a 2048 bits, che confermano l'importanza non sempre scontata del numero di bits di cui è costituita una chiave.



In questo grafico confrontiamo il tempo medio impiegato dall'algoritmo RSA rispettivamente a 512, 1024, 2048 bits per la generazione di 100, 1000 e 10.000 chiavi

Dal grafico non emergono informazioni rilevanti, possiamo osservare un picco nei tempi medi di generazione delle chiavi private quando la lunghezza delle chiavi è di 2048 bits. La differenza tra i tempi tra le chiavi a 1024 bits e quelle a 2048 bits, segno che sacrifichiamo il fattore tempo in favore di una maggiore entropia nella generazione delle chiavi.

IL COMPORTAMENTO DI ECC 320

Per generare chiavi asimmetriche ECC e per cifrare e decifrare dei file di testo usando le stesse, abbiamo utilizzato il software OpenSSL.
In particolare il comando di cui abbiamo fatto utilizzo per la generazione di chiavi private è:

```
os.system('openssl genrsa -out privateKey 1024')
```

Mentre il comando utilizzato per la generazione di chiavi pubbliche è:

```
os.system('openssl rsa -in privKey -pubout -out pubKey')
```

Vediamo ora rispettivamente i comandi di cui abbiamo ausilio per la cifratura e la decifratura dei messaggi:

3.
```
subprocess.Popen("dd count=1 skip=" + str(index) + " if=" + large_file_path + " bs=245 | openssl rsautl -encrypt -inkey " + pub_key + " -pubin", shell=True, stdout=subprocess.PIPE).stdoutj
```

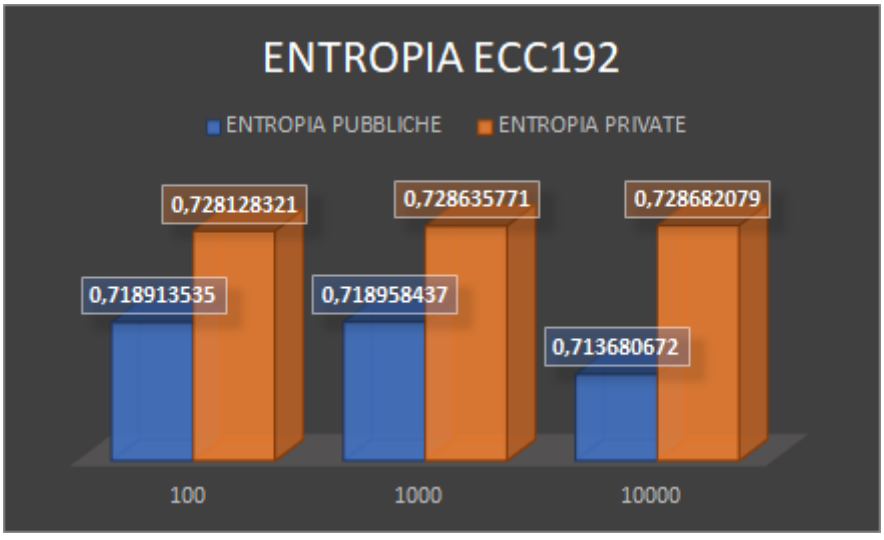
4.
```
os.popen("dd count=1 skip=" + str(index) + " if=" + crypt_file_path + " bs=256 | openssl rsautl -decrypt -inkey " + pvt_key)
```

Diffie Hellman fa parte degli algoritmi asimmetrici (o a chiave pubblica/privata) sebbene utilizzi solamente una chiave sia per cifrare che per decifrare i messaggi.
Allora perché è considerato un algoritmo asimmetrico ?
Diffie Hellman risolve tramite una coppia di chiavi (pubblica e privata) quello che era il collo di bottiglia degli algoritmi a chiave simmetrica, cioè lo scambio della chiave stessa.
I due protagonisti che vogliono scambiarsi il messaggio non devono quindi incontrarsi per concordare una chiave di cifratura, ma possono generarne una condivisa utilizzando le loro coppie di chiavi.
A genera la chiave condivisa utilizzando la propria chiave privata e la chiave pubblica di B, e B analogamente genererà la stessa chiave condivisa.
Ora A e B conoscono la stessa chiave che possono utilizzare per cifrare e decifrare messaggi e non si sono mai dovuti "incontrare" per deciderla.

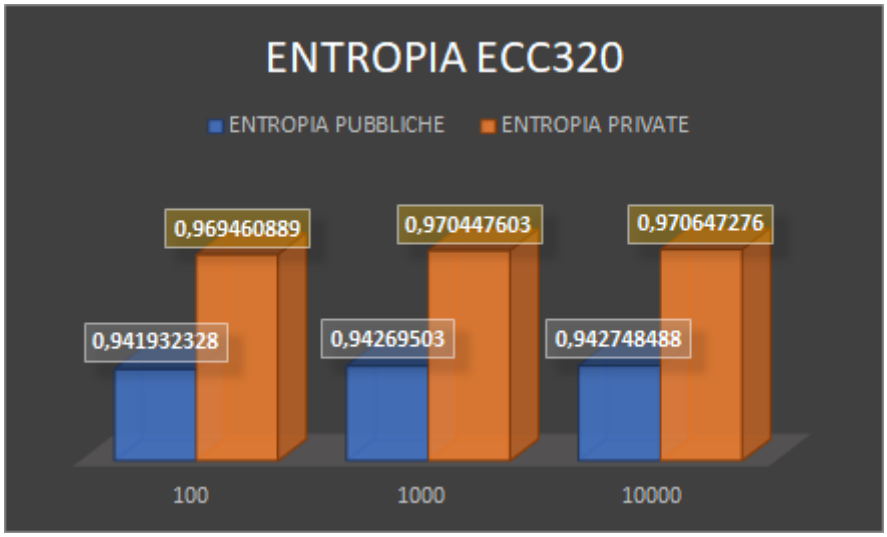
Abbiamo effettuato diverse prove sull'algoritmo di cifratura ECC, di seguito elenchiamo la tabella dei risultati ottenuti.

VALORE	RISULTATO	RISULTATO	RISULTATO	RISULTATO	RISULTATO	RISULTATO
NUMERO BIT	192	192	192	320	320	320
NUMERO CHIAVI GENERATE	100	1000	10000	100	1000	10000
TEMPO TOTALE PRIVATE (Sec.)	8,339253	57,564900	582,21195100	8,4793	60,078800	637,055048
TEMPO TOTALE PUBBLICHE (Sec.)	7,555058	55,683047	566,110723	9,2976	57,853800	613,156637
AVG TEMPO PRIVATE (Sec.)	0,08339253	0,0575654	0,058221195	0,08479	0,06	0,0637055
AVG TEMPO PUBBLICHE (Sec.)	0,07555058	0,055683047	0,056611072	0,0929	0,0578	0,06131
LUNGHEZZA CHIAVE CONCATENATA PUB (Caratteri)	10400	144000	1440000	14400	144000	1440000
LUNGHEZZA CHIAVE CONCATENATA PVT(Caratteri)	10400	132000	1320000	19600	196000	1960000
ENTROPIA PUBBLICHE	0,718913535	0,718958437	0,713680672	0,941932328	0,94269503	0,942748488
ENTROPIA PRIVATE	0,728128321	0,728635771	0,728682079	0,969460889	0,970447603	0,970647276

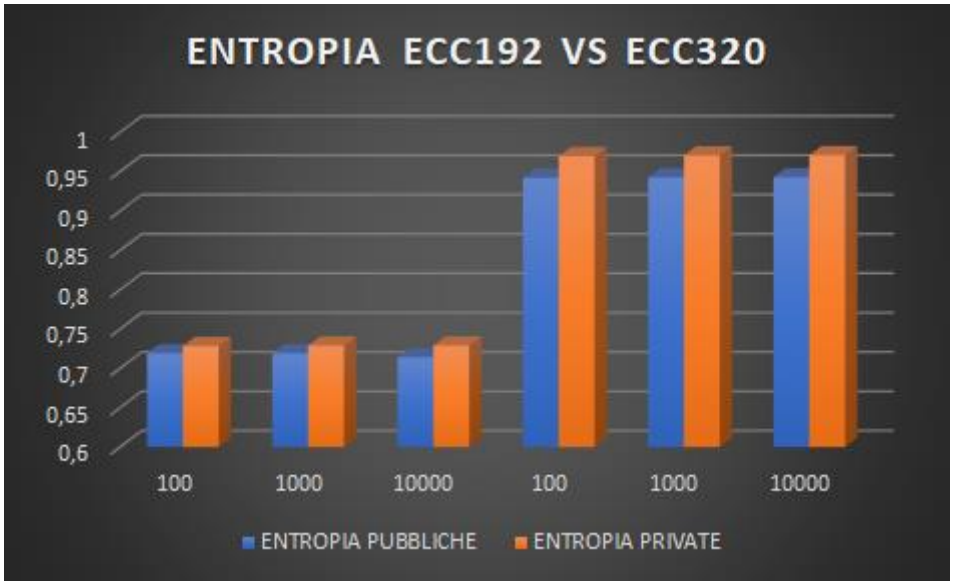
Dalla tabella ci sono diversi dati molto importanti ai fini della nostra valutazione, a partire dall'entropia nettamente inferiore delle chiavi sia pubbliche che private a 192 bits, che sottolineano la profonda differenza di applicazione dell'algoritmo ECC nel caso di chiavi a 192 e 320 bits. I tempi medi di generazione delle chiavi rimangono in media sempre gli stessi, circa 0.07, contrariamente a quanto avveniva in RSA. Analizziamo ora questi fattori mediante l'uso di grafici.



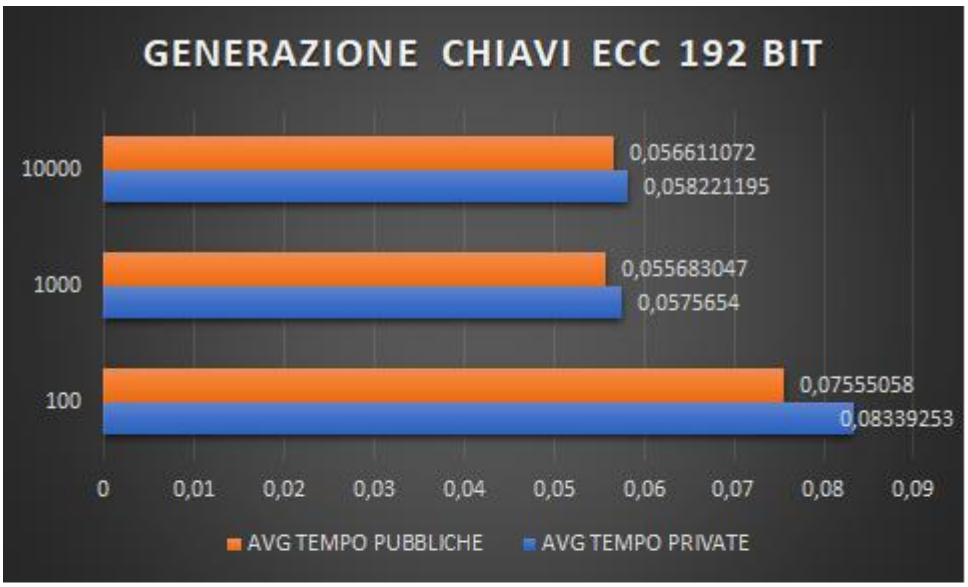
Nel grafico precedente osserviamo il comportamento dell'entropia in ECC a 192 bits, otteniamo dei valori insufficienti per un algoritmo di crittografia valido e non lo riteniamo utilizzabile in alcuni tipo di ambiente nel quale viene utilizzata crittografia. Questo fortunatamente cambia notevolmente nel caso di ECC a 320 bits come osserviamo nel grafico successivo.

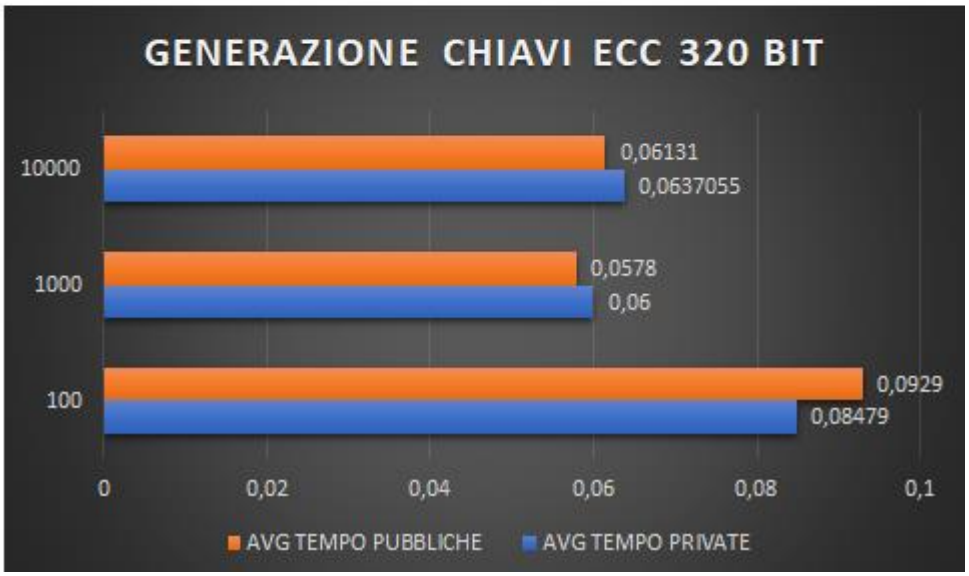


I valori di entropia relativi alle chiavi, seppur non ancora ritenuti buoni, aumentano notevolmente rispetto alle chiavi a 192 bits. Possiamo dedurre che l'algoritmo ECC diventa utilizzabile con chiavi di lunghezza >= 320 bits. Mettiamoli a confronto in un unico grafico per capire ancora meglio la differenza.



