Università degli Studi dell'Aquila
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

---

Tesi di Laurea in Ingegneria Informatica e Automatica

Crittografia ellittica e implementazione in Java del protocollo ECDH

Relatore:
Prof. Gabriele Di Stefano

Laureando:
Fabrizio D'Ascenzo

Anno Accademico 2017-2018

```
While(1){
   WorkHard();
}
```

My Code.

Indice

Introduzione

La crittografia ellittica è una tipologia di crittografia a chiave pubblica basata sulle curve ellittiche su campi finiti. L'idea di utilizzare le curve ellittiche in crittografia è nata nel 1985 dalle menti di *Neal Koblitz* e *Victor Miller*. Da quel momento in poi l'interesse verso le curve ellittiche è cresciuto notevolmente e i motivi sono molteplici. Il principale tra questi è che non esiste un algoritmo sub-esponenziale che risolve il problema del logaritmo discreto sulle curve ellittiche. Peraltro, come vedremo nel capitolo 3, tutti i crittosistemi basati sul problema del logaritmo discreto sui campi finiti, hanno un loro analogo in crittografia ellittica.

In questo lavoro di tesi, la trattazione delle curve ellittiche è stata appunto incentrata alle sue applicazioni crittografiche e finalizzata alla realizzazione di un sistema software in grado di eseguire l'*Elliptic Curve Diffie-Hellman Exchange* (ECDH). Quest'ultimo si tratta di un protocollo che ha come obiettivo quello di consentire a due interlocutori di stabilire una chiave segreta condivisa attraverso un canale di comunicazione insicuro.

Struttura della tesi

Nel primo capitolo è presente una introduzione alla crittografia in cui saranno analizzati crittosistemi basati sul problema della fattorizzazione degli interi e su quello del logaritmo discreto. Nel secondo capitolo si sono invece trattate le curve ellittiche dal punto di vista matematico a partire dalle *equazioni di Weierstrass* fino alle curve ellittiche su campi finiti. Nel terzo capitolo sono invece analizzati i crittosistemi basati sulle curve ellittiche analoghi a quelli visti nel primo capitolo e il problema del

logaritmo discreto su curve ellittiche (ECDLP). Infine nel quarto capitolo è trattata la progettazione e l'implementazione del sistema software suddetto.

Lista degli acronimi utilizzati

DH Diffie-Hellman

DHP Diffie-Hellman Problem

DL Discrete Logarithm

DLP Discrete Logarithm Problem

DSA Digital Signature Algorithm

ECC Elliptic Curve Cryptography

ECDH Elliptic Curve Diffie-Hellman

ECDHP Elliptic Curve Diffie-Hellman Problem

ECMQV Elliptic Curve Menezes-Qu-Vanstone

ECDSA Elliptic Curve Digital Signature Algorithm

IFP Integer Factorization Problem

RSA Rivest-Shamir-Adleman

Capitolo 1

Nozioni di base sulla crittografia

In questo capitolo sono riportati alcuni concetti basilari della crittografia. In particolare, nella prima parte vengono descritti gli obiettivi della crittografia stessa, nonché l'importante distinzione tra crittografia simmetrica e asimmetrica. Nella seconda parte saranno invece analizzati specifici crittosistemi, distinti in virtù del particolare problema matematico che vi è alla base.

1.1 Obiettivi della crittografia

Nella figura (1.1), le due entità A e B stanno comunicando attraverso un canale non sicuro. Assumiamo infatti, la presenza di un terzo E, il cui obiettivo può essere quello di leggere i messaggi scambiati tra A e B, modificare parte di essi e/o impersonare una delle due entità comunicanti. Al fine di garantire la sicurezza della comunicazione, possono quindi essere individuati i seguenti obiettivi:

- *Riservatezza*: mantenere segreti i dati per i soggetti non autorizzati;
- *Integrità dei dati*: garantire che i dati non vengano alterati da soggetti non autorizzati;
- *Autenticazione delle entità*: confermare l'identità dei soggetti comunicanti;
- *Non ripudio*: impedire che il mittente possa negare di aver scritto un messaggio.

Mediante la crittografia è possibile raggiungere, con diverse modalità, tutti questi obiettivi.

