Sicurezza della crittografia a chiave pubblica: RSA e ECC

L. Gesuele

1 Marzo 2020

1 Introduzione

La crittografia a chiave pubblica è un tipo di crittografia in cui la chiave di cifratura e la chiave di decifrazione sono diverse tra loro. Ciascun utente che desidera ricevere un messaggio sceglie la propria chiave di cifratura e la rende pubblica, permettendo, così, a chiunque di mandargli un messaggio cifrato; al contempo crea una chiave di decifrazione che mantiene invece privata, di cui si servirà per decifrare i messaggi ricevuti. L'intento di questo articolo è di presentare due dei più famosi cifrari asimmetrici, RSA e ElGamal su curve ellittiche prime e di analizzarne la sicurezza al fine di confrontare l'enorme differenza che vi è tra i due, a parità di dimensioni delle chiavi.

1.1 Le funzioni One-way Trap-door

La sicurezza e l'efficienza della crittografia asimmetrica dipendono fortemente dalle funzioni di cifratura $\mathcal{C}$ e di decifrazione $\mathcal{D}$ e dalla relazione che esiste tra le chiavi *k[pub]* e *k[prv]*, ovvero:

1. dati *msg* e *k[pub]* per il mittente è facile calcolare $crt = \mathcal{C}(msg,k[pub])$ e per il destinatario è facile calcolare $msg=\mathcal{D}(crt,k[prv])$;
2. la coppia $\langle k[pub], k[prv]\rangle$ deve essere facile da generare e due utenti non devono generare la stessa chiave;
3. anche conoscendo *crt, k[pub]*, $\mathcal{C}$ e $\mathcal{D}$, per un crittoanalista è difficile risalire a *msg*.

Quando si dice *facile* e *difficile* si intende **computazionalmente facile** e **computazionalmente difficile**. In linea generale, un problema si dice **computazionalmente facile** da calcolare quando esiste un algoritmo di complessità polinomiale in grado di risolvere il problema; analogamente, un problema si dice **computazionalmente difficile** da calcolare se non esiste un algoritmo di complessità polinomiale in grado di risolvere il problema. Per poter rispettare queste proprietà, la funzione di cifratura $\mathcal{C}$ deve essere una funzione $f(x)$ detta *one-way*, ovvero una funzione che sia computazionalmente facile da calcolare per ogni x, ma che sia computazionalmente difficile da invertire se non mediante un meccanismo segreto detto *trap-door*. Nella crittografia asimmetrica, la conoscenza di *k[pub]* non fornisce indicazioni sul meccanismo segreto di $\mathcal{C}$, che è invece svelato tramite *k[prv]* quando questa viene inserita nella funzione $\mathcal{D}$. Non è detto che tali funzioni esistano in matematica, anche se dopo numerosi studi crittografici sono state trovate delle funzioni che soddisfano tali caratteristiche basandosi su proprietà dell'algebra modulare e della teoria dei numeri. Le più rilevanti per la crittografia sono: *fattorizzazione, calcolo del logaritmo discreto, calcolo della radice in modulo.* Nel corso dell'articolo ci concentreremo sui primi due.

2 Il cifrario RSA

Il cifrario RSA è un cifrario asimmetrico nato alla fine degli anni *'70* che prende il nome dalle iniziali dei suoi creatori. Ha avuto un enorme successo nel mondo della crittografia per via della sua semplicità strutturale e per via della sua resistenza agli attacchi algebrici. Questi sono anche i principali motivi per cui tutt'oggi è utilizzato in moltissime applicazioni, tra cui servizi di posta elettronica, sistemi di messaggistica istantanea o persino sportelli automatici delle banche. Analizziamone la struttura:

Creazione della chiave. Se un utente U desidera ricevere un messaggio svolge le seguenti operazioni:

- sceglie due numeri primi a caso molto grandi p e q;
- calcola $n = p \times q$ e la funzione di Eulero $\phi(n) = (p-1)(q-1)$;
- sceglie un intero $e < \phi(n)$ primo con esso
- calcola $d = e^{-1}$ *mod* $\phi(n)$ che esiste perchè e e $\phi(n)$ sono coprimi
- rende pubblica la chiave $k[pub] = \langle e, n \rangle$ e mantiene segreta $k[prv] = \langle d \rangle$

Messaggio. Ogni messaggio è un intero $m < n$, e se si vuole cifrare un messaggio più grande viene suddiviso a blocchi.

Cifratura. Per inviare un messaggio m a U, un utente V genera c calcolando $c = \mathcal{C}(m, k[pub]) = m^e \bmod n$. Risulterà inoltre $c < n$.

Decifrazione U riceve il crittogramma c e lo decifra calcolando $m = \mathcal{D}(c, k[prv]) = c^d \bmod n$.

2.1 Dimostrazione di correttezza

Per poter dimostrare la correttezza del cifrario RSA, occorre anzitutto conoscere il teorema di Eulero, che è così formulato:

Teorema di Eulero 2.1 Per $n > 1$ e $\forall a \equiv 1 \bmod n$ si ha $a^{\phi(n)} \equiv 1 \bmod n$
Affinchè il funzionamento del cifrario sia corretto, bisogna dimostrare che la scelta di un qualunque intero $e \equiv 1 \bmod \phi(n)$ implica che la funzione $x^e \bmod n$ sia una permutazione di Z_n, poichè ciò garantirebbe l'invertibilità di $\mathcal{C}$ ossia che $\mathcal{D}(\mathcal{C}(m, k[pub]), k[prv]) = m$ ovvero $(m^e \bmod n)^d \bmod n = m$. Da qui il teorema:

Teorema 2.2. Per qualunque intero $m < n$ si ha: $(m^e \bmod n)^d \bmod n = m$, ove n, e, d sono i parametri del cifrario RSA.