Procediamo ora con l'analisi dei tempi di generazione delle chiavi.

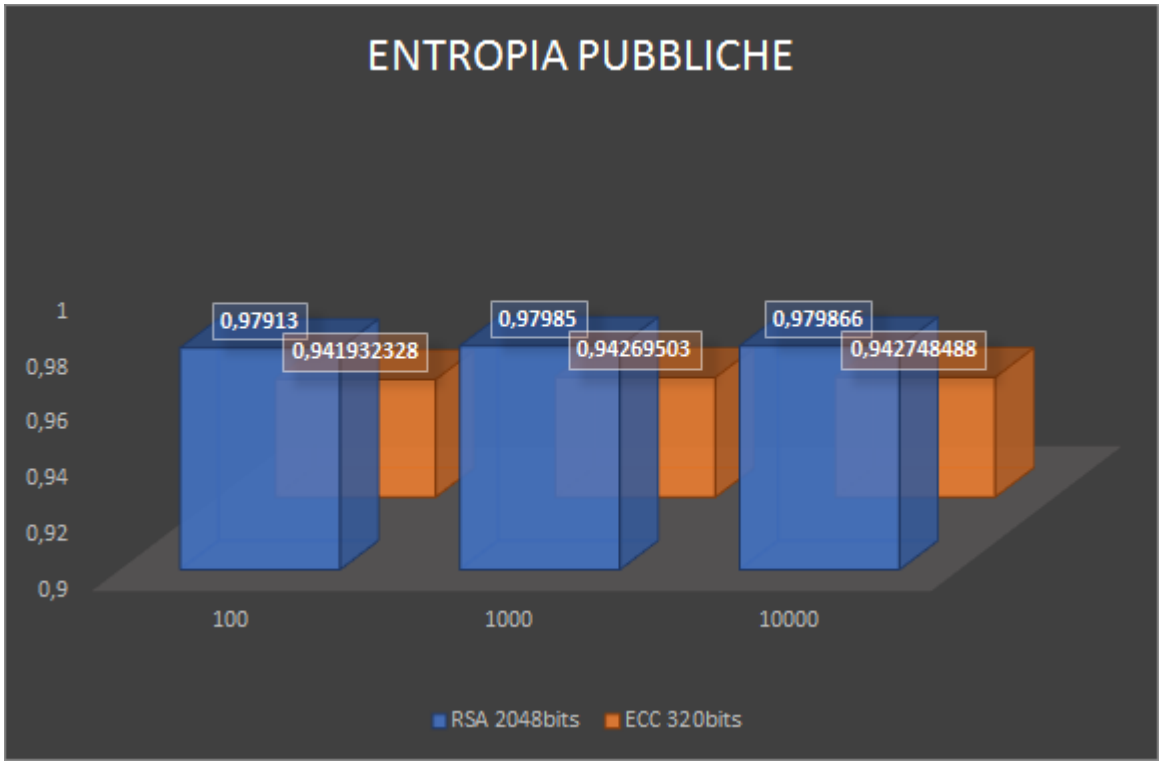


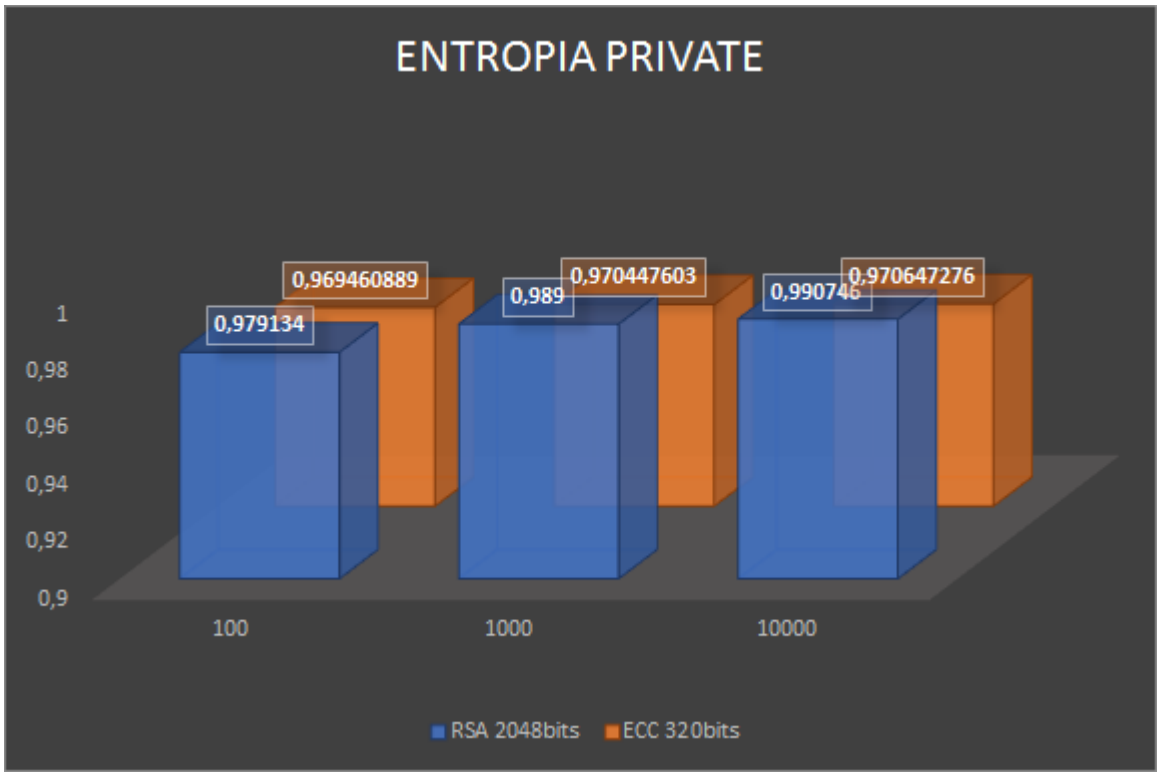


Come inizialmente descritto, i tempi medi per la generazione delle chiavi non subiscono un cambiamento sostanziale, questo è un aspetto positivo al fine di giudicare la velocità dell'algoritmo nella generazioni delle chiavi.

CONFRONTO DATI TRA RSA E ECC

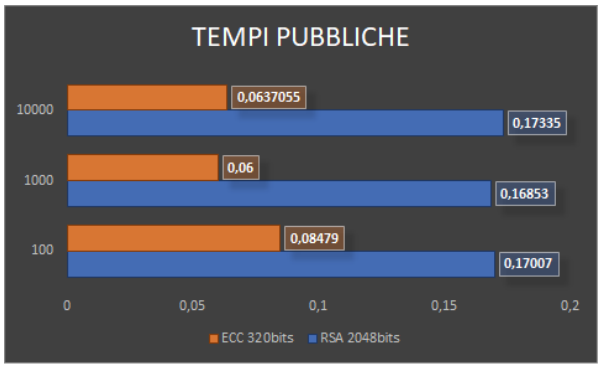
Andiamo ora a confrontare i due algoritmi di crittografia asimmetrici analizzando i tempi, la differenza di entropia delle chiavi generate ed il loro comportamento nella cifratura e decifratura di file di 10,50 e 100MB al fine di trarre delle conclusioni.

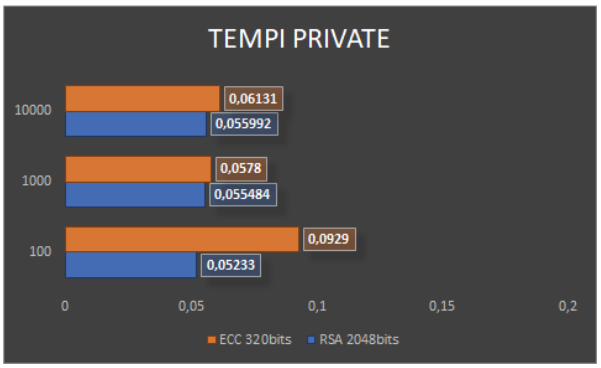




Osserviamo che l'entropia delle chiavi generate con RSA è in modo rilevante maggiore di quelle generate con l'algoritmo ECC, sia nel caso delle chiavi pubbliche che private. Questo fattore è molto importante per confrontare i due algoritmi, e verrà preso pesantemente in conto nelle conclusioni.

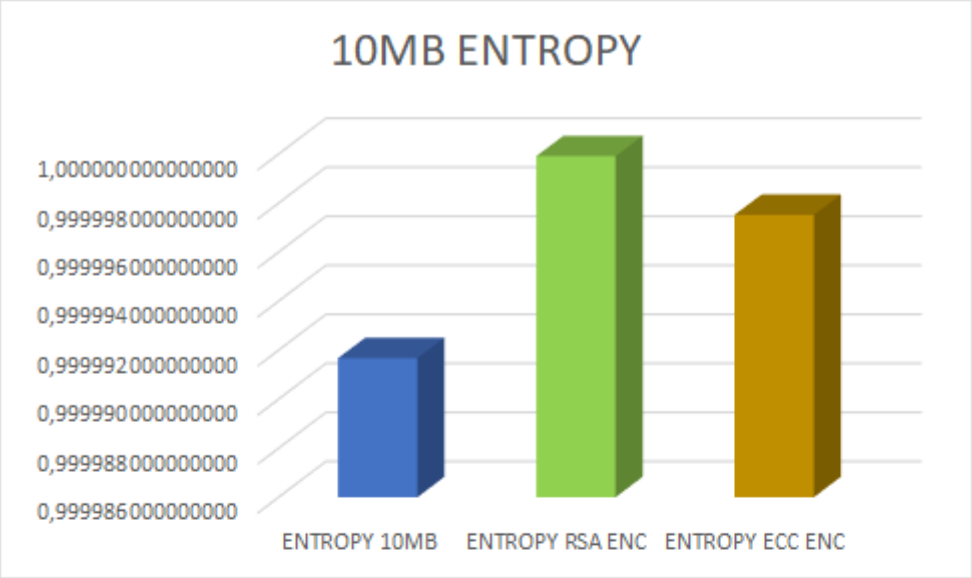
Andiamo ora ad analizzare i dati sui tempi per la generazione delle chiavi in entrambi gli algoritmi.

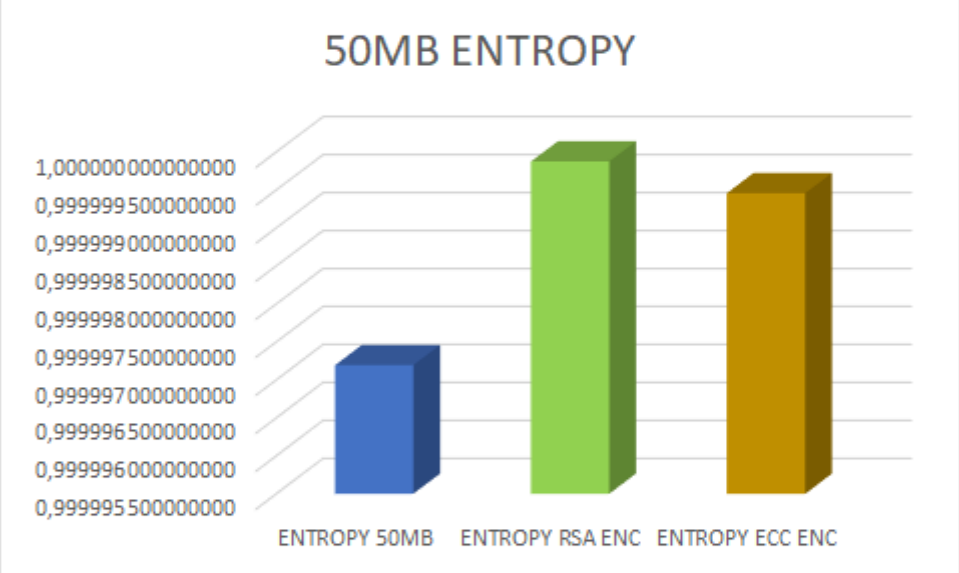


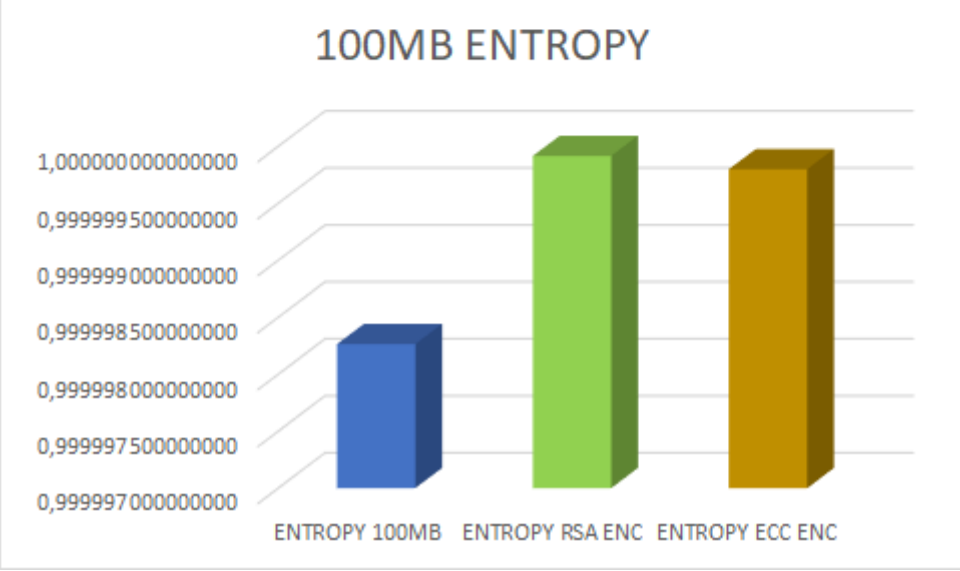


Notiamo che l'algoritmo ECC è nettamente più veloce nella generazione di chiavi pubbliche rispetto ad RSA, molto meno evidente è la differenza nel caso delle chiavi private, nelle quali i tempi si avvicinano tra loro, fino ad essere in media quasi identici.