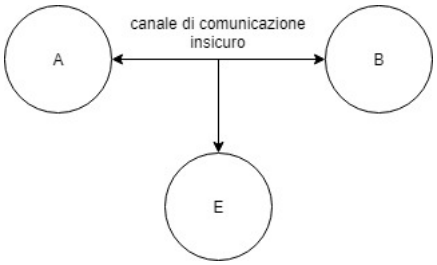

Figura 1.1: Modello base di comunicazione

1.2 Algoritmi di crittografia

Un *algoritmo di crittografia*, consente:

- dato un messaggio m e una chiave di cifratura K_{E}, di cifrare, mediante una apposita funzione matematica E, il messaggio m con la chiave K_{E};

$$E(m, K_{\mathrm{E}}) = c$$

- dato il messaggio cifrato c e una chiave di decifratura K_{D}, di decifrare, mediante una apposita funzione matematica D, il messaggio cifrato c con la chiave K_{D}.

$$D(c, K_{\mathrm{D}}) = m$$

Gli algoritmi di crittografia possono essere classificati in:

- *Simmetrici o "a chiave segreta"*: per le due operazioni di cifratura e decifratura viene utlizzata una stessa chiave K_{S} (quindi vale: $K_{\mathrm{S}} = K_{\mathrm{E}} = K_{\mathrm{D}}$). Il principale vantaggio della crittografia simmetrica è l'elevata efficienza, tuttavia, presenta il problema della distribuzione della chiave, cioè della modalità con la quale le entità comunicanti possono concordare la chiave segreta.

- *Asimmetrici o "a chiave pubblica"*: la nozione di crittografia asimmetrica è stata introdotta nel 1976 da Whitfield Diffie e Martin Hellman nell'articolo *"New Directions in Cryptography"* [1] e pone rimedio al problema della distribuzione della chiave sopracitato. Ogni entità dispone di una coppia di chiavi (K^+,K^-), dove K^+ è detta *chiave pubblica* e K^- è detta *chiave privata*. La chiave pubblica K^+ di una entità, come dice il nome stesso, è pubblicamente nota; la chiave privata K^- è invece tenuta segreta e quindi nota soltanto all'entità stessa. Inoltre, la chiave pubblica e quella privata di una stessa entità sono *complementari*, cioè un messaggio cifrato con la chiave pubblica K^+ di una entità, può essere decifrato soltanto mediante la chiave privata K^- della stessa entità. Quindi, il messaggio m viene cifrato con la chiave pubblica del destinatario e, quest'ultimo poi, potrà decifrare il messaggio cifrato c mediante la propria chiave privata.

$$K^+(m) = c \Rightarrow K^-(c) = m$$

  In generale, gli algoritmi di crittografia asimmetrica risultano essere più lenti rispetto alla controparte simmetrica. Appunto per questo motivo trovano largo utilizzo soluzioni di tipo ibrido, nelle quali la crittografia simmetrica viene utilizzata per cifrare e decifrare i messaggi e quella asimmetrica per lo scambio delle chiavi simmetriche. Ovviamente, il funzionamento della crittografia asimmetrica si basa sul fatto che le chiavi godano della proprietà che sia computazionalmente difficile determinare la chiave privata a partire dalla conoscenza della chiave pubblica. Questa difficoltà è dovuta all'intrattabilità di alcuni problemi matematici, utilizzati per l'implementazione di sistemi crittografici a chiave pubblica. Alcuni di questi problemi matematici sono:

  - *Problema della fattorizzazione degli interi (**IFP**)*: dato un numero intero $n = pq$, con p e q numeri primi molto grandi, trovare p e q. Questo problema è alla base del sistema crittografico *RSA*.
  - *Problema del logaritmo discreto (**DLP**)*: dato un gruppo ciclico G di ordine n. Sia g un generatore di G, quindi ogni elemento $b \in G$ può essere scritto come $b = g^{\mathrm{x}}$, allora chiamiamo x il logaritmo discreto di b

in base g. L'elevata complessità del calcolo del logaritmo discreto è alla base del sistema crittografico *ElGamal* e di firma digitale *DSA*.