Dimostrazione:

1. Se p e q non dividono m: abbiamo *mcd(m,n) = 1* e per il Teorema di Eulero abbiamo $m^{\phi(n)} \equiv 1 \bmod n$. Poichè $d = e^{-1} \bmod \phi(n)$, abbiamo $e \times d \equiv 1 \bmod \phi(n)$, ovvero $e \times d = 1 + r\phi(n)$ con $r > 0$. Abbiamo quindi $m^{ed} \bmod n = m^{1+r\phi(n)} \bmod n = m \times (m^{\phi(n)})^r \bmod n = m \times 1^r \bmod n = m$ poichè $m < n$.

2. Se p (oppure q) divide m, ma q (oppure p) non divide m: Poichè p divide m abbiamo $m \equiv m^r \equiv 0 \bmod p$, ovvero $(m^r - m) \equiv 0 \bmod p$ per ogni $r > 0$. Abbiamo quindi $m^{ed} \bmod q = m^{1+r\phi(n)} \bmod q = m \times m^{r(p-1)(q-1)} \bmod q = m \times (m^{(q-1)})^{r(p-1)} \bmod q = m \bmod q$ poichè $m^{(q-1)} \equiv 1 \bmod q$ per il Teorema di Eulero. Di conseguenza $(m^{ed} - m)$ è divisibile sia per p che per q e quindi anche per n, e deriviamo dunque la tesi $m^{ed} \equiv m \bmod n$.

2.2 Il problema della fattorizzazione

Calcolare il prodotto n di due interi p e q è computazionalmente facile in quanto richieda un tempo polinomiale nella lunghezza della loro rappresentazione. Ciò che è computazionalmente difficile da calcolare è l'inversione di tale funzione, ovvero di calcolare p e q a partire da n che è possibile se e solo se p e q sono primi. Calcolare p e q richiede tempo esponenziale, anche se non esiste una dimostrazione che questo problema appartenga alla categoria **NP-Hard**; tuttavia è molto improbabile che esista un algoritmo che è in grado di fattorizzare in tempo polinomiale. Un eventuale attacco enumerativo al cifrario RSA consisterebbe appunto nel tentare di fattorizzare n calcolando quindi p e q, così da poter ricavare $\phi(n) = (p-1)(q-1)$ e quindi ricavare la chiave segreta $d = e^{-1} \bmod \phi(n)$, con la quale poter poi decifrare ogni messaggio m.

3 Crittografia su curve ellittiche prime (ECC)

La **ECC** (**E**lliptic **C**urve **C**ryptography) è un tipo di crittografia asimmetrica basata sulle curve ellittiche definite su campi finiti. Questo metodo è stato pensato nella seconda metà degli anni *'80* da Koblitz e Miller, ed è ad oggi considerato uno standard su cui si basano diversi cifrari a chiave pubblica, tra cui il protocollo per lo scambio di chiavi **ECDH** e il protocollo per lo scambio di messaggi di **ElGamal**. Quest'ultimo è quello che analizzeremo nel dettaglio.

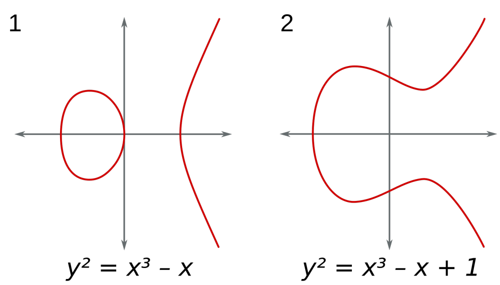
3.1 Curve ellittiche sui numeri reali

Per poter comprendere il funzionamento delle curve ellittiche prime nella crittografia, è necessario capire che cosa siano le curve ellittiche definite su R, poichè nonostante la loro inapplicabilità crittografica, vi sono alcune proprietà che si riscontrano anche nelle curve definite su Z_p.
Le curve ellittiche sui numeri reali sono delle curve algebriche definite sul campo R che soddisfano l'equazione cubica in forma normale di Weierstrass, e che comprendono l'elemento neutro $\mathcal{O}$ detto punto all'infinito sull'asse y. Più formalmente, una curva ellittica $E(a, b)$ è così definita:

- $E(a, b) = \{(x, y) \in R^2 | y^2 = x^3 + ax + b\} \cup \{\mathcal{O}\}$

dove $a, b \in R$ e dove vi è assenza di radici multiple, cuspidi o nodi, ovvero deve valere la disuguaglianza $4a^3+27b^2 \neq 0$. L'insieme dei punti di una curva ellittica $E(a, b)$ ha la struttura algebrica di un **gruppo abeliano** in quanto soddisfa le seguenti proprietà:

- **Chiusura**: $\forall P, Q \in E(a, b), P + Q \in E(a, b)$;
- **Elemento neutro**: $\forall P \in E(a, b), P + \mathcal{O} = \mathcal{O} + P = P$;
- **Inverso**: $\forall P \in E(a, b), \exists! Q = -P \in E(a, b) | P + Q = Q + P = \mathcal{O}$;
- **Associatività**: $\forall P, Q, R \in E(a, b), P + (Q + R) = (P + Q) + R$;
- **Commutatività**: $\forall P, Q \in E(a, b), P + Q = Q + P$.