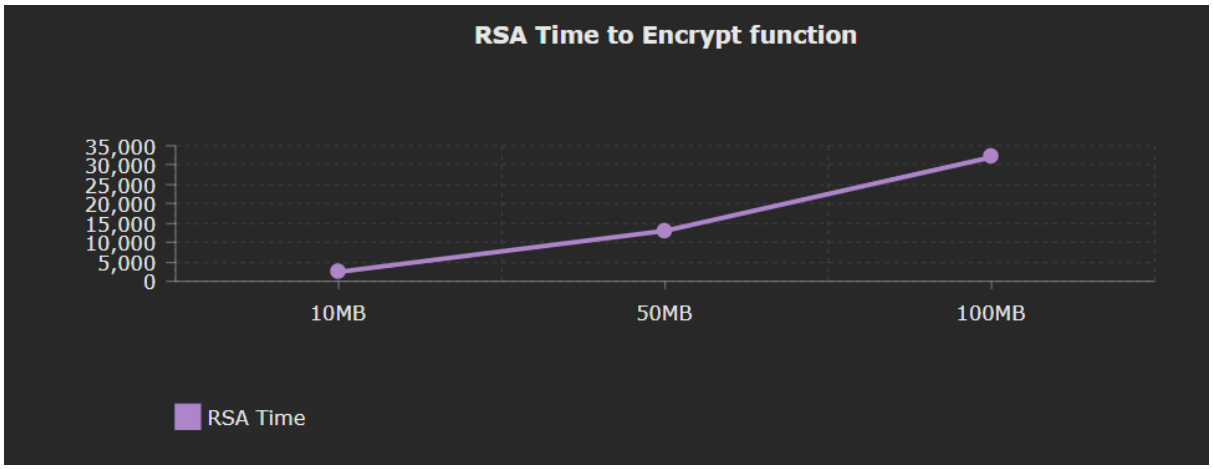
Andiamo ad analizzare adesso il comportamento dei due algoritmi nelle operazioni di cifratura e decifratura, iniziamo dal valore dell'entropia del plaintext confrontata con l'entropia dei file criptati con entrambi gli algoritmi, come nella sezione precedente su file di 10, 50 e 100MB.


10MB ENTROPY
1,000000000000000
0,999998000000000
0,999996000000000
0,999994000000000
0,999992000000000
0,999990000000000
0,999988000000000
0,999986000000000
ENTROPY 10MB
ENTROPY RSA ENC
ENTROPY ECC ENC


50MB ENTROPY
1,000000000000000
0,999999500000000
0,999999000000000
0,999998500000000
0,999998000000000
0,999997500000000
0,999997000000000
0,999996500000000
0,999996000000000
0,999995500000000
ENTROPY 50MB
ENTROPY RSA ENC
ENTROPY ECC ENC

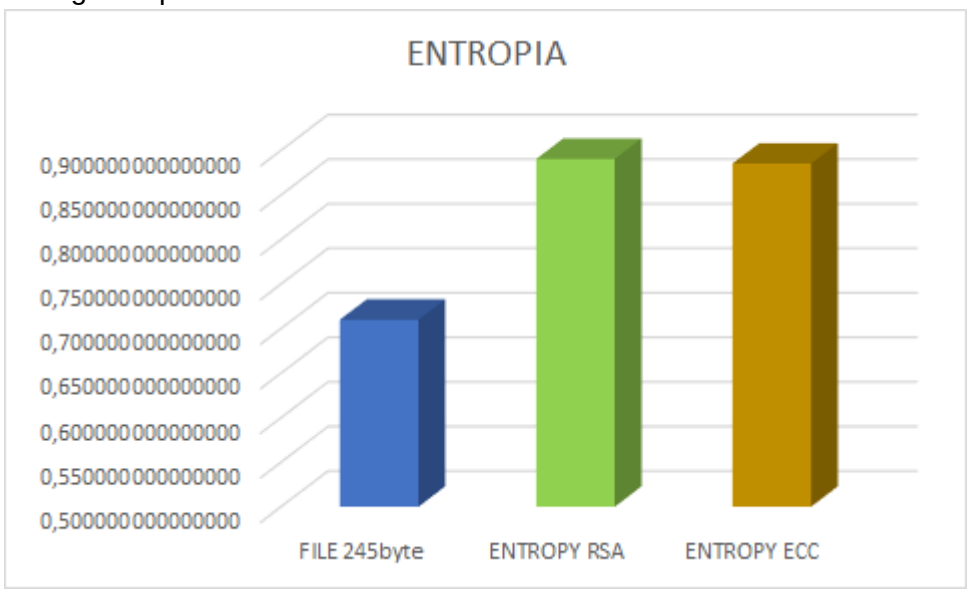

100MB ENTROPY
1,000000000000000
0,999999500000000
0,999999000000000
0,999998500000000
0,999998000000000
0,999997500000000
0,999997000000000
ENTROPY 100MB
ENTROPY RSA ENC
ENTROPY ECC ENC

Osserviamo dai grafici sovrastanti che il valore dell'entropia di RSA risulta maggiore in tutti e 3 i file criptati. Nel caso del file di 100MB la differenza di entropia tende a diminuire, questo è un fattore che riteniamo positivo per l'algoritmo ECC. Sottolineamo però la profonda superiorità dell'algoritmo RSA, che dall'analisi dei dati riteniamo molto più sicuro. Certo bisogna tener conto del limite di RSA 2048 bit, il quale con file superiori alla dimensione della chiave ha bisogno di suddividere il processo di cifratura e decifratura in $\frac{DimensioneFile}{245}$ passi, questo fa crescere la funzione tempo di cifratura come segue:



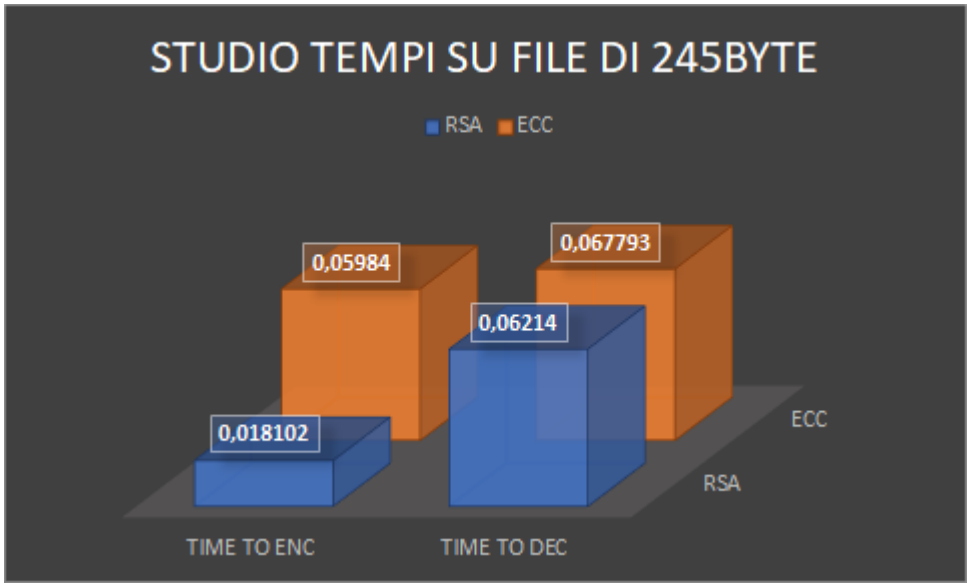
Rappresentazione del tempo impiegato dall'algoritmo RSA per cifrare file di diverse dimensioni

Nella figura precedente vediamo come cresce il tempo di cifratura all'aumentare della dimensione del file.
Per file superiori a 100 MB il tempo diviene di circa 17h, valore che lo rende inusabile in contesti un cui il tempo gioca un fattore fondamentale.
Ai fini del confronto sui tempi tra gli algoritmi RSA e l'algoritmo ECC, abbiamo deciso di analizzare il comportamento tra i due nel caso di un file minore della dimensione della chiave.
Di seguito riportiamo i risultati ottenuti.



L'entropia del file, avendo dimensioni molto piccole risulta avere un'entropia non molto elevata, di circa 0.71, cifrandola con entrambi gli algoritmi il file cifrato con RSA risulta

leggermente più entropico rispetto a quello generato con ECC, questo conferma i dati risultanti dai file di dimensioni maggiori. Ora andiamo a valutare il fattore tempo in questo contesto, che risulta molto più coerente e significativo.



Osserviamo che RSA risulta essere molto più veloce nella cifratura del file rispetto a ECC, mentre il distacco si riduce nell'operazione di decifratura.

CONCLUSIONI

Dall'analisi dei dati emerge una superiorità importante di RSA rispetto ad ECC, sia per quanto riguarda l'entropia relativa alle chiavi che per quanto riguarda l'entropia relativa ai file cifrati. Inoltre anche il fattore tempo risulta essere a favore di RSA nel caso di file di dimensione < CHIAVE.
Altrettanto significativo è il fatto che l'algoritmo RSA non è utilizzabile per cifrare file di dimensioni elevate, poichè i tempi per svolgere queste operazioni sono inaccettabili in qualunque tipo di scenario reale. Analizzeremo tutti gli algoritmi e trarremo le nostre conclusioni nella sezione successiva.

OSSERVAZIONI SULL'AGGIUNTA DEL PADDING NELLA CRITTOGRAFIA SIMMETRICA E ASIMMETRICA

In crittografia il **padding** è una sequenza di bit, che per semplicità possiamo pensare come un numero che è incluso nei dati che vogliamo criptare, quindi all'inizio, nel mezzo o alla fine del messaggio.
Quindi il padding contiene parti di testo random per evitare il fatto che il messaggio possa avere dei pattern prevedibili.

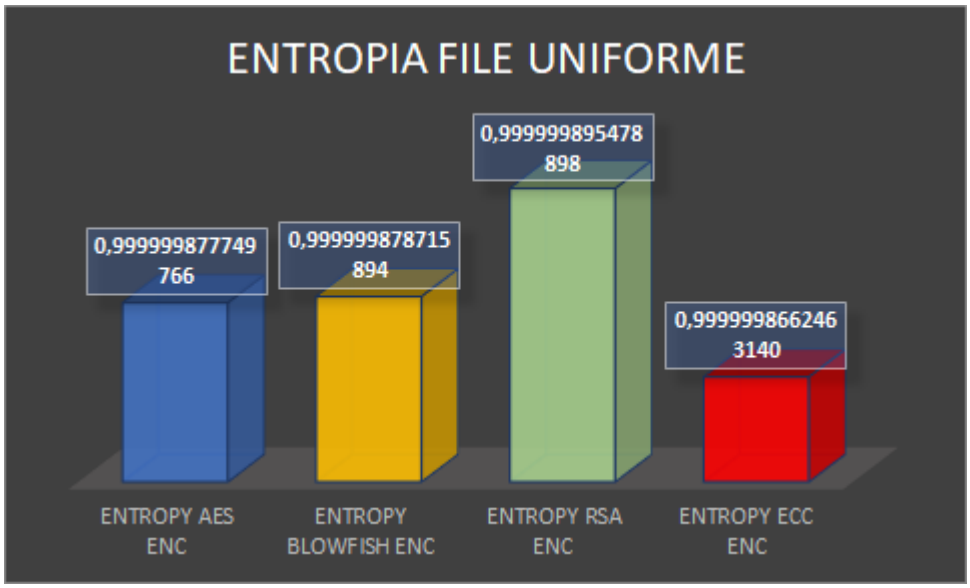
Per avere quindi una visione più completa di questi algoritmi di cifratura al fine di trarre conclusioni più veritiere, abbiamo eseguito i test precedentemente descritti anche con l'aggiunta di padding.

Abbiamo utilizzato lo standard PKCS incluso nel software OpenSSL per simulare un messaggio con padding strutturato.

Si è notato che il tempo di generazione delle chiavi e i tempi di esecuzione degli algoritmi aumentano di poco, ma questo solamente perchè l'aggiunta di padding aumenta la lunghezza del messaggio da cifrare.
Per quanto riguarda l'entropia del messaggio criptato risultante, anche questa ha una leggerissima variazione, poichè queste componente random del padding aggiunte aumentano anche si di poco l'entropia del messaggio.
Queste conclusione valgono per messaggi che hanno bassa entropia, a differenza dei file prodotti con butterly visti precedentemente, dove invece il valore dell'entropia rimane pressochè invariato poiché prossima ad 1, quindi l'aggiunta del padding risulta irrilevante.