– *Problema del logaritmo discreto su curve ellittiche (**ECDLP**)*: data una curva ellittica E su un campo finito F_p. Siano $P, Q \in E$ due punti, tali che $Q = xP$, determinare x. Questo problema, ad esempio, è alla base del sistema crittografico su curve ellittiche analogo ad ElGamal. L'argomento sarà approfondito più avanti.

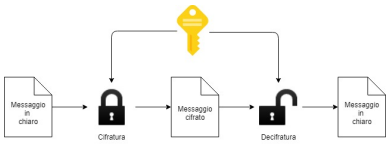

Figura 1.2: Crittografia simmetrica

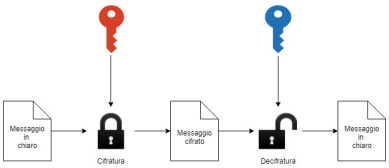

Figura 1.3: Crittografia asimmetrica

1.3 RSA e problema della fattorizzazione intera

Il sistema crittografico RSA, il cui nome deriva dai suoi autori *Rivest*, *Shamir* e *Adleman*, è stato introdotto nel 1977, quindi poco tempo dopo la definizione del concetto di crittografia a chiave pubblica.

Prima di analizzare il sistema RSA diamo alcune definizioni e teoremi che ci saranno utili nella discussione:

Definizione 1.1 (Problema della fattorizzazione degli interi [5])**.** Dato un numero intero positivo N, il problema della fattorizzazione intera consiste nel calcolare la decomposizione in numeri primi $N = \prod p_i^{e_i}$ con $e^i \geq 1$.

Definizione 1.2 (Funzione di Eulero o Toziente)**.** La funzione ϕ di Eulero è definita, per ogni intero positivo n, come il numero di interi compresi tra 1 e n *coprimi* con n (cioè il loro MCD$=1$). Nel caso generale, poiché ϕ è *moltiplicativa* (cioè $\phi(pq) = \phi(p)\phi(q)$), si può dimostrare che:

$$\phi(n) = n[(1 - \frac{1}{p_1})(1 - \frac{1}{p_2})...(1 - \frac{1}{p_r})] = n\prod_{p|n}(1 - \frac{1}{p})$$

dove gli p_{i} sono tutti i primi che compongono la fattorizzazione di n.

Teorema 1.1 (Teorema di Eulero)**.** *Dato un numero intero positivo n sia a coprimo con n allora vale:*

$$a^{\phi(\mathrm{n})} \equiv 1 \; mod \; n.$$

Si tratta di una generalizzazione del cd. "Piccolo Teorema di Fermat".

Nel sistema RSA, la generazione della coppia di chiavi viene svolta attraverso i seguenti passi:

1. vengono scelti casualmente due numeri primi p e q molto grandi.
2. Si calcola $n = pq$,detto *modulo*, e $\phi(n)$. Essendo p e q primi, per la proprietà sopracitata della funzione toziente, vale che $\phi(n) = (p-1)(q-1)$.
3. Si sceglie casualmente un intero e, detto *esponente pubblico* con $1 < e < \phi(n)$ e $\mathrm{MCD}(e, \phi) = 1$, cioè e deve essere coprimo con $\phi(n)$.
4. Si trova un intero d, detto *esponente privato*, tale che $1 < d < \phi(n)$ e $ed \equiv 1 \; mod \; \phi(n)$.
5. La chiave pubblica sarà (n, e) e quella privata (n, d).

È facile dimostrare che, volendo ottenere la chiave privata d a partire dalla conoscenza della chiave pubblica (n, e), si ricadrebbe nel problema della fattorizzazione degli interi. Infatti, per ottenere la chiave privata d si dovrebbe risolvere:

$$e \cdot x = 1 \; mod \; \phi(n)$$

A tal fine si dovrebbe calcolare $\phi(n)$ e quindi sarebbe necessaria la scomposizione in fattori primi di n.

Per quanto riguarda, invece, gli algoritmi di cifratura e decifratura si ha:

- *Cifratura*: $c = m^{\mathrm{e}} \; mod \; n$;
- *Decifratura*: $m = c^{\mathrm{d}} \; mod \; n$.