La definizione dell'operazione di addizione sulle curve ellittiche è molto importante in ambito crittografico; essa ci permette di calcolare le coordinate del punto $S = P+Q$ a partire dalle coordinate di P e Q. Potremmo poi ad esempio associare un eventuale messaggio m al punto della curva P, calcolare $S = P+Q$ e spedire quindi S. Una volta ricevuto S, un destinatario che conosce Q potrebbe estrarre P calcolando $P = S-Q$ e ricavare il messaggio.Vediamo come è dunque definita la somma tra due punti.
Siano $P = (x_s, y_s)$ e $Q = (x_q, y_q), P \neq Q, P \neq -Q$. La loro somma S è data da:

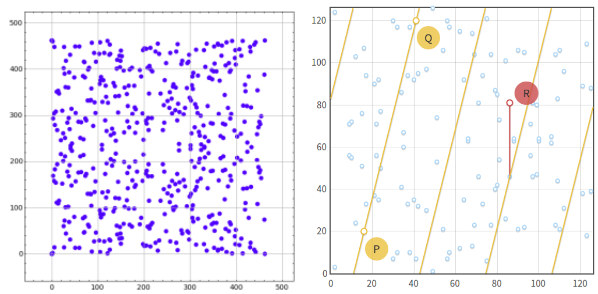
- $x_s = \lambda^2 - x_p - x_q$
- $y_s = -y_p + \lambda(x_p - x_s)$ con $\lambda = (y_q - y_p)/(x_q - x_p)$

Se $P = Q$, calcolare $S = P + Q$ equivale a raddoppiare P, ovvero $S = 2P$. Le formule algebriche restano le solite tranne ovviamente per λ, che diventa $\lambda = (3x_p^2 + a)/2y_p$. Inoltre, se $y_p = 0$ si ha che $2P = \mathcal{O}$.

3.2 Curve ellittiche prime

Le curve ellittiche prime sono quelle curve ellittiche definite sul campo Z_p e, dunque, presi $a, b \in Z_p$, la curva ellittica prima $E_p(a, b)$ è definita come l'insieme dei punti che soddisfano l'equazione:

- $E_p(a, b) = \{(x, y) \in Z_p^2 | y^2 \bmod p = (x^3 + ax + b) \bmod p\} \cup \{\mathcal{O}\}$



Come si può vedere dalle immagini sopra riportate, in questo caso le curve sono composte da punti distinti nel piano, e sono molto più interessanti per la crittografia in quanto contengono un numero finito di punti. Analogamente alle curve ellittiche definite sui reali, se il polinomio $(x^3+ax+b) \bmod p$ non ha radici multiple, ovvero soddisfa $(4a^3 + 27b^2) \bmod p \neq 0$, i punti della curva ellittica prima $E_p(a, b)$ formano un gruppo abeliano finito rispetto all'operazione di addizione. Quest'ultima è definita allo stesso modo di quella delle curve ellittiche sui reali, con l'unica accortezza di lavorare in modulo p. Un parametro importante è l'**ordine N** di una curva, ovvero il suo numero di punti. Si può inoltre dimostrare che esattamente $(p - 1)/2$ elementi di Z_p siano *residui quadratici*, ovvero:

Definizione: *In* ***teoria dei numeri****, un numero intero q si dice* ***residuo quadratico*** *modulo p se esiste un intero x tale che $x^2 \equiv q \pmod p$.*

Questa è un'altra proprietà interessante per le applicazioni crittografiche. Il seguente teorema inoltre ci consente di stabilire un limite superiore all'ordine di una curva:

Teorema di Hasse 3.1: L'ordine N di una curva ellittica $E_p(a, b)$ verifica la disuguaglianza $|N - (p + 1)| \leq 2\sqrt{p}$.

3.3 Scambio di messaggi con ElGamal su ECC

Per poter scambiare un messaggio m usufruendo delle curve ellittiche prime è necessario prima trasformare m in un punto della curva ellittica utilizzando un opportuno algoritmo come quello probabilistico di **Koblitz**, questo perchè un algoritmo di tipo deterministico polinomiale efficiente non è ancora stato trovato, ma come vedremo, l'algoritmo di Koblitz ha una probabilità di successo sufficientemente alta da ignorare ciò. Una volta effettuata tale trasformazione, il meccanismo di cifratura e decifrazione è il seguente:

Cifratura: Alice trasforma m in P_m sulla curva. sceglie un intero casuale r e calcola $V = rB$, dove B è un punto della curva sul quale gli interlocutori si sono messi d'accordo. Poi calcola $W = P_m + rP_d$ dove P_d è invece la chiave pubblica del destinatario Bob, che quest'ultimo si è calcolato tramite la propria chiave privata $P_d = k[prv]B$. Infine Alice spedisce $\langle V, W\rangle$ a Bob.

Decifrazione: Bob riceve $\langle V, W\rangle$ e ricostruisce P_m con la sua chiave privata $k[prv]$ calcolando: $W - k[prv]V = P_m + rP_d - k[prv]rB = P_m + rP_d - rP_d = P_m$ e infine trasforma il punto P_m ricavato nel messaggio m.

3.3.1 Algoritmo di Koblitz

L'algoritmo di Koblitz permette di trasformare, con alta probabilità di successo, un messaggio $m < p$ in un punto della curva $E_p(a, b)$. Utilizzando infatti m come ascissa, la probabilità di trovare un punto della curva è pari alla probabilità che $(m^3 + am + b) \bmod p$ sia un residuo quadratico, che è circa $1/2$ come abbiamo visto in precedenza. Pertanto si fissa un intero k t.c. $(m + 1)k < p$ e si considerano come potenziale ascissa gli interi $x = mk + i \; \forall i \in [0, k - 1]$. Ma poichè $(x^3 + ax + b) \bmod p$ deve eguagliare il quadrato di y, per ciascun x si prova ad estrarre la radice quadrata di $(x^3 + ax + b) \bmod p$, che è un'operazione polinomiale se p è primo. Se questa radice esiste si prende $P_m = (x, y)$ come punto sulla curva corrispondente a m, altrimenti si itera fino a che non si trova una radice o fino a che i eguagli k. Siccome la probabilità di trovare una radice è circa $1/2$, la probabilità complessiva di fallimento equivale a $(1/2)^k$, e come si può ben intuire al crescere di k la probabilità di successo aumenta. Si può dedurre inoltre che la probabilità di successo di questo algoritmo dipenda strettamente dalla lunghezza di m, che potrebbe essere problematico per messaggi molto grandi in quanto richiederebbero chiavi molto grandi. In realtà, in tal caso si scomporrebbe m in blocchi opportunamente più piccoli. Per risalire nuovamente al messaggio occorre, quindi, invertire la formula calcolando $m = \lfloor x/k \rfloor$. Qualora l'algoritmo dovesse fallire, solitamente, basta modificare leggermente il messaggio m senza comprometterne il significato.