VALUTAZIONE ENTROPIA CON STRINGA UNIFORME

In seguito alla generazione di un file uniforme di 100Mb (sequenza di “1”) abbiamo analizzato l’entropia del file in output usando i quattro algoritmi analizzati.
Premesso che l’entropia del file in chiaro risulta essere nulla, questi sono i risultati che abbiamo ottenuto sono:

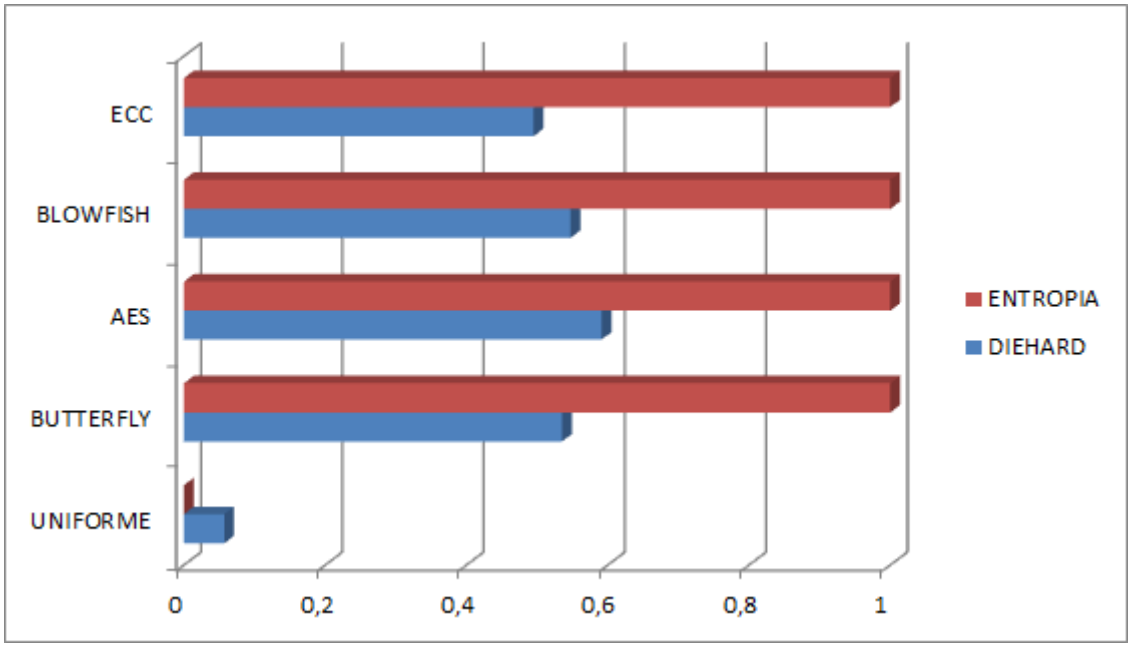


Notiamo che l’entropia maggiore è ottenuta in seguito all’applicazione dell’algoritmo RSA, risultato che ci aspettavamo anche dalle precedenti conclusioni.
Per quanto riguarda gli algoritmi simmetrici AES e BLOWFISH, l’entropia risultante sembra essere pressochè la stessa, abbiamo variazioni dalla settima cifra decimale dove Blowfish ha un maggiore valore.
Infine l’entropia più bassa risulta essere quella ottenuta dall'algoritmo ECC.
Comunque tutti e quattro gli algoritmi riescono ad aumentare notevolmente l’entropia rispetto al file in input.

CONFRONTO DIE-HARD VS ENTROPIA

La maggior parte dei test in Diehard restituisce un valore p, che dovrebbe essere uniforme su [0,1) se il file di input contiene bit casuali realmente indipendenti. Questi valori p sono ottenuti da p = F (X), dove F è la distribuzione presunta, spesso normale, della variabile casuale X del campione.

Nei grafici che seguono viene confrontata la media dei p-value con il valore dell'entropia.



LE NOSTRE CONCLUSIONI

Di seguito inseriamo una tabella riassuntiva ai fini di comparare i vari algoritmi analizzati in base ai diversi fattori

.

NOME	RSA	ECC	AES	BLOWFISH
TIPO	ASIMMETRICO	ASIMMETRICO	SIMMETRICO	SIMMETRICO
VELOCITÀ GENERAZIONI CHIAVI	0,06	0,06	0,07	0,06
VELOCITÀ ENCRYPT	32025	1,7410890	1,592711	1,681610
VELOCITÀ DECRYPT	31581	1,32526100	1,299341	3,489115
ENTROPIA FILE	0,999999915892140	0,99999979864927	0,999999761628837	0,999999757792085
ENTROPIA CHIAVI	OTTIMA	DISCRETA	OTTIMA	OTTIMA
DIMENSIONE MASSIMA FILE	245byte	INF	INF	INF

Analizziamo ora diversi scenari di applicazione dei vari algoritmi di crittografia, indicando quale algoritmo riteniamo migliore e giustificando la nostra scelta.

- **FILE DI GRANDI DIMENSIONI, CON INFORMAZIONI SENSIBILI MEMORIZZATE SU DI UN SERVER, POTREBBE ESSERE IL CASO DI UN ENTE PUBBLICO QUALE AD ESEMPIO INPS O IL PARLAMENTO**

  Viste le dimensioni significative dei file, escludiamo l'uso di RSA per le motivazioni precedentemente descritte, ovvero che la tempistica per il processo di cifratura/decifratura diviene troppo elevata. Poiché abbiamo a che fare con informazioni sensibili preferiamo l'uso della crittografia asimmetrica, la nostra scelta ricade dunque sull'algoritmo di cifratura ECC.

- **FILE DI GRANDI DIMENSIONI, SENZA INFORMAZIONI SENSIBILI MEMORIZZATI SU DI UN SERVER (VIDEO, MUSICA, IMMAGINI). POTREBBE ESSERE IL CASO DI UN PROVIDER CLOUD OPPURE DI UN QUALUNQUE SITO PER LA CONDIVISIONE DI MATERIALE MULTIMEDIALE**

  In questo scenario dove le informazioni risultano non sensibili ma necessitano comunque di essere scambiate in dimensioni piuttosto notevoli, la scelta di cifratura è di utilizzare la crittografia simmetrica in particolare scegliamo AES rispetto a Blowfish poichè risulta essere più veloce soprattutto in fase di decrypt.

- **FILE DI PICCOLE DIMENSIONI, CON INFORMAZIONI SENSIBILI MEMORIZZATE SU DI UN SERVER, POTREBBE ESSERE IL CASO DI APPLICAZIONE DI MESSAGGISTICA CHE UTILIZZA LA CRITTOGRAFIA PER I MESSAGGI**

  In questo caso l'algoritmo scelto è l'RSA, parliamo di file di piccole dimensioni, quindi compatibili con l'algoritmo che però contengono informazioni sensibili. Vista l'affidabilità di RSA in questo scenario è la scelta a parere nostro migliore.

- **FILE DI PICCOLE DIMENSIONI,SENZA INFORMAZIONI SENSIBILI MEMORIZZATE SU DI UN SERVER, POTREBBE ESSERE IL CASO UN FORUM PUBBLICO**

In questo caso, con file di piccole dimensioni che non hanno informazioni sensibili possiamo scegliere di affidarci alla crittografia simmetrica, andando a migliorare notevolmente il fattore tempo. Scegliamo di adottare l'algoritmo AES, data la sua efficienza e l'alta affidabilità.

- **COMUNICAZIONE DI RETE CLIENT SERVER**
  In questo tipo di comunicazioni, una policy importante che il server deve rispettare è la segretezza dei dati ricevuti dal client.
  L'algoritmo piu comunemente usato per applicazioni client-server è l'RSA, grazie al quale ogni client fornisce solamente la propria chiave pubblica al server, e questo gli permette una comunicazione sicura con lo stesso.
  Il server per maggiore sicurezza potrebbe utilizzare chiavi differenti per i vari servizi che offre, in modo tale che non basti una sola violazione delle sue chiavi per appropriarsi di tutti i suoi privilegi.
  Inoltre potrebbe essere opportuno criptare a sua volta ogni chiave privata per una maggiore sicurezza.

APPROFONDIMENTI

Crittografia e Crittografia Moderna

Fino al ventesimo secolo la crittografia viene definita come "arte". Costruire codici buoni e spezzarli si basava solamente su skills e creatività personali.
Dalla fine del ventesimo secolo la crittografia è radicalmente cambiata, è stata iniziata a studiare come vera e propria *scienza.*
Possiamo definire la crittografia moderna come *lo studio scientifico di tecniche per la sicurezza di informazione digitale, transazioni, e computazioni distribuite.*
Un altra importante differenza tra la crittografia classica e quella moderna è l'uso che se fa.

Storicamente la crittografia era usata in ambito militare e di organizzazioni di intelligence, oggi la crittografia è ovunque, (e.g dati di utenti su un server…) per questo è diventata quindi un topic sempre più studiato in *computer science*.

Crittografia Perfetta

Intuitivamente immaginiamo un un nemico che conosca la distribuzione di probabilità sullo spazio dei messaggi possibili *M*.
Supponiamo che il nemico riesca anche ad osservare il *ciphertext* inviato tra le due parti di un canale di comunicazione.
Idealmente quest'ultima osservazione non deve avere effetti sulla sua conoscenza riguardo al messaggio *m* in chiaro.

Definizione Perfectly-Secret

Lo schema crittografico *(Gen,Enc,Dec)* su un insieme di messaggi *M*, si dice *perfectly-secret* se per ogni distribuzione di probabilità su *M,* ogni messaggio $m \in M$, e ogni ciphertext $c \in C$, per le quali *Pr[C=c] > 0:*

$$P[M = m \mid C = c] = P[M = m]$$

Teorema Di Shannon

Shannon definì un' altra caratterizzazione di un sistema di crittografia perfetto,dove assume che :

$$| K | = | M | = | C |$$

(cardinalità degli spazi delle chiavi private, dei messaggi e dei chipertext).

Teorema:
Sia (Gen,Enc,Dec) un o schema crittografico su uno spazio dei messaggi M per il quale vale che |K|=|M|=|C|. Lo schema è perfectly-secret se:

1. Ogni chiave $k \in K$ è scelta con probabilità $1/|k|$ dall'algoritmo Gen
2. Per ogni messaggio $m \in M$ e ogni $c \in C$, esiste un chiave unica $k \in K$ tale che:

$$Enc_k(m) = c$$

Certificati Digitali

Un **certificato digitale** è un documento elettronico che attesta l'associazione univoca tra una chiave pubblica e l'identità di un soggetto (una persona, una società, un pc, etc) che dichiara di utilizzarla nell'ambito delle procedure di cifratura asimmetrica e/o autenticazione tramite firma digitale.

Il certificato digitale contiene informazioni sulla chiave, informazioni sull'identità del proprietario (denominato oggetto) e la firma digitale di un'entità che ha verificato i contenuti del certificato (denominato emittente). Se la firma è valida e il software che esamina il certificato si affida all'emittente, allora può utilizzare tale chiave per comunicare in modo sicuro con il soggetto del certificato. Nella crittografia email, nella firma di codice e nei sistemi di firma elettronica, un soggetto del certificato è tipicamente una persona o un'organizzazione.

Tale certificato, fornito da un ente terzo fidato e riconosciuto come autorità di certificazione (CA), è a sua volta autenticato per evitarne la falsificazione sempre attraverso firma digitale ovvero cifrato con la chiave privata dell'associazione la quale fornisce poi la rispettiva chiave pubblica associata per verificarlo.

Il certificato digitale è impossibile da duplicare o falsificare e può essere facilmente verificato online (o via cellulare) dal possibile acquirente. Il "costo marginale di produzione" del certificato digitale è vicino allo zero, il che vuol dire che sarebbe possibile proteggere anche oggetti di prezzo unitario relativamente basso

Funzionamento

Alice inserisce la sua chiave pubblica in un certificato firmato da una terza parte fidata ("*trusted third party*"): tutti quelli che riconoscono questa terza parte devono semplicemente controllarne la firma per decidere se la chiave pubblica appartiene veramente a *Alice*.

In una PKI, la terza parte fidata sarà un'autorità di certificazione che apporrà anch'essa una firma sul certificato per validarlo. Nel web of trust, invece, la terza parte può essere un utente qualsiasi (ovvero la firma è quella o dello stesso utente (un'auto-certificazione) oppure di altri utenti ("*endorsements*")) e sarà compito di chi vuole comunicare con *Alice* decidere se questa terza parte è *abbastanza fidata*.

In entrambi i casi, la firma *certifica* che la chiave pubblica dichiarata nel certificato appartiene al soggetto descritto dalle informazioni presenti sul certificato stesso (nome, cognome, indirizzo abitazione, indirizzo IP, etc.).

Guile

LISP che significa "**Lis**t **P**rocessor language" (linguaggio per processare liste), è un linguaggio di programmazione che nasce ad opera di John McCarthy nel 1959 molto utilizzato nel campo dell'intelligenza artificiale.

Orientato alla logica ed ad una programmazione funzionale ricorsiva è stato impiegato per programmazione di basso livello e di alto livello ad oggetti. È stato il primo linguaggio ad adottare i concetti di *Virtual Machine* e *Virtual Memory management*.

Usato oggi per creare siti web dinamici, sistemi esperti, progetti di intelligenza artificiale., motori inferenziali con base dati della conoscenza al servizio di *problem solving* di strategia, NASA, NCSA, e Difesa Americana lo vedono ancora come strumento strategico per le potenti opportunità che ancora oggi fanno di questo linguaggio il maggior *competitor* di nicchia, in contrapposizione a Java, C++, C#.

Il **Lisp** è un linguaggio di programmazione elegante, sintatticamente semplice e di facile apprendimento, usato per la sua flessibilità e la propensione a manipolare variabili con astrazione di oggetti complessi quali liste di dati. La sua particolare caratteristica funzionale e ricorsiva lo rende tanto flessibile da essere l'unico linguaggio in grado di interpretare/compilare se stesso con un polimorfismo nel quale i dati diventano codice e il codice viene manipolato come un dato.

Guile è un interprete Scheme creato come parte del progetto GNU, con l'intento di permettere l'estensione/*scripting* di altre applicazioni. Visto che un Scheme adatto interprete non esisteva ancora, Guile è stato creato assemblando diverso materiale esistente.

Scheme è un linguaggio di programmazione funzionale, un dialetto del Lisp di cui mantiene tutte le caratteristiche, che è stato sviluppato negli anni settanta da Guy L. Steele e Gerald Jay Sussman.

A differenza della maggior parte degli altri linguaggi di programmazione, Scheme utilizza una notazione prefissa, ovvero una notazione in cui al posto di scrivere (2 + 3) si scrive (+ 2 3). Questa notazione si propaga a tutte le funzioni, sicché se abbiamo una funzione N-aria f, la sua rappresentazione sarà (f argomento1 argomento2... argomentoN).

Scheme implementa tutti i tipi di dato fondamentale: booleani, numeri, caratteri, stringhe, vettori. Tuttavia include anche tipi speciali, tra cui le liste (coppie), le porte (flussi di dati), i simboli e le procedure.

Le liste sono un particolare tipo di dato. Come gli array, rappresentano collezioni ordinate di elementi; a differenza degli array, gli elementi possono essere eterogenei (di tipi differenti fra loro), e inoltre non possono essere indicizzati. Le liste sono realizzate come coppie: (2 3) rappresenta un esempio di lista che è evidentemente una coppia; ma anche (2 3 4) è in realtà una coppia, formata dal primo elemento (2) e da tutti gli altri (3 4). A sua volta, l'elemento "tutti gli altri" è una coppia formata dal suo primo elemento (3) e da tutti gli altri (in questo caso solo il 4). La lista è quindi descritta in modo molto naturale in termini ricorsivi.

Una lista può contenere qualunque tipo di dato, come caratteri, stringhe, booleani, e anche altre liste (esempio:"(2 3 (4 5))"); come già detto, possono anche contenere tipi di dato misti (esempio:"(#\T 4 (4 #F ("stringa")))").

Le due funzioni fondamentali per agire sulle liste si rifanno alla definizione ricorsiva accennata sopra: abbiamo così la funzione **car**, che restituisce il primo elemento, e **cdr**, che restituisce il secondo elemento (l'elemento "tutti gli altri").

Condizione semplice (if)

```
(if condizione espressione_nel_caso_la_condizione_sia_vera
        eventuale_espressione_nel_caso_la_condizione_sia_falsa)
```

Condizione composta (cond)

```
(cond
  (prima_condizione espressione)
  (seconda_condizione espressione)
  ...
  (else espressione))
```

Scheme è particolarmente adatto a esprimere gli algoritmi in forma ricorsiva. La ricorsione semplice si ottiene richiamando un'unica volta la procedura stessa.

Numeri Primi

Un *test di primalità* è un algoritmo che permette di stabilire se un dato numero è primo oppure no. Nella teoria della complessità computazionale, questo problema è stato recentemente dimostrato appartenere alla classe di complessità P

Il più semplice test di primalità è quello che sfrutta il brute force, che consiste nell'applicare direttamente la definizione di numero primo: si prova a dividere il numero *N* per tutti i numeri minori di *N*: se nessuno di questi lo divide, allora il numero è primo. Un semplice miglioramento di questo metodo si ottiene limitando i tentativi di divisione ai numeri primi minori di $\sqrt{N}$.

Tale metodo è poco usato nella pratica, perché richiede tempi di calcolo esponenziali rispetto alla grandezza dell'input N. Esso tuttavia fornisce anche i suoi fattori primi (ed è quindi un algoritmo di fattorizzazione): questo non succede nel caso di algoritmi più sofisticati, che riescono a stabilire se un numero non è primo anche senza determinare alcun divisore non banale.

Un altro algoritmo di primalità molto usato e semplice da implementare è il **test di fermat** il quale si basa sul piccolo teorema di Fermat.

Altri, come il test di Miller-Rabin, sono probabilistici, ovvero danno una risposta certa solo se affermano che il numero *non* è primo, mentre se si ottiene come risultato che il numero è primo, allora c'è solo un'alta probabilità che il numero effettivamente lo sia.

PICCOLO TEOREMA DI FERMAT

Il piccolo teorema di Fermat afferma che:

$$ap = a\ (mod\ p)$$

per ogni numero primo p e intero a.

DIMOSTRAZIONE

E da notare che basta provare: $ap - 1\ \equiv\ 1\ (mod\ p)$ per ogni intero a coprimo con p. Moltiplicando ambo i membri dell'ultima espressione per a si ottiene la versione esposta a inizio pagina del teorema. Se a non fosse primo con p allora $ap\ \equiv\ 0\ \equiv\ a\ (mod\ p)$ed il teorema risulterebbe vero in ogni caso.
Sfrutteremo quindi la semplificazione sopra riportata. Si considerino i multipli di a che vanno da a stesso fino a $(p-1)\ a$. Nessuno di questi multipli può dare resto 0 diviso per *p* perchè nè *p-1* nè a sono multipli interi di *p*. Inoltre non può esistere una coppia di questi multipli che sia congrua modulo *p*, perchè, se fosse per esempio: $ra\ \equiv\ sa\ (mod\ p)$ si avrebbe $(r-a)\ a\ \equiv\ 0\ (mod\ p)$ .Ma questo è impossibile, perchè allora *p* dovrebbe dividere uno dei due fattori.
Ma *a* è primo con *p*, e *r−s*, essendo *r* ed *s* numeri naturali compresi tra *1* e *p*, `e $(r-s)\ <\ p$. Per cui i multipli considerati hanno un resto nella divisione per *p* differente per ciascuno di essi, e differente da *0*. Siccome consideriamo *p−1* multipli, tali multipli devono essere necessariamente congrui *(modulo p)* ai numeri *1,2,3,...,p−1* in un certo ordine.
Ne segue, per il prodotto di tutti questi multipli:

$$a\ (2a)\ (3a)\ldots(p-1)\ a\ =\ 1*2*3*\ldots*(p-1)\ ap-1\ \equiv\ 1*2*3*\ldots*(p-1)\ (mod\ p)$$

da cui, ponendo $K = 1*2*3*\ldots*(p-1)$ si ha $K\ (ap\ -1\ -1\ )\ \equiv\ 0\ (mod\ p)$.

Dato che *p* è primo, l'unico modo affinchè ciò avvenga è che o *K* o il secondo fattore sia divisibile per *p*. (con *p* primo le classi di modulo *n* costituiscono un dominio di integrità). Ma *K* non è divisibile per *p*, perchè non lo è nessuno dei suoi fattori; quindi deve essere: $ap - 1 - 1 \equiv 0(mod\ p)$.

Randomness e Pseudorandomness

Tutti noi abbiamo almeno un'idea di cosa sia la randomicità, quindi un numero random?
Ma cosa intendiamo con numero *pseudorandom* ?
Un numero pseudorandom è un'approssimazione di un numero random generato da un software. I software però che girano su hardware comuni sono deterministici, risulta quindi molto difficile creare qualcosa di puramente casuale.
Quello che il software vorrebbe simulare è la generazione di un numero in un certo range, dove ogni numero che ne appartiene ha la stessa probabilità di essere generato.
I numeri pseudorandom generati da un software dipendono generalmente da quello che viene chiamato *seed*, cioè una sorta di numero iniziale da cui dipenderà il nostro risultato pseudocasuale.
Il seed può essere generato da fattori come, un input da tastiere, o il movimento del mouse su schermo, oppure il tocco su uno schermo touchscreen, eventi quindi "imprevedibili".
Poiché i calcolatori che usiamo tutti i giorni sono deterministici, l'unica cosa che possiamo fare in campi come la crittografia è accontentarci di numeri pseudocasuali, per questo i generatori pseudocasuali sono molto importanti nella computer science e non solo.

Crittografia Quantistica

La **crittografia quantistica** consiste in un approccio alla crittografia che utilizza peculiari proprietà della meccanica quantistica nella fase dello scambio della chiave per evitare che questa possa essere intercettata da un attaccante senza che le due parti in gioco se ne accorgano. Si utilizza questo principio per realizzare un cifrario perfetto del tipo One Time Pad, senza il problema di dover scambiare la chiave (anche se lunga quanto il messaggio) necessariamente su un canale sicuro. La prima rete a crittografia quantistica funzionante è stata il DARPA Quantum Network.

L'enorme rivoluzione introdotta da questa tecnica sembra mettere la parola fine alla perpetua lotta tra crittografia e crittoanalisi, che da sempre si rincorrono, l'una per creare cifrari sempre più complessi, l'altra per sviluppare nuove tecniche atte a decriptarli. Finora l'unico cifrario perfetto (di cui quindi è stata dimostrata matematicamente l'indecifrabilità) realizzato è quello di Vernam. Il grosso problema di questa tecnica è legato però allo scambio della chiave tra l'emettitore e il ricevente, in quanto questa deve essere lunga quanto il messaggio, casuale (random), deve rimanere segreta e può essere utilizzata solo una volta; la chiave dovrà quindi essere scambiata su un canale sicuro. È ovvio quindi che se è possibile disporre di un canale sicuro per scambiare una chiave lunga quanto il messaggio lo stesso canale sarebbe da utilizzare per scambiare il messaggio stesso. Quest'aspetto ha reso di fatto la tecnica difficilmente realizzabile su vasta

scala. L'avvento della crittografia quantistica risolve definitivamente quest'aspetto rendendo possibile cifrare messaggi in maniera tale che nessuna spia possa decifrarli.

Un principio chiave alla base della crittografia quantistica è il principio di indeterminazione di Heisenberg. Qui non è importante conoscerne i dettagli, ma è importante mettere in rilievo alcune caratteristiche delle particelle elementari che ne derivano. Un'interpretazione del principio di Heisenberg sostiene che non è possibile conoscere, simultaneamente e con precisione assoluta, alcune particolari coppie di caratteristiche di un oggetto quantistico, come ad esempio la posizione e la quantità di moto: se si cerca di misurare esattamente la posizione di un elettrone, si perde la possibilità di verificarne la quantità di moto. Se invece si misura con precisione assoluta la quantità di moto, si perdono inevitabilmente informazioni sul luogo in cui l'elettrone si trova. I fotoni, oggetti quantistici, sono quindi soggetti al principio di indeterminazione di Heisenberg.

Un appunto a tal riguardo va fatto per chiarire che queste sono limitazioni anche in linea di principio e quindi sussistono, per quanto possa essere buono l'apparato di misura. Secondo la meccanica quantistica, per quante tecniche nuove e più precise si possano sviluppare per eseguire queste misure, il limite non è concettualmente eliminabile.