3.4 Logaritmo discreto per le curve ellittiche

L'operazione di addizione di punti di una curva ellittica su un campo finito ha similarità con il prodotto modulare. Sia k un valore positivo, possiamo mettere in relazione l'elevamento a potenza k di un intero in modulo con la *moltiplicazione scalare* di k per il punto P. Difatti, eseguire $y = x^k \bmod p$ richiede $\Theta(\log(k))$ moltiplicazioni se si procede per quadrature successive; nelle curve ellittiche, analogamente si può calcolare $Q = kP$ con il metodo *Double and add*, che consiste nell'effettuare $\Theta(log(k))$ raddoppi e $O(\log(k))$ addizioni di punti. Formalmente, si ha $\sum_{k=0}^{z} k_i 2^i$ con $z = \lfloor \log_2(k) \rfloor$ e dove $k_z k_{z-1}..k_0$ è la rappresentazione binaria di k. Il numero intero k, se esiste, è detto *logaritmo in base P del punto Q*; per cui per calcolare, dati P e Q, il $\min\{k|Q = kP\}$ corrisponde a calcolare il *logaritmo discreto per le curve ellittiche*, che è computazionalmente difficile se i parametri della curva sono scelti opportunamente, ed occorre un metodo forza bruta esponenziale per calcolarlo. Infatti, un eventuale attacco enumerativo alla **ECC** consisterebbe appunto nel tentare di risolvere il logaritmo discreto ricavando r da B e V per poter poi calcolare $P_m = W - rP_d$ da cui poi estrarre il messaggio m.

4 Discussione sulla sicurezza dei due cifrari

Come accennato precedentemente, il cifrario RSA basa la sua sicurezza sul problema della *fattorizzazione* di n, che è considerato computazionalmente difficile se e solo se si scelgono accuratamente p e q, che devono essere degli interi primi molto grandi. Inoltre bisogna stare attenti anche al fatto che la differenza $|p - q|$ sia molto grande, altrimenti si potrebbero utilizzare degli algoritmi in grado di fattorizzare n più velocemente, basati sul fatto che $(p + q)/2$ sia prossimo a $\sqrt{n}$, che scandiscano quindi soltanto gli interi maggiori di $\sqrt{n}$. Qualsiasi cifrario sulle EC invece basa la propria sicurezza sul *calcolo del logaritmo discreto di un punto*. Nonostante ad oggi manchi una dimostrazione formale della sua difficoltà, il problema del calcolo del logaritmo discreto sulle curve ellittiche è estremamente difficile, molto più di quello della fattorizzazione. Questo perchè i gruppi abeliani sono strutture algebriche *"deboli"*, e ciò spiega perchè ad oggi non sono stati trovati degli algoritmi più efficienti per risolvere in tempo *subesponenziale* il problema. Per questo motivo, a *parità di lunghezza delle chiavi, i protocolli basati sulle curve ellittiche sono molto più sicuri del cifrario RSA*, e quindi per garantire lo stesso livello di sicurezza, RSA richiede chiavi di lunghezza maggiore. Il **NIST** (**N**ational **I**nstitute of **S**tandards and **T**echnology) ha pubblicato in merito a ciò una tabella per il confronto tra i due:

RSA	ECC
1024 bit	160 bit
2048 bit	224 bit
3072 bit	256 bit
7680 bit	384 bit
15360 bit	512 bit

Ciascuna riga della tabella rappresenta il numero di bit che le chiavi devono avere, per ciascun sistema, per essere equivalentemente sicuri, e quindi per richiedere lo stesso costo computazionale per essere forzati.

4.1 Scopo e codice

Lo scopo di questo articolo è dunque quello di presentare due semplici implementazioni in linguaggio **Java** del *cifrario RSA* e del protocollo *ElGamal su ECC*, che operano su interi grandi (*BigIntegers di Java*). Dopodichè i due cifrari verranno analizzati attraverso una serie di test con chiavi di rispettivamente *10,16,20,24 e 25 bit* (quindi estremamente piccole rispetto agli standard), al fine di osservare, al crescere della chiave, l'enorme differenza di sicurezza tra i due. Per valutare la sicurezza, degli stessi messaggi interi m vengono cifrati con entrambe le implementazioni e, mediante degli algoritmi di attacco enumerativi, si tenta di risalire al messaggio originale m misurandone infine i tempi.

4.1.1 RSA

L'implementazione dell'RSA è molto semplice: consiste di una classe *RSA.java* (**6.1**) che ha come variabili d'istanza i parametri standard del cifrario rappresentati mediante il tipo **BigInteger** (**4.1.3**) p, q, n, phi, e, d. Nel metodo costruttore sono inizializzati tutti i parametri, tra cui il parametro e che viene generato casualmente rispettando la condizione $mcd(e, phi) = 1$. Infine vi sono i metodi di cifratura *Encrypt* ($O(1)$), che prende in input msg, e, n e restituisce msg^e mod n, e di decifrazione *Decrypt* ($O(1)$) che prende in input crt, n, d e restituisce crt^d mod n. Nel *SecurityTest.java*(**6.3.2**) vi è inoltre la funzione per eseguire l'attacco enumerativo al cifrario, *RSA_Break* che prende in input il cifrario da rompere e il crittogramma da decifrare e ricava la chiave segreta d (e di conseguenza decifra msg) e ha complessità $O(n)$. Inoltre, in *SecurityTest.java* sono presenti anche due metodi ausiliari, *generaPrimi* e *isPrime*, di complessità rispettivamente $O(n)$ e $O(1)$.

4.1.2 ECC

L'implementazione della ECC consiste invece di una classe *ECPoint.java*(**6.2**) e di una classe *ECC.java*(**6.2.1**). La prima ha come variabili di istanza i parametri di un punto di una curva ellittica prima x, y, a, b, p inizializzate nel costruttore. La seconda ha invece come variabili di istanza i parametri della curva a, b, p e il valore h che serve per convertire un messaggio in un punto della curva. Vi sono alcuni metodi ausiliari come *residuoQuadratico* che restituisce *true* se un dato intero è residuo quadratico ($\Theta(1)$); *koblitz* che trasforma un messaggio in un punto della curva ($O(h)$); *PointToMessage* che estrae un messaggio da una curva ($\Theta(1)$); *pointAdd* che effettua la somma di due punti distini ($\Theta(1)$); *pointSub* che effettua la sottrazione tra due punti distinti ($\Theta(1)$); *pointDouble* che effettua la somma di due punti uguali ($\Theta(1)$); *doubleAndAdd* che calcola il prodotto di uno scalare per un punto della curva mediante il metodo dei raddoppi precedentemente illustrato ($\Theta(b)$ dove b è la lunghezza in bit della rappresentazione binaria di r). Infine abbiamo i metodi di cifratura *ECEncrypt* che prende in input un punto P_m, la chiave pubblica P_d del destinatario e il punto della curva in comune B e restituisce la coppia di punti $\langle V, W\rangle$ ($O(b)$); e di decifrazione *ECDecrypt* che prende in input la coppia $\langle V, W\rangle$ e la chiave privata *prvKey* e restituisce il punto P_m ($O(b)$). Nel *SecurityTest.java* vi è inoltre contenuto il metodo *ECC_Break*, che esegue il calcolo del logaritmo discreto per l'attacco al cifrario; prende in input il cifrario da rompere, il crittogramma da decifrare, il punto comune B e la chiave pubblica P_d, e calcola infine il messaggio originale. La sua complessità è $O(2^b)$ dove b è la dimensione in bit del numero r. Anche questa implementazione lavora utilizzando il tipo **BigInteger** (**4.1.3**)

4.1.3 BigInteger

La classe *BigInteger* è una classe del linguaggio **Java** utilizzata per operazioni matematiche su interi molto grandi, che sono al di fuori della portata di qualsiasi altro tipo primitivo. Si è rilevata particolarmente utile per l'implementazione dei due cifrari soprattutto perchè contiene la definizione di numerose operazioni di algebra modulare nonchè di metodi per calcolare il *mcd*, *inverso in modulo e potenza in modulo.*

5 Conclusioni

Vediamo l'esito di diversi test che a parità di chiavi, che per ciascun test hanno lunghezza diversa, la sicurezza delle curve ellittiche è evidentemente maggiore rispetto a quella dell'RSA, seppur con chiavi estremamente piccole rispetto agli standard.

Cifrario	Cifratura	Decifrazione	**Rottura**
RSA-10 bit	0,0789 secondi	0,00172 secondi	**0,0132 secondi**
ECC-10 bit	0,0414 secondi	0,00501 secondi	**0,0258 secondi**
RSA-16 bit	0,000340 secondi	0,000823 secondi	**0,0345 secondi**
ECC-16 bit	0,00195 secondi	0,00141 secondi	**1,69 secondi**
RSA-20 bit	0,000418 secondi	0,000470 secondi	**0,328 secondi**
ECC-20 bit	0,00638 secondi	0,00122 secondi	**29,5 secondi**
RSA-24 bit	0,000422 secondi	0,000717 secondi	**15,1 secondi**
ECC-24 bit	0,00207 secondi	0,000855 secondi	**78,9 secondi**
RSA-25 bit	0,000295 secondi	0,000969 secondi	**32,3 secondi**
ECC-25 bit	0,00364 secondi	0,000961 secondi	**30,1 minuti**

Dalla tabella sopra riportata, si evince che i tempi di cifratura e decifrazione di entrambi i cifrari sono pressochè uguali e molto brevi; d'altro canto, come ci aspettavamo, i tempi invece di rottura dei cifrari, a parità della dimensione delle chiavi, cambiano notevolmente. Ad esempio su chiavi a 25 bit, il tempo richiesto per la *Fattorizzazione* (in azzurro) è di $32,3$ secondi, mentre il tempo richiesto per la risoluzione del logaritmo discreto sulla curva (in verde) con chiavi a 25 bit è di $30,1$ minuti, quindi di ben $55,8$ volte maggiore. Possiamo inoltre vedere come i tempi richiesti per la fattorizzazione su chiavi a 25 bit siano pressochè uguali ai tempi richiesti per risolvere il logaritmo discreto su curve a 20 bit ($32,3$ secondi contro $29,5$ secondi), e quindi la sicurezza dell' **RSA a** 25 **bit** è circa equivalente alla sicurezza di **ElGamal su ECC a** 20 **bit**. Possiamo dunque concludere che l'esperimento ha avuto successo e ha confermato ciò che afferma il **NIST** sulla sicurezza dei due cifrari asimmetrici esposti, ovvero che le curve ellittiche sono molto più efficienti in termini di sicurezza in quanto occorrono chiavi più grandi all'RSA per eguagliare la sicurezza della ECC. I parametri utilizzati per i test sono consultabili nel sorgente *SecurityTest.java* (**6.3.1**).