TEST DIE-HARDER

1 PARAMETRI DI VALUTAZIONE

Risultati del Test di Die-Hard sul un file di testo di 100MB uniforme

```
#=============================================================================
========#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================
========#
   rng_name    |           filename             |rands/second|
 file_input_raw|                  generated.txt|  2.40e+07  |
#=============================================================================
========#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================
========#
   diehard_birthdays|   0|       100|     100|0.00000000|  FAILED
      diehard_operm5|   0|   1000000|     100|0.62564441|  PASSED
```

```
  diehard_rank_32x32|   0|     40000|     100|0.00000000|  FAILED
    diehard_rank_6x8|   0|    100000|     100|0.00000000|  FAILED
   diehard_bitstream|   0|   2097152|     100|0.00000000|  FAILED
        diehard_opso|   0|   2097152|     100|0.00000000|  FAILED
        diehard_oqso|   0|   2097152|     100|0.00000000|  FAILED
         diehard_dna|   0|   2097152|     100|0.00000000|  FAILED
diehard_count_1s_str|   0|    256000|     100|0.00000000|  FAILED
diehard_count_1s_byt|   0|    256000|     100|0.00000000|  FAILED
 diehard_parking_lot|   0|     12000|     100|0.00000000|  FAILED
    diehard_2dsphere|   2|      8000|     100|0.00000000|  FAILED
    diehard_3dsphere|   3|      4000|     100|0.00000000|  FAILED
     diehard_squeeze|   0|    100000|     100|0.00000000|  FAILED
        diehard_sums|   0|       100|     100|0.00000000|  FAILED
        diehard_runs|   0|    100000|     100|0.99372028|  PASSED
        diehard_runs|   0|    100000|     100|0.96567691|  PASSED
       diehard_craps|   0|    200000|     100|0.00000000|  FAILED
       diehard_craps|   0|    200000|     100|0.00000000|  FAILED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.00000000|  FAILED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.00000000|  FAILED
         sts_monobit|   1|    100000|     100|0.00000000|  FAILED
            sts_runs|   2|    100000|     100|0.00000000|  FAILED
          sts_serial|   1|    100000|     100|0.00000000|  FAILED
          sts_serial|   2|    100000|     100|0.00000000|  FAILED
          sts_serial|   3|    100000|     100|0.00000000|  FAILED
          sts_serial|   3|    100000|     100|0.00000000|  FAILED
          sts_serial|   4|    100000|     100|0.00000000|  FAILED
          sts_serial|   4|    100000|     100|0.00000000|  FAILED
          sts_serial|   5|    100000|     100|0.00000000|  FAILED
          sts_serial|   5|    100000|     100|0.00000000|  FAILED
          sts_serial|   6|    100000|     100|0.00000000|  FAILED
          sts_serial|   6|    100000|     100|0.00000000|  FAILED
          sts_serial|   7|    100000|     100|0.00000000|  FAILED
          sts_serial|   7|    100000|     100|0.00000000|  FAILED
          sts_serial|   8|    100000|     100|0.00000000|  FAILED
          sts_serial|   8|    100000|     100|0.00000000|  FAILED
          sts_serial|   9|    100000|     100|0.00000000|  FAILED
          sts_serial|   9|    100000|     100|0.00000000|  FAILED
          sts_serial|  10|    100000|     100|0.00000000|  FAILED
          sts_serial|  10|    100000|     100|0.00000000|  FAILED
          sts_serial|  11|    100000|     100|0.00000000|  FAILED
          sts_serial|  11|    100000|     100|0.00000000|  FAILED
          sts_serial|  12|    100000|     100|0.00000000|  FAILED
          sts_serial|  12|    100000|     100|0.00000000|  FAILED
          sts_serial|  13|    100000|     100|0.00000000|  FAILED
          sts_serial|  13|    100000|     100|0.00000000|  FAILED
          sts_serial|  14|    100000|     100|0.00000000|  FAILED
          sts_serial|  14|    100000|     100|0.00000000|  FAILED
          sts_serial|  15|    100000|     100|0.00000000|  FAILED
```

```
          sts_serial|  15|   100000|    100|0.00000000|  FAILED
          sts_serial|  16|   100000|    100|0.00000000|  FAILED
          sts_serial|  16|   100000|    100|0.00000000|  FAILED
         rgb_bitdist|   1|   100000|    100|0.00000000|  FAILED
         rgb_bitdist|   2|   100000|    100|0.00000000|  FAILED
         rgb_bitdist|   3|   100000|    100|0.00000000|  FAILED
         rgb_bitdist|   4|   100000|    100|0.00000000|  FAILED
         rgb_bitdist|   5|   100000|    100|0.00000000|  FAILED
         rgb_bitdist|   6|   100000|    100|0.00000000|  FAILED
         rgb_bitdist|   7|   100000|    100|0.00000000|  FAILED
         rgb_bitdist|   8|   100000|    100|0.00000000|  FAILED
         rgb_bitdist|   9|   100000|    100|0.00000000|  FAILED
         rgb_bitdist|  10|   100000|    100|0.00000000|  FAILED
         rgb_bitdist|  11|   100000|    100|0.00000000|  FAILED
         rgb_bitdist|  12|   100000|    100|0.00000000|  FAILED
rgb_minimum_distance|   2|    10000|   1000|0.00000000|  FAILED
rgb_minimum_distance|   3|    10000|   1000|0.00000000|  FAILED
rgb_minimum_distance|   4|    10000|   1000|0.00000000|  FAILED
rgb_minimum_distance|   5|    10000|   1000|0.00000000|  FAILED
   rgb_permutations|   2|   100000|    100|0.98803991|  PASSED
   rgb_permutations|   3|   100000|    100|0.28322826|  PASSED
   rgb_permutations|   4|   100000|    100|0.80373589|  PASSED
   rgb_permutations|   5|   100000|    100|0.83478854|  PASSED
     rgb_lagged_sum|   0|  1000000|    100|0.00000000|  FAILED
     rgb_lagged_sum|   1|  1000000|    100|0.00000000|  FAILED
     rgb_lagged_sum|   2|  1000000|    100|0.00000000|  FAILED
     rgb_lagged_sum|   3|  1000000|    100|0.00000000|  FAILED
     rgb_lagged_sum|   4|  1000000|    100|0.00000000|  FAILED
     rgb_lagged_sum|   5|  1000000|    100|0.00000000|  FAILED
     rgb_lagged_sum|   6|  1000000|    100|0.00000000|  FAILED
     rgb_lagged_sum|   7|  1000000|    100|0.00000000|  FAILED
     rgb_lagged_sum|   8|  1000000|    100|0.00000000|  FAILED
     rgb_lagged_sum|   9|  1000000|    100|0.00000000|  FAILED
     rgb_lagged_sum|  10|  1000000|    100|0.00000000|  FAILED
     rgb_lagged_sum|  11|  1000000|    100|0.00000000|  FAILED
     rgb_lagged_sum|  12|  1000000|    100|0.00000000|  FAILED
     rgb_lagged_sum|  13|  1000000|    100|0.00000000|  FAILED
     rgb_lagged_sum|  14|  1000000|    100|0.00000000|  FAILED
     rgb_lagged_sum|  15|  1000000|    100|0.00000000|  FAILED
     rgb_lagged_sum|  16|  1000000|    100|0.00000000|  FAILED
     rgb_lagged_sum|  17|  1000000|    100|0.00000000|  FAILED
     rgb_lagged_sum|  18|  1000000|    100|0.00000000|  FAILED
     rgb_lagged_sum|  19|  1000000|    100|0.00000000|  FAILED
     rgb_lagged_sum|  20|  1000000|    100|0.00000000|  FAILED
     rgb_lagged_sum|  21|  1000000|    100|0.00000000|  FAILED
     rgb_lagged_sum|  22|  1000000|    100|0.00000000|  FAILED
     rgb_lagged_sum|  23|  1000000|    100|0.00000000|  FAILED
     rgb_lagged_sum|  24|  1000000|    100|0.00000000|  FAILED
```

```
     rgb_lagged_sum|  25|   1000000|     100|0.00000000|  FAILED
     rgb_lagged_sum|  26|   1000000|     100|0.00000000|  FAILED
     rgb_lagged_sum|  27|   1000000|     100|0.00000000|  FAILED
     rgb_lagged_sum|  28|   1000000|     100|0.00000000|  FAILED
     rgb_lagged_sum|  29|   1000000|     100|0.00000000|  FAILED
     rgb_lagged_sum|  30|   1000000|     100|0.00000000|  FAILED
     rgb_lagged_sum|  31|   1000000|     100|0.00000000|  FAILED
     rgb_lagged_sum|  32|   1000000|     100|0.00000000|  FAILED
    rgb_kstest_test|   0|     10000|    1000|0.00000000|  FAILED
    dab_bytedistrib|   0|  51200000|       1|0.00000000|  FAILED
            dab_dct| 256|     50000|       1|0.00000000|  FAILED
Preparing to run test 207.  ntuple = 0
       dab_filltree|  32|  15000000|       1|0.00554687|  PASSED
       dab_filltree|  32|  15000000|       1|0.01019370|  PASSED
Preparing to run test 208.  ntuple = 0
      dab_filltree2|   0|   5000000|       1|0.00000000|  FAILED
      dab_filltree2|   1|   5000000|       1|0.00000000|  FAILED
Preparing to run test 209.  ntuple = 0
       dab_monobit2|  12|  65000000|       1|1.00000000|  FAILED
```

Risultati del Test di Die-Hard sul un file di testo di 100MB generato in modo random con buttefly

```
#=============================================================================
========#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown         #
#=============================================================================
========#
   rng_name    |           filename             |rands/second|
        mt19937|               butterfly_100MB|  4.90e+07  |
#=============================================================================
========#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================
========#
   diehard_birthdays|   0|       100|     100|0.84225841|  PASSED
      diehard_operm5|   0|   1000000|     100|0.37039320|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.32963273|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.72375507|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.58536343|  PASSED
```

```
       diehard_opso|   0|   2097152|     100|0.79720823|  PASSED
       diehard_oqso|   0|   2097152|     100|0.63031273|  PASSED
        diehard_dna|   0|   2097152|     100|0.56330222|  PASSED
diehard_count_1s_str|   0|    256000|     100|0.19180751|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.29507863|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.12845620|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.62607653|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.91622238|  PASSED
     diehard_squeeze|   0|    100000|     100|0.87902338|  PASSED
       diehard_sums|   0|       100|     100|0.01616922|  PASSED
       diehard_runs|   0|    100000|     100|0.52033887|  PASSED
       diehard_runs|   0|    100000|     100|0.46191838|  PASSED
      diehard_craps|   0|    200000|     100|0.21597051|  PASSED
      diehard_craps|   0|    200000|     100|0.99532663|   WEAK
 marsaglia_tsang_gcd|   0|  10000000|     100|0.77669282|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.03801824|  PASSED
       sts_monobit|   1|    100000|     100|0.15428952|  PASSED
          sts_runs|   2|    100000|     100|0.33315170|  PASSED
        sts_serial|   1|    100000|     100|0.40787222|  PASSED
        sts_serial|   2|    100000|     100|0.62039791|  PASSED
        sts_serial|   3|    100000|     100|0.42890336|  PASSED
        sts_serial|   3|    100000|     100|0.66060597|  PASSED
        sts_serial|   4|    100000|     100|0.85938290|  PASSED
        sts_serial|   4|    100000|     100|0.69776305|  PASSED
        sts_serial|   5|    100000|     100|0.46511940|  PASSED
        sts_serial|   5|    100000|     100|0.82945097|  PASSED
        sts_serial|   6|    100000|     100|0.79692023|  PASSED
        sts_serial|   6|    100000|     100|0.91685369|  PASSED
        sts_serial|   7|    100000|     100|0.77722222|  PASSED
        sts_serial|   7|    100000|     100|0.69098126|  PASSED
        sts_serial|   8|    100000|     100|0.80221227|  PASSED
        sts_serial|   8|    100000|     100|0.92281124|  PASSED
        sts_serial|   9|    100000|     100|0.66211712|  PASSED
        sts_serial|   9|    100000|     100|0.36354008|  PASSED
        sts_serial|  10|    100000|     100|0.97439307|  PASSED
        sts_serial|  10|    100000|     100|0.92141158|  PASSED
        sts_serial|  11|    100000|     100|0.67161482|  PASSED
        sts_serial|  11|    100000|     100|0.80405976|  PASSED
        sts_serial|  12|    100000|     100|0.50517585|  PASSED
        sts_serial|  12|    100000|     100|0.07468727|  PASSED
        sts_serial|  13|    100000|     100|0.56239688|  PASSED
        sts_serial|  13|    100000|     100|0.37852538|  PASSED
        sts_serial|  14|    100000|     100|0.74028064|  PASSED
        sts_serial|  14|    100000|     100|0.32143501|  PASSED
        sts_serial|  15|    100000|     100|0.98004003|  PASSED
        sts_serial|  15|    100000|     100|0.95033848|  PASSED
        sts_serial|  16|    100000|     100|0.33779247|  PASSED
        sts_serial|  16|    100000|     100|0.34392116|  PASSED
```

```
        rgb_bitdist|  1|    100000|     100|0.77120476|  PASSED
        rgb_bitdist|  2|    100000|     100|0.24848340|  PASSED
        rgb_bitdist|  3|    100000|     100|0.54589464|  PASSED
        rgb_bitdist|  4|    100000|     100|0.80823596|  PASSED
        rgb_bitdist|  5|    100000|     100|0.33677383|  PASSED
        rgb_bitdist|  6|    100000|     100|0.06092379|  PASSED
        rgb_bitdist|  7|    100000|     100|0.71253547|  PASSED
        rgb_bitdist|  8|    100000|     100|0.12555908|  PASSED
        rgb_bitdist|  9|    100000|     100|0.94036777|  PASSED
        rgb_bitdist| 10|    100000|     100|0.98023805|  PASSED
        rgb_bitdist| 11|    100000|     100|0.75768304|  PASSED
        rgb_bitdist| 12|    100000|     100|0.86181297|  PASSED
rgb_minimum_distance|  2|     10000|    1000|0.05592572|  PASSED
rgb_minimum_distance|  3|     10000|    1000|0.67553176|  PASSED
rgb_minimum_distance|  4|     10000|    1000|0.64533200|  PASSED
rgb_minimum_distance|  5|     10000|    1000|0.36049541|  PASSED
   rgb_permutations|  2|    100000|     100|0.94244312|  PASSED
   rgb_permutations|  3|    100000|     100|0.96133467|  PASSED
   rgb_permutations|  4|    100000|     100|0.09763819|  PASSED
   rgb_permutations|  5|    100000|     100|0.71546255|  PASSED
     rgb_lagged_sum|  0|   1000000|     100|0.51619811|  PASSED
     rgb_lagged_sum|  1|   1000000|     100|0.94585346|  PASSED
     rgb_lagged_sum|  2|   1000000|     100|0.61573272|  PASSED
     rgb_lagged_sum|  3|   1000000|     100|0.31671721|  PASSED
     rgb_lagged_sum|  4|   1000000|     100|0.20362127|  PASSED
     rgb_lagged_sum|  5|   1000000|     100|0.12438149|  PASSED
     rgb_lagged_sum|  6|   1000000|     100|0.83976823|  PASSED
     rgb_lagged_sum|  7|   1000000|     100|0.18549841|  PASSED
     rgb_lagged_sum|  8|   1000000|     100|0.11920042|  PASSED
     rgb_lagged_sum|  9|   1000000|     100|0.08223686|  PASSED
     rgb_lagged_sum| 10|   1000000|     100|0.21231934|  PASSED
     rgb_lagged_sum| 11|   1000000|     100|0.99631437|   WEAK
     rgb_lagged_sum| 12|   1000000|     100|0.64370947|  PASSED
     rgb_lagged_sum| 13|   1000000|     100|0.62163019|  PASSED
     rgb_lagged_sum| 14|   1000000|     100|0.36456645|  PASSED
     rgb_lagged_sum| 15|   1000000|     100|0.65123672|  PASSED
     rgb_lagged_sum| 16|   1000000|     100|0.23434376|  PASSED
     rgb_lagged_sum| 17|   1000000|     100|0.35461975|  PASSED
     rgb_lagged_sum| 18|   1000000|     100|0.96937437|  PASSED
     rgb_lagged_sum| 19|   1000000|     100|0.92879167|  PASSED
     rgb_lagged_sum| 20|   1000000|     100|0.45244515|  PASSED
     rgb_lagged_sum| 21|   1000000|     100|0.31603280|  PASSED
     rgb_lagged_sum| 22|   1000000|     100|0.75882388|  PASSED
     rgb_lagged_sum| 23|   1000000|     100|0.05719090|  PASSED
     rgb_lagged_sum| 24|   1000000|     100|0.43564781|  PASSED
     rgb_lagged_sum| 25|   1000000|     100|0.61451403|  PASSED
     rgb_lagged_sum| 26|   1000000|     100|0.40939773|  PASSED
     rgb_lagged_sum| 27|   1000000|     100|0.39678578|  PASSED
```

```
     rgb_lagged_sum|  28|   1000000|     100|0.28825349|  PASSED
     rgb_lagged_sum|  29|   1000000|     100|0.45878396|  PASSED
     rgb_lagged_sum|  30|   1000000|     100|0.70610372|  PASSED
     rgb_lagged_sum|  31|   1000000|     100|0.26014710|  PASSED
     rgb_lagged_sum|  32|   1000000|     100|0.46752209|  PASSED
    rgb_kstest_test|   0|     10000|    1000|0.27613376|  PASSED
    dab_bytedistrib|   0|  51200000|       1|0.11843955|  PASSED
            dab_dct| 256|     50000|       1|0.10296972|  PASSED
Preparing to run test 207.  ntuple = 0
       dab_filltree|  32|  15000000|       1|0.81812004|  PASSED
       dab_filltree|  32|  15000000|       1|0.45692177|  PASSED
Preparing to run test 208.  ntuple = 0
      dab_filltree2|   0|   5000000|       1|0.13837738|  PASSED
      dab_filltree2|   1|   5000000|       1|0.56134388|  PASSED
Preparing to run test 209.  ntuple = 0
       dab_monobit2|  12|  65000000|       1|0.56020359|  PASSED
```