6 Appendice

6.1 RSA.java

```java
import java.math.BigInteger;
import java.util.Random;

public class RSA {
    private BigInteger p;
    private BigInteger q;
    private BigInteger n;
    private BigInteger phi;
    private BigInteger e;
    private BigInteger d;

    public RSA(BigInteger P, BigInteger Q){
        p=P;
        q=Q;
        n = p.multiply(q);
        phi = (p.subtract(BigInteger.valueOf(1))).multiply((q.subtract(BigInteger.valueOf(1))));
        Random r = new Random();
        //Genero e a caso coprimo con phi (come limite inferiore uso 3 poich  1 lascerebbe invariato m, 2 facilmente calcolabile.
        BigInteger aux = BigInteger.valueOf(r.nextInt(((phi.subtract(BigInteger.valueOf(1))).intValue() - 3) + 1) + 3);
        while(!(aux.gcd(phi).equals(BigInteger.valueOf(1)))){
            aux = BigInteger.valueOf(r.nextInt(((phi.subtract(BigInteger.valueOf(1))).intValue() - 3) + 1) + 3);
        }
        e = aux;
        d = e.modInverse(phi);
    }
    public BigInteger getE(){ return e; }
    public BigInteger getN(){ return n; }
    public BigInteger getD(){ return d; }
```

6.1.1 Encrypt e Decrypt RSA

```java
    public BigInteger Encrypt(BigInteger msg, BigInteger n,
        BigInteger e){
        return (msg.modPow(e,n));
    }

    public BigInteger Decrypt(BigInteger crt, BigInteger n,
        BigInteger d){
        return (crt.modPow(d,n));
    }
```

6.2 ECPoint.java

```java
import java.math.BigInteger;

public class ECPoint {
    private BigInteger x;
    private BigInteger y;
    private BigInteger p;
    private BigInteger a;
    private BigInteger b;

    public ECPoint(BigInteger A, BigInteger B, BigInteger P,
        BigInteger X, BigInteger Y){
        p = P;
        a = A;
        b = B;
        x = X;
        y = Y;
    }

    public ECPoint(ECC curve, BigInteger X, BigInteger Y){
        p = curve.getP();
        a = curve.getA();
        b = curve.getB();
        x = X;
        y = Y;
    }
    public BigInteger getX() {
        return x;
    }

    public BigInteger getY() {
        return y;
    }
}
```

6.2.1 ECC.java

```java
import Exceptions.NoPointException;
import Exceptions.PointToInfiniteException;
import javafx.util.Pair;

import java.math.BigInteger;
import java.util.Random;

public class ECC {
    //variabili d'istanza (y^2)mod(p) = (x^3 + ax + b)mod(p)
    private BigInteger a;
    private BigInteger b;
    private BigInteger p;
    private BigInteger h;

    public ECC(BigInteger A, BigInteger B, BigInteger P){
        a = A;
        b = B;
        p = P;
    }

    public BigInteger getP(){
        return p;
    }

    public BigInteger getA(){
        return a;
    }

    public BigInteger getB() {
        return b;
    }

    public BigInteger getH(){  return h; }
    public void setH(BigInteger h1){h = h1;}
    //Controlla che y sia residuo quadratico
    private boolean residuoQuadratico(BigInteger z){
        z = z.mod(getP());
        BigInteger exp = getP().subtract(BigInteger.valueOf(1)).divide(BigInteger.valueOf(2));
        // a    residuo quadratico modulo p se a^(p-1)/2 = 1 mod p
        return z.modPow(exp,getP()).compareTo(BigInteger.valueOf(1)) == 0;
    }
```

6.2.2 Koblitz e PointToMessage

```java
    //genera un punto nella curva con alta probabilit (1-(1/2)^h)
    public ECPoint koblitz(BigInteger m) throws NoPointException {
        for(int i=0;i<getH().intValue();i++){
            BigInteger x = (m.multiply(getH()).add(BigInteger.valueOf(i))).mod(getP());
            BigInteger z = (x.multiply(x).multiply(x).add(getA().multiply(x)).add(getB())).mod(getP());
            //   residuo quadratico?
            if(residuoQuadratico(z)){
                return new ECPoint(getA(),getB(),getP(),x,(z.sqrt()).mod(getP()));
            }
        }
        throw new NoPointException("Non e' stato generato un punto nella curva");
    }
    //Trasforma un punto nella curva in un messaggio
    public BigInteger PointToMessage(ECPoint Pm){
        int m = (int)Math.floor((Pm.getX().divide(getH()).intValue()));
        return BigInteger.valueOf(m);
    }
```

6.2.3 Operazioni con punti

```java

    //SOTTRAZIONE TRA DUE PUNTI DISTINTI
    public ECPoint pointSub(ECPoint P, ECPoint Q){

        ECPoint Q1 = new ECPoint(getA(),getB(),getP(),Q.getX(),Q.getY().multiply(BigInteger.valueOf(-1)));
        try {
            return pointAdd(P,Q1);
        } catch (PointToInfiniteException e) {
            e.printStackTrace();
        }
        return null;
    }
```

```
    //SOMMA DUE PUNTI DISTINTI
    public ECPoint pointAdd(ECPoint P, ECPoint Q)  throws
        PointToInfiniteException{
        //escludere xQ = xP
        if(P.getX().equals(Q.getX())) throw new
            PointToInfiniteException("Il punto che e' stato
            generato e' un punto all'infinito; non puo' essere
            usato per cifrare");
        //lambda = (yQ-yP)*(xQ-xP)^-1 mod p
        BigInteger lambda = ((Q.getY().subtract(P.getY())).
            multiply((Q.getX().subtract(P.getX())).modInverse(
            getP()))).mod(getP());
        //xS = lambda^2 - xP - xQ
        BigInteger xS = (lambda.multiply(lambda).subtract(P.
            getX()).subtract(Q.getX())).mod(getP());
        //yS = -yP + lambda(x1 - xS)
        BigInteger yS = (P.getY().multiply(BigInteger.valueOf
            (-1)).add(lambda.multiply(P.getX().subtract(xS)))).
            mod(getP());

        return new ECPoint(getA(),getB(),getP(),xS,yS);
    }

    //SOMMA DI PUNTI UGUALI
    public ECPoint doublePoint(ECPoint P) throws
        PointToInfiniteException {
        //VANNO ESCLUSI I PUNTI CON y = 0 se da raddoppiare
        if(P.getY().equals(BigInteger.valueOf(0))) throw new
            PointToInfiniteException("Il punto che e' stato
            generato e' un punto all'infinito; non puo' essere
            usato per cifrare");
        //lambda = (3xP + a)*(2yP)^-1 mod p
        BigInteger lambda = (BigInteger.valueOf(3).multiply((P.
            getX()).pow(2)).add(getA()).multiply((BigInteger.
            valueOf(2).multiply(P.getY())).modInverse(getP()))).
            mod(getP());
        //xS = lambda^2 - 2*xP
        BigInteger xS = (lambda.pow(2).subtract(BigInteger.
            valueOf(2).multiply(P.getX()))).mod(getP());
        //yS = -yP + lambda(xP-xS)
        BigInteger yS = (BigInteger.valueOf(-1).multiply((P.
            getY().add(lambda.multiply((xS.subtract(P.getX()))))
            ))).mod(getP());

        return new ECPoint(getA(),getB(),getP(),xS,yS);
    }
```