Risultati del Test di Die-Hard sul un file di testo di 100MB cifrato con l'algoritmo AES

```
#=============================================================================
========#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================
========#
   rng_name    |           filename             |rands/second|
        mt19937|        encrypted_AES_100MB.txt|  1.08e+08  |
#=============================================================================
========#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================
========#
   diehard_birthdays|   0|       100|     100|0.55475862|  PASSED
      diehard_operm5|   0|   1000000|     100|0.14447443|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.96095439|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.39017282|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.92874113|  PASSED
        diehard_opso|   0|   2097152|     100|0.47876260|  PASSED
        diehard_oqso|   0|   2097152|     100|0.70311549|  PASSED
         diehard_dna|   0|   2097152|     100|0.12437192|  PASSED
diehard_count_1s_str|   0|    256000|     100|0.42047235|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.93548257|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.04820412|  PASSED
```

```
   diehard_2dsphere|  2|     8000|    100|0.41843422|  PASSED
   diehard_3dsphere|  3|     4000|    100|0.65084154|  PASSED
    diehard_squeeze|  0|   100000|    100|0.03668673|  PASSED
       diehard_sums|  0|      100|    100|0.08256669|  PASSED
       diehard_runs|  0|   100000|    100|0.79444805|  PASSED
       diehard_runs|  0|   100000|    100|0.66159705|  PASSED
      diehard_craps|  0|   200000|    100|0.82157510|  PASSED
      diehard_craps|  0|   200000|    100|0.61817747|  PASSED
marsaglia_tsang_gcd|  0| 10000000|    100|0.92169375|  PASSED
marsaglia_tsang_gcd|  0| 10000000|    100|0.53084549|  PASSED
       sts_monobit|  1|   100000|    100|0.81445699|  PASSED
         sts_runs|  2|   100000|    100|0.24048197|  PASSED
        sts_serial|  1|   100000|    100|0.48448385|  PASSED
        sts_serial|  2|   100000|    100|0.14748328|  PASSED
        sts_serial|  3|   100000|    100|0.59705862|  PASSED
        sts_serial|  3|   100000|    100|0.09324241|  PASSED
        sts_serial|  4|   100000|    100|0.90834686|  PASSED
        sts_serial|  4|   100000|    100|0.74915941|  PASSED
        sts_serial|  5|   100000|    100|0.78294941|  PASSED
        sts_serial|  5|   100000|    100|0.73987115|  PASSED
        sts_serial|  6|   100000|    100|0.66565640|  PASSED
        sts_serial|  6|   100000|    100|0.51804639|  PASSED
        sts_serial|  7|   100000|    100|0.81022953|  PASSED
        sts_serial|  7|   100000|    100|0.31971823|  PASSED
        sts_serial|  8|   100000|    100|0.72342237|  PASSED
        sts_serial|  8|   100000|    100|0.43811282|  PASSED
        sts_serial|  9|   100000|    100|0.97504386|  PASSED
        sts_serial|  9|   100000|    100|0.99967763|   WEAK
        sts_serial| 10|   100000|    100|0.47907036|  PASSED
        sts_serial| 10|   100000|    100|0.92722084|  PASSED
        sts_serial| 11|   100000|    100|0.33844341|  PASSED
        sts_serial| 11|   100000|    100|0.84010303|  PASSED
        sts_serial| 12|   100000|    100|0.17084016|  PASSED
        sts_serial| 12|   100000|    100|0.72080376|  PASSED
        sts_serial| 13|   100000|    100|0.73692223|  PASSED
        sts_serial| 13|   100000|    100|0.89048690|  PASSED
        sts_serial| 14|   100000|    100|0.96499157|  PASSED
        sts_serial| 14|   100000|    100|0.47750380|  PASSED
        sts_serial| 15|   100000|    100|0.84360711|  PASSED
        sts_serial| 15|   100000|    100|0.46076020|  PASSED
        sts_serial| 16|   100000|    100|0.85962795|  PASSED
        sts_serial| 16|   100000|    100|0.65947674|  PASSED
       rgb_bitdist|  1|   100000|    100|0.20857435|  PASSED
       rgb_bitdist|  2|   100000|    100|0.84468915|  PASSED
       rgb_bitdist|  3|   100000|    100|0.94857905|  PASSED
       rgb_bitdist|  4|   100000|    100|0.90281404|  PASSED
       rgb_bitdist|  5|   100000|    100|0.32575523|  PASSED
       rgb_bitdist|  6|   100000|    100|0.64724893|  PASSED
```

```
        rgb_bitdist|   7|    100000|     100|0.19803117|  PASSED
        rgb_bitdist|   8|    100000|     100|0.54592944|  PASSED
        rgb_bitdist|   9|    100000|     100|0.82519258|  PASSED
        rgb_bitdist|  10|    100000|     100|0.55390557|  PASSED
        rgb_bitdist|  11|    100000|     100|0.60526930|  PASSED
        rgb_bitdist|  12|    100000|     100|0.96268730|  PASSED
rgb_minimum_distance|   2|     10000|    1000|0.21543552|  PASSED
rgb_minimum_distance|   3|     10000|    1000|0.93289059|  PASSED
rgb_minimum_distance|   4|     10000|    1000|0.91978256|  PASSED
rgb_minimum_distance|   5|     10000|    1000|0.46253161|  PASSED
   rgb_permutations|   2|    100000|     100|0.64397283|  PASSED
   rgb_permutations|   3|    100000|     100|0.11461857|  PASSED
   rgb_permutations|   4|    100000|     100|0.04699421|  PASSED
   rgb_permutations|   5|    100000|     100|0.29532672|  PASSED
     rgb_lagged_sum|   0|   1000000|     100|0.30002777|  PASSED
     rgb_lagged_sum|   1|   1000000|     100|0.77534140|  PASSED
     rgb_lagged_sum|   2|   1000000|     100|0.58660846|  PASSED
     rgb_lagged_sum|   3|   1000000|     100|0.86225078|  PASSED
     rgb_lagged_sum|   4|   1000000|     100|0.13459416|  PASSED
     rgb_lagged_sum|   5|   1000000|     100|0.80439866|  PASSED
     rgb_lagged_sum|   6|   1000000|     100|0.69154991|  PASSED
     rgb_lagged_sum|   7|   1000000|     100|0.24364522|  PASSED
     rgb_lagged_sum|   8|   1000000|     100|0.79195952|  PASSED
     rgb_lagged_sum|   9|   1000000|     100|0.77636408|  PASSED
     rgb_lagged_sum|  10|   1000000|     100|0.95526453|  PASSED
     rgb_lagged_sum|  11|   1000000|     100|0.64596548|  PASSED
     rgb_lagged_sum|  12|   1000000|     100|0.15915302|  PASSED
     rgb_lagged_sum|  13|   1000000|     100|0.54630179|  PASSED
     rgb_lagged_sum|  14|   1000000|     100|0.34796947|  PASSED
     rgb_lagged_sum|  15|   1000000|     100|0.75060606|  PASSED
     rgb_lagged_sum|  16|   1000000|     100|0.72087440|  PASSED
     rgb_lagged_sum|  17|   1000000|     100|0.26171861|  PASSED
     rgb_lagged_sum|  18|   1000000|     100|0.93702118|  PASSED
     rgb_lagged_sum|  19|   1000000|     100|0.58068108|  PASSED
     rgb_lagged_sum|  20|   1000000|     100|0.71242922|  PASSED
     rgb_lagged_sum|  21|   1000000|     100|0.94885248|  PASSED
     rgb_lagged_sum|  22|   1000000|     100|0.76926784|  PASSED
     rgb_lagged_sum|  23|   1000000|     100|0.64512592|  PASSED
     rgb_lagged_sum|  24|   1000000|     100|0.30152964|  PASSED
     rgb_lagged_sum|  25|   1000000|     100|0.99800978|   WEAK
     rgb_lagged_sum|  26|   1000000|     100|0.49130165|  PASSED
     rgb_lagged_sum|  27|   1000000|     100|0.84173138|  PASSED
     rgb_lagged_sum|  28|   1000000|     100|0.96777903|  PASSED
     rgb_lagged_sum|  29|   1000000|     100|0.82960813|  PASSED
     rgb_lagged_sum|  30|   1000000|     100|0.73773437|  PASSED
     rgb_lagged_sum|  31|   1000000|     100|0.60281214|  PASSED
     rgb_lagged_sum|  32|   1000000|     100|0.14221317|  PASSED
    rgb_kstest_test|   0|     10000|    1000|0.48477075|  PASSED
```

```
    dab_bytedistrib|   0|  51200000|       1|0.13325536|  PASSED
          dab_dct| 256|     50000|       1|0.23564834|  PASSED
Preparing to run test 207.  ntuple = 0
       dab_filltree|  32|  15000000|       1|0.75169882|  PASSED
       dab_filltree|  32|  15000000|       1|0.47504853|  PASSED
Preparing to run test 208.  ntuple = 0
      dab_filltree2|   0|   5000000|       1|0.60525402|  PASSED
      dab_filltree2|   1|   5000000|       1|0.36537758|  PASSED
Preparing to run test 209.  ntuple = 0
       dab_monobit2|  12|  65000000|       1|0.67894710|  PASSED
```

Risultati del Test di Die-Hard sul un file di testo di 100MB cifrato con BLOWFISH

```
#=============================================================================
========#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown         #
#=============================================================================
========#
   rng_name    |           filename             |rands/second|
       mt19937|    encrypted_BLOWFISH_100MB.txt|  1.06e+08  |
#=============================================================================
========#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================
========#
   diehard_birthdays|   0|       100|     100|0.95090947|  PASSED
      diehard_operm5|   0|   1000000|     100|0.73287214|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.19373020|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.00181891|   WEAK
   diehard_bitstream|   0|   2097152|     100|0.92557052|  PASSED
        diehard_opso|   0|   2097152|     100|0.93025126|  PASSED
        diehard_oqso|   0|   2097152|     100|0.56894905|  PASSED
         diehard_dna|   0|   2097152|     100|0.13072346|  PASSED
diehard_count_1s_str|   0|    256000|     100|0.58302465|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.23584429|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.46927986|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.32958487|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.53819772|  PASSED
     diehard_squeeze|   0|    100000|     100|0.51541533|  PASSED
        diehard_sums|   0|       100|     100|0.84129232|  PASSED
        diehard_runs|   0|    100000|     100|0.21163239|  PASSED
        diehard_runs|   0|    100000|     100|0.99707034|   WEAK
       diehard_craps|   0|    200000|     100|0.81770202|  PASSED
```

```
      diehard_craps|  0|   200000|     100|0.80562622|  PASSED
 marsaglia_tsang_gcd|  0| 10000000|     100|0.76451208|  PASSED
 marsaglia_tsang_gcd|  0| 10000000|     100|0.67805345|  PASSED
       sts_monobit|  1|   100000|     100|0.41237794|  PASSED
         sts_runs|  2|   100000|     100|0.16993245|  PASSED
        sts_serial|  1|   100000|     100|0.84151514|  PASSED
        sts_serial|  2|   100000|     100|0.21247702|  PASSED
        sts_serial|  3|   100000|     100|0.14663134|  PASSED
        sts_serial|  3|   100000|     100|0.11808858|  PASSED
        sts_serial|  4|   100000|     100|0.65910086|  PASSED
        sts_serial|  4|   100000|     100|0.98192116|  PASSED
        sts_serial|  5|   100000|     100|0.47278490|  PASSED
        sts_serial|  5|   100000|     100|0.80021370|  PASSED
        sts_serial|  6|   100000|     100|0.92276667|  PASSED
        sts_serial|  6|   100000|     100|0.61188861|  PASSED
        sts_serial|  7|   100000|     100|0.57072016|  PASSED
        sts_serial|  7|   100000|     100|0.81552029|  PASSED
        sts_serial|  8|   100000|     100|0.67385694|  PASSED
        sts_serial|  8|   100000|     100|0.22481152|  PASSED
        sts_serial|  9|   100000|     100|0.25784686|  PASSED
        sts_serial|  9|   100000|     100|0.10652390|  PASSED
        sts_serial| 10|   100000|     100|0.06621271|  PASSED
        sts_serial| 10|   100000|     100|0.39728910|  PASSED
        sts_serial| 11|   100000|     100|0.00995010|  PASSED
        sts_serial| 11|   100000|     100|0.12095054|  PASSED
        sts_serial| 12|   100000|     100|0.30042611|  PASSED
        sts_serial| 12|   100000|     100|0.65595315|  PASSED
        sts_serial| 13|   100000|     100|0.44894218|  PASSED
        sts_serial| 13|   100000|     100|0.60192220|  PASSED
        sts_serial| 14|   100000|     100|0.75894170|  PASSED
        sts_serial| 14|   100000|     100|0.20496893|  PASSED
        sts_serial| 15|   100000|     100|0.46731424|  PASSED
        sts_serial| 15|   100000|     100|0.87841309|  PASSED
        sts_serial| 16|   100000|     100|0.70572967|  PASSED
        sts_serial| 16|   100000|     100|0.62489347|  PASSED
       rgb_bitdist|  1|   100000|     100|0.45873484|  PASSED
       rgb_bitdist|  2|   100000|     100|0.24725687|  PASSED
       rgb_bitdist|  3|   100000|     100|0.88041097|  PASSED
       rgb_bitdist|  4|   100000|     100|0.95659255|  PASSED
       rgb_bitdist|  5|   100000|     100|0.38986227|  PASSED
       rgb_bitdist|  6|   100000|     100|0.94442884|  PASSED
       rgb_bitdist|  7|   100000|     100|0.48314673|  PASSED
       rgb_bitdist|  8|   100000|     100|0.84948307|  PASSED
       rgb_bitdist|  9|   100000|     100|0.09174229|  PASSED
       rgb_bitdist| 10|   100000|     100|0.76745317|  PASSED
       rgb_bitdist| 11|   100000|     100|0.14609063|  PASSED
       rgb_bitdist| 12|   100000|     100|0.24874682|  PASSED
rgb_minimum_distance|  2|    10000|    1000|0.50067491|  PASSED
```

```
rgb_minimum_distance|   3|     10000|    1000|0.11165736|  PASSED
rgb_minimum_distance|   4|     10000|    1000|0.02034843|  PASSED
rgb_minimum_distance|   5|     10000|    1000|0.29020093|  PASSED
   rgb_permutations|   2|    100000|     100|0.72324277|  PASSED
   rgb_permutations|   3|    100000|     100|0.91704846|  PASSED
   rgb_permutations|   4|    100000|     100|0.21609825|  PASSED
   rgb_permutations|   5|    100000|     100|0.97518021|  PASSED
     rgb_lagged_sum|   0|   1000000|     100|0.72283220|  PASSED
     rgb_lagged_sum|   1|   1000000|     100|0.38058599|  PASSED
     rgb_lagged_sum|   2|   1000000|     100|0.50915670|  PASSED
     rgb_lagged_sum|   3|   1000000|     100|0.59542358|  PASSED
     rgb_lagged_sum|   4|   1000000|     100|0.68293761|  PASSED
     rgb_lagged_sum|   5|   1000000|     100|0.99519029|   WEAK
     rgb_lagged_sum|   6|   1000000|     100|0.42706487|  PASSED
     rgb_lagged_sum|   7|   1000000|     100|0.97917695|  PASSED
     rgb_lagged_sum|   8|   1000000|     100|0.49816899|  PASSED
     rgb_lagged_sum|   9|   1000000|     100|0.99455596|  PASSED
     rgb_lagged_sum|  10|   1000000|     100|0.82650532|  PASSED
     rgb_lagged_sum|  11|   1000000|     100|0.25720233|  PASSED
     rgb_lagged_sum|  12|   1000000|     100|0.95305533|  PASSED
     rgb_lagged_sum|  13|   1000000|     100|0.74440695|  PASSED
     rgb_lagged_sum|  14|   1000000|     100|0.10295513|  PASSED
     rgb_lagged_sum|  15|   1000000|     100|0.39462099|  PASSED
     rgb_lagged_sum|  16|   1000000|     100|0.60594050|  PASSED
     rgb_lagged_sum|  17|   1000000|     100|0.55736519|  PASSED
     rgb_lagged_sum|  18|   1000000|     100|0.39333977|  PASSED
     rgb_lagged_sum|  19|   1000000|     100|0.38448281|  PASSED
     rgb_lagged_sum|  20|   1000000|     100|0.97837507|  PASSED
     rgb_lagged_sum|  21|   1000000|     100|0.66374697|  PASSED
     rgb_lagged_sum|  22|   1000000|     100|0.27279362|  PASSED
     rgb_lagged_sum|  23|   1000000|     100|0.74157411|  PASSED
     rgb_lagged_sum|  24|   1000000|     100|0.41000856|  PASSED
     rgb_lagged_sum|  25|   1000000|     100|0.61881628|  PASSED
     rgb_lagged_sum|  26|   1000000|     100|0.96120476|  PASSED
     rgb_lagged_sum|  27|   1000000|     100|0.21895030|  PASSED
     rgb_lagged_sum|  28|   1000000|     100|0.40419402|  PASSED
     rgb_lagged_sum|  29|   1000000|     100|0.89747440|  PASSED
     rgb_lagged_sum|  30|   1000000|     100|0.90643353|  PASSED
     rgb_lagged_sum|  31|   1000000|     100|0.76383919|  PASSED
     rgb_lagged_sum|  32|   1000000|     100|0.17770607|  PASSED
    rgb_kstest_test|   0|     10000|    1000|0.28129847|  PASSED
    dab_bytedistrib|   0|  51200000|       1|0.58674086|  PASSED
           dab_dct| 256|     50000|       1|0.27549952|  PASSED
Preparing to run test 207.  ntuple = 0
       dab_filltree|  32|  15000000|       1|0.83511124|  PASSED
       dab_filltree|  32|  15000000|       1|0.96826888|  PASSED
Preparing to run test 208.  ntuple = 0
      dab_filltree2|   0|   5000000|       1|0.76135850|  PASSED
```

```
      dab_filltree2|   1|   5000000|       1|0.78262385|  PASSED
Preparing to run test 209.  ntuple = 0
       dab_monobit2|  12|  65000000|       1|0.20230609|  PASSED
```

Risultati del Test di Die-Hard sul un file di testo di 100MB cifrato con ECL

```
#=============================================================================
========#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown         #
#=============================================================================
========#
   rng_name    |           filename             |rands/second|
       mt19937|        encrypted_ECL_100MB.txt|  1.06e+08  |
#=============================================================================
========#
       test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================
========#
   diehard_birthdays|   0|       100|     100|0.75959029|  PASSED
      diehard_operm5|   0|   1000000|     100|0.33187238|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.75689182|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.19180854|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.95931380|  PASSED
        diehard_opso|   0|   2097152|     100|0.50947183|  PASSED
        diehard_oqso|   0|   2097152|     100|0.34378596|  PASSED
         diehard_dna|   0|   2097152|     100|0.73303581|  PASSED
diehard_count_1s_str|   0|    256000|     100|0.40492670|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.44986143|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.43004782|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.68822206|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.55967843|  PASSED
     diehard_squeeze|   0|    100000|     100|0.20866410|  PASSED
        diehard_sums|   0|       100|     100|0.40443859|  PASSED
        diehard_runs|   0|    100000|     100|0.76771654|  PASSED
        diehard_runs|   0|    100000|     100|0.85060638|  PASSED
       diehard_craps|   0|    200000|     100|0.72885933|  PASSED
       diehard_craps|   0|    200000|     100|0.47906536|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.78940305|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.73595815|  PASSED
         sts_monobit|   1|    100000|     100|0.61464771|  PASSED
            sts_runs|   2|    100000|     100|0.63276345|  PASSED
          sts_serial|   1|    100000|     100|0.05400285|  PASSED
          sts_serial|   2|    100000|     100|0.10805684|  PASSED
```

```
          sts_serial|  3|   100000|    100|0.53418875|  PASSED
          sts_serial|  3|   100000|    100|0.79185556|  PASSED
          sts_serial|  4|   100000|    100|0.09839596|  PASSED
          sts_serial|  4|   100000|    100|0.57421150|  PASSED
          sts_serial|  5|   100000|    100|0.01939543|  PASSED
          sts_serial|  5|   100000|    100|0.30494866|  PASSED
          sts_serial|  6|   100000|    100|0.26314797|  PASSED
          sts_serial|  6|   100000|    100|0.87699164|  PASSED
          sts_serial|  7|   100000|    100|0.80857781|  PASSED
          sts_serial|  7|   100000|    100|0.22515202|  PASSED
          sts_serial|  8|   100000|    100|0.75027920|  PASSED
          sts_serial|  8|   100000|    100|0.56356045|  PASSED
          sts_serial|  9|   100000|    100|0.20242836|  PASSED
          sts_serial|  9|   100000|    100|0.40261186|  PASSED
          sts_serial| 10|   100000|    100|0.83856070|  PASSED
          sts_serial| 10|   100000|    100|0.54139695|  PASSED
          sts_serial| 11|   100000|    100|0.24858297|  PASSED
          sts_serial| 11|   100000|    100|0.66875041|  PASSED
          sts_serial| 12|   100000|    100|0.75445841|  PASSED
          sts_serial| 12|   100000|    100|0.15464802|  PASSED
          sts_serial| 13|   100000|    100|0.58984652|  PASSED
          sts_serial| 13|   100000|    100|0.87565310|  PASSED
          sts_serial| 14|   100000|    100|0.02818870|  PASSED
          sts_serial| 14|   100000|    100|0.15783955|  PASSED
          sts_serial| 15|   100000|    100|0.94898539|  PASSED
          sts_serial| 15|   100000|    100|0.91387200|  PASSED
          sts_serial| 16|   100000|    100|0.43479622|  PASSED
          sts_serial| 16|   100000|    100|0.24425515|  PASSED
         rgb_bitdist|  1|   100000|    100|0.43719463|  PASSED
         rgb_bitdist|  2|   100000|    100|0.19380689|  PASSED
         rgb_bitdist|  3|   100000|    100|0.51378115|  PASSED
         rgb_bitdist|  4|   100000|    100|0.97776294|  PASSED
         rgb_bitdist|  5|   100000|    100|0.40221365|  PASSED
         rgb_bitdist|  6|   100000|    100|0.08543702|  PASSED
         rgb_bitdist|  7|   100000|    100|0.62816055|  PASSED
         rgb_bitdist|  8|   100000|    100|0.90798243|  PASSED
         rgb_bitdist|  9|   100000|    100|0.96162680|  PASSED
         rgb_bitdist| 10|   100000|    100|0.34151855|  PASSED
         rgb_bitdist| 11|   100000|    100|0.58642810|  PASSED
         rgb_bitdist| 12|   100000|    100|0.11017742|  PASSED
rgb_minimum_distance|  2|    10000|   1000|0.99677014|   WEAK
rgb_minimum_distance|  3|    10000|   1000|0.48727076|  PASSED
rgb_minimum_distance|  4|    10000|   1000|0.11506937|  PASSED
rgb_minimum_distance|  5|    10000|   1000|0.03363645|  PASSED
   rgb_permutations|  2|   100000|    100|0.15116523|  PASSED
   rgb_permutations|  3|   100000|    100|0.24019011|  PASSED
   rgb_permutations|  4|   100000|    100|0.15833106|  PASSED
   rgb_permutations|  5|   100000|    100|0.95367714|  PASSED
```

```
     rgb_lagged_sum|   0|   1000000|     100|0.68547039|  PASSED
     rgb_lagged_sum|   1|   1000000|     100|0.52903363|  PASSED
     rgb_lagged_sum|   2|   1000000|     100|0.12946839|  PASSED
     rgb_lagged_sum|   3|   1000000|     100|0.98587206|  PASSED
     rgb_lagged_sum|   4|   1000000|     100|0.32470532|  PASSED
     rgb_lagged_sum|   5|   1000000|     100|0.02587038|  PASSED
     rgb_lagged_sum|   6|   1000000|     100|0.14008360|  PASSED
     rgb_lagged_sum|   7|   1000000|     100|0.50469772|  PASSED
     rgb_lagged_sum|   8|   1000000|     100|0.27415294|  PASSED
     rgb_lagged_sum|   9|   1000000|     100|0.59704990|  PASSED
     rgb_lagged_sum|  10|   1000000|     100|0.03546625|  PASSED
     rgb_lagged_sum|  11|   1000000|     100|0.99870747|   WEAK
     rgb_lagged_sum|  12|   1000000|     100|0.07775616|  PASSED
     rgb_lagged_sum|  13|   1000000|     100|0.73236097|  PASSED
     rgb_lagged_sum|  14|   1000000|     100|0.97160502|  PASSED
     rgb_lagged_sum|  15|   1000000|     100|0.28199373|  PASSED
     rgb_lagged_sum|  16|   1000000|     100|0.33475142|  PASSED
     rgb_lagged_sum|  17|   1000000|     100|0.76375855|  PASSED
     rgb_lagged_sum|  18|   1000000|     100|0.14010203|  PASSED
     rgb_lagged_sum|  19|   1000000|     100|0.91632343|  PASSED
     rgb_lagged_sum|  20|   1000000|     100|0.66208408|  PASSED
     rgb_lagged_sum|  21|   1000000|     100|0.14743662|  PASSED
     rgb_lagged_sum|  22|   1000000|     100|0.37643737|  PASSED
     rgb_lagged_sum|  23|   1000000|     100|0.27757621|  PASSED
     rgb_lagged_sum|  24|   1000000|     100|0.40828109|  PASSED
     rgb_lagged_sum|  25|   1000000|     100|0.25071208|  PASSED
     rgb_lagged_sum|  26|   1000000|     100|0.56363518|  PASSED
     rgb_lagged_sum|  27|   1000000|     100|0.95067628|  PASSED
     rgb_lagged_sum|  28|   1000000|     100|0.35043140|  PASSED
     rgb_lagged_sum|  29|   1000000|     100|0.78661343|  PASSED
     rgb_lagged_sum|  30|   1000000|     100|0.16418628|  PASSED
     rgb_lagged_sum|  31|   1000000|     100|0.58298474|  PASSED
     rgb_lagged_sum|  32|   1000000|     100|0.35268963|  PASSED
    rgb_kstest_test|   0|     10000|    1000|0.22326145|  PASSED
    dab_bytedistrib|   0|  51200000|       1|0.18218663|  PASSED
           dab_dct| 256|     50000|       1|0.54929145|  PASSED
Preparing to run test 207.  ntuple = 0
       dab_filltree|  32|  15000000|       1|0.77053673|  PASSED
       dab_filltree|  32|  15000000|       1|0.40915376|  PASSED
Preparing to run test 208.  ntuple = 0
      dab_filltree2|   0|   5000000|       1|0.85295176|  PASSED
      dab_filltree2|   1|   5000000|       1|0.56945601|  PASSED
Preparing to run test 209.  ntuple = 0
       dab_monobit2|  12|  65000000|       1|0.99519805|   WEAK
```