6.2.4 DoubleAndAdd

```java
    public ECPoint doubleAndAdd(BigInteger rIn, ECPoint A) throws PointToInfiniteException {
        BigInteger r = rIn.mod(getP());
        int l = r.bitLength();
        if (l != 0){
            ECPoint R = A;
            for (int i = l-2;i>=0;--i){
                try {
                    R = doublePoint(R);
                } catch (PointToInfiniteException e) {
                    e.printStackTrace();
                }
                if(r.testBit(i)){
                    try {
                        R = pointAdd(R,A);
                    } catch (PointToInfiniteException e) {
                        e.printStackTrace();
                    }
                }
            }
            return R;
        }
        throw new PointToInfiniteException("Il punto che e' stato generato e' un punto all'infinito; non puo' essere usato per cifrare");
    }
```

6.2.5 ECDecrypt

```java
    public ECPoint ECDecrypt(Pair<ECPoint,ECPoint> crt, BigInteger prvKey){
        ECPoint ndV = null;
        try {
            ndV = doubleAndAdd(prvKey,crt.getKey());
        } catch (PointToInfiniteException e) {
            e.printStackTrace();
        }
        ECPoint W = crt.getValue();
        if(ndV == null) throw new NullPointerException();
        return pointSub(W,ndV);
    }
```

6.2.6 ECEncrypt

```
    public Pair<ECPoint,ECPoint> ECEncrypt(ECPoint Pm, ECPoint
        B, ECPoint pubKey){
        Random c = new Random();

        int rIn = c.nextInt((((getP().subtract(BigInteger.
            valueOf(5))).intValue()) - 7) + 1 ) + 7;

        //Devo generare V = rB
        BigInteger r = BigInteger.valueOf(rIn);
        ECPoint V = null;
        try {
            V = doubleAndAdd(r,B);
        } catch (PointToInfiniteException e) {
            e.printStackTrace();
        }
        //GENERO W = Pm + rPd
        ECPoint rPd = null;
        try {
            rPd = doubleAndAdd(r,pubKey);
        } catch (PointToInfiniteException e) {
            e.printStackTrace();
        }
        ECPoint W = null;
        if(rPd == null) throw new NullPointerException();
        try {
            W = pointAdd(Pm,rPd);
        } catch (PointToInfiniteException e) {
            e.printStackTrace();
        }

        return new Pair<>(V,W);
    }
```

6.3 SecurityTest.java

6.3.1 Parametri utilizzati

```
    //Messaggi cifrati per 10,16,20,24,25 bit:
    private static int m10 = 7;
    private static int m16 = 13;
    private static int m20 = 17;
    private static int m24 = 29;
    private static int m25 = 251;
    //RSA:
    //p=37, q=19 10-bit
    private static RSA cfrRSA10 = new RSA(BigInteger.valueOf
        (37),BigInteger.valueOf(19));
```

```
    //p=241 q=157 16-bit
    private static RSA cfrRSA16 = new RSA(BigInteger.valueOf
        (241),BigInteger.valueOf(157));
    //p=397 q=1597 20-bit
    private static RSA cfrRSA20 = new RSA(BigInteger.valueOf
        (397),BigInteger.valueOf(1597));
    //p=3947 q=2557 24-bit
    private static RSA cfrRSA24 = new RSA(BigInteger.valueOf
        (3947),BigInteger.valueOf(2557));
    //p=3001 q=5743 25-bit
    private static RSA cfrRSA25 = new RSA(BigInteger.valueOf
        (3001),BigInteger.valueOf(5743));


    //ECC:
    //10 bit E_823(-1,1)
    private static ECC cfrECC10 = new ECC(BigInteger.valueOf
        (-1),BigInteger.valueOf(1),BigInteger.valueOf(823));
    private static ECPoint B10 = new ECPoint(cfrECC10.getA(),
        cfrECC10.getB(),cfrECC10.getP(),BigInteger.valueOf(19),
        BigInteger.valueOf(293));
    private static ECPoint Pd10;
    private static BigInteger prvKey10 = BigInteger.valueOf
        (899);
    //16 bit E_46133(-1,1)
    private static ECC cfrECC16 = new ECC(BigInteger.valueOf
        (-1),BigInteger.valueOf(1),BigInteger.valueOf(46133));
    private static ECPoint B16 = new ECPoint(cfrECC16.getA(),
        cfrECC16.getB(),cfrECC16.getP(),BigInteger.valueOf(113),
        BigInteger.valueOf(35151));
    private static ECPoint Pd16;
    private static BigInteger prvKey16 = BigInteger.valueOf
        (41899);
    //20 bit E_761291(-1,1)
    private static ECC cfrECC20 = new ECC(BigInteger.valueOf
        (-1),BigInteger.valueOf(1),BigInteger.valueOf(761291));
    private static ECPoint B20 = new ECPoint(cfrECC20.getA(),
        cfrECC20.getB(),cfrECC20.getP(),BigInteger.valueOf(167),
        BigInteger.valueOf(539846));
    private static ECPoint Pd20;
    private static BigInteger prvKey20 = BigInteger.valueOf
        (641899);
    //24 bit E_8812313(-1,1)
    private static ECC cfrECC24 = new ECC(BigInteger.valueOf
        (-1),BigInteger.valueOf(1),BigInteger.valueOf(8812313));
    private static ECPoint B24 = new ECPoint(cfrECC24.getA(),
        cfrECC10.getB(),cfrECC10.getP(),BigInteger.valueOf(773),
        BigInteger.valueOf(3443266));
    private static ECPoint Pd24;
```

```java
    private static BigInteger prvKey24 = BigInteger.valueOf(9341899);
    //25 bit E_28123133(-1,1)
    private static ECC cfrECC25 = new ECC(BigInteger.valueOf(-1),BigInteger.valueOf(1),BigInteger.valueOf(28123133));
    private static ECPoint B25 = new ECPoint(cfrECC25.getA(),cfrECC25.getB(),cfrECC25.getP(),BigInteger.valueOf(2903),BigInteger.valueOf(14901307));
    private static ECPoint Pd25;
    private static BigInteger prvKey25 = BigInteger.valueOf(17341899);
```

6.3.2 RSA_Break

```java
    public static void RSA_Break(RSA cfr, BigInteger crt){
        System.out.println("Provo a fattorizzare n con un attacco enumerativo:\n");
        List<BigInteger> primi = generaPrimi(cfr.getN());
        //Inizio il brute-force
        BigInteger p = BigInteger.valueOf(0),q =  BigInteger.valueOf(0);
        for (BigInteger t:primi) {
            p = cfr.getN().divide(t);
            if((p.multiply(t)).equals(cfr.getN())){
                q = t;
                break;
            }
        }
        System.out.println("La fattorizzazione e' avvenuta con successo e i due numeri primi sono rispettivamente: p = "+p+", q = "+q+".\n\n");
        BigInteger phi = (p.subtract(BigInteger.valueOf(1))).multiply(q.subtract(BigInteger.valueOf(1)));
        BigInteger d = cfr.getE().modInverse(phi);
        System.out.println("Da questi posso dedurre che phi = "+phi+" e dunque posso ricavarmi la chiave privata d = "+d +" dalla quale posso risalire al messaggio m = "+crt.modPow(d,cfr.getN())+"\n\n");
    }
```

6.3.3 ECC_Break

```java
    public static void ECC_Break(ECC cfr,Pair<ECPoint,ECPoint> crt, ECPoint B, ECPoint Pd){
        System.out.println("Provo a calcolare il logaritmo discreto con un attacco enumerativo:\n");
        BigInteger r = BigInteger.valueOf(0);
        for(int k=2;k<cfr.getP().intValue();k++){
            try {
                ECPoint V1 = cfr.doubleAndAdd(BigInteger.valueOf(k),B);
                if(V1.getX().equals(crt.getKey().getX()) && V1.getY().equals(crt.getKey().getY())){
                    r = BigInteger.valueOf(k);
                    break;
                }
            } catch (PointToInfiniteException e) {
                e.printStackTrace();
            }
        }
        //A questo punto ho ricavato r, posso calcolare W = Pm + rPd - rPd, dunque calcolo rPd = r*Pd
        System.out.println("A questo punto sono riuscito a ricavare r = "+r+", dalla quale posso calcolare rPd e dunque calcolare Pm = W - rPd = Pm + rPd - rPd:\n");
        ECPoint Pm = null;
        try {
            ECPoint rPd = cfr.doubleAndAdd(r,Pd);
            Pm = cfr.pointSub(crt.getValue(),rPd);
        } catch (PointToInfiniteException e) {
            e.printStackTrace();
        }
        BigInteger msg = cfr.PointToMessage(Pm);
        System.out.println("Ho trovato il punto Pm = ("+Pm.getX()+","+Pm.getY()+"). Ora provo ad estrarre il messaggio dal punto della curva:");
        System.out.println("Il logaritmo discreto e' stato svolto con successo, il numero r = "+r+" e il messaggio m = "+msg +".");
    }
```

6.3.4 generaPrimi e isPrime

```
    public static List<BigInteger> generaPrimi(BigInteger n) {
        List<BigInteger> primi = new LinkedList<>();
        if(n.intValue() >= 2) {
            primi.add(BigInteger.valueOf(2));
        }
        for(int i = 3; i <= n.intValue(); i += 2) {
            if (isPrime(BigInteger.valueOf(i))) {
                primi.add(BigInteger.valueOf(i));
            }
        }
        return primi;
    }
    private static boolean isPrime(BigInteger number) {
        for (int i = 2; i*i < number.intValue(); i++)
            if (number.intValue() % i == 0)
                return false;
        return true;
    }
```

References

[1] Anna Bernasconi, Paolo Ferragina, Fabrizio Luccio. *Elementi di Crittografia.* Pisa University Press, 2014

[2] *Elliptic curve point multiplication*
https://en.wikipedia.org/wiki/Elliptic_curve_point_multiplication

[3] *Quadratic Residue*
https://en.wikipedia.org/wiki/Quadratic_residue

[4] *A relatively easy to understand primer on elliptic curve cryptography.*
https://arstechnica.com/information-technology/2013/10/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/