Consideriamo ora un esempio di comunicazione tra due interlocutori Alice e Bob:

- Sia Alice che Bob determinano la propria chiave pubblicha e privata con il procedimento visto sopra. Si avrà quindi:
  - *Alice*: $K_{\mathrm{A}}^{+} = (n_{\mathrm{A}}, e_{\mathrm{A}})$, $K_{\mathrm{A}}^{-} = (n_{\mathrm{A}}, d_{\mathrm{A}})$;
  - *Bob*: $K_{\mathrm{B}}^{+} = (n_{\mathrm{B}}, e_{\mathrm{B}})$, $K_{\mathrm{B}}^{-} = (n_{\mathrm{B}}, d_{\mathrm{B}})$.
- Alice codifica il messaggio m da inviare a Bob con la chiave pubblica K_{B}^{+} di Bob:
$$c = m^{\mathrm{e_B}} \; mod \; n_{\mathrm{B}}.$$
- Ricevuto il messaggio cifrato c, Bob lo decodifica mediante la sua chiave privata K_{B}^{-}:
$$m = c^{\mathrm{d_B}} \; mod \; n_{\mathrm{B}}.$$

Dimostriamo ora che questo risultato è generalmente valido:

Dimostrazione.

$m = c^{\mathrm{d_B}} \; mod \; n$	effettuiamo la sostituzione $c = m^{\mathrm{e_B}} \; mod \; n$
$m = m^{\mathrm{e_B d_B}} \; mod \; n$	Poiché $e_{\mathrm{B}} d_{\mathrm{B}} \equiv 1 \; mod \; \phi(n)$ si ha che $e_{\mathrm{B}} d_{\mathrm{B}} = k \cdot \phi(n) + 1$
$m = m^{\mathrm{k}\cdot\phi(\mathrm{n})+1} \; mod \; n$	Scrivendo diversamente
$m = m(m^{\phi(\mathrm{n})})^{\mathrm{k}} \; mod \; n$	Per il teorema di Eulero $m^{\phi(\mathrm{n})} \equiv 1 \; mod \; n$
$m = m(1)^{\mathrm{k}} \; mod \; n$	Ottenendo quindi
$m = m \; mod \; n$	Per $m < n$.

□

1.4 Possibili attacchi al sistema RSA

Fin dalla sua pubblicazione, il sistema RSA è stato analizzato alla ricerca di una qualche vulnerabilità. Tuttavia, finora, il risultato è un insieme di possibili attacchi che mostrano soltanto i pericoli di un utilizzo improprio di RSA [6]. Vediamo alcuni di questi attacchi.

1.4.1 Fattorizzazione del modulo n

Gli attacchi che hanno come obiettivo la fattorizzazione del modulo n nei suoi fattori primi, in materia, sono considerati dei *"Brute Force attack"*. Tali attacchi non rappresentano per RSA una seria minaccia, poiché non è stato ancora pubblicato un algoritmo, eseguibile su un computer "classico", che può fattorizzare un intero in tempo polinomiale. Attualmente, infatti, l'algoritmo "classico" più efficiente è il *"General Number Field Sieve"* o *"Crivello dei campi di numeri generale"* che ha una complessità computazionale pari a:

$$O(e^{(c+o(1))(\log n)^{\frac{1}{3}}(\log\log n)^{\frac{2}{3}}})$$

dove c è una costante che dipende dalla variante dell'algoritmo utilizzata [6].
Quindi, per evitare questo tipo di attacchi è sufficiente aumentare la dimensione del modulo n (l'attuale standard è pari a 2048 bit, molto ben oltre il record di fattorizzazione, pari a 768 bit).
È da riportare, comunque, l'esistenza di un algoritmo quantistico, noto come *"Algoritmo di fattorizzazione di Shor"*, in grado di fattorizzare un intero in tempo polinomiale. Per l'implementazione di tale algoritmo, tuttavia, non sono sufficienti dei computer "classici" ma sono richiesti dei computer quantistici, i quali sono ancora soggetti a limitazioni relative ai costi e soprattutto alle tecnologie attuali. È però ipotizzabile che in futuro il calcolo quantistico sarà in grado di violare RSA. In previsione di ciò si è venuto a creare un campo della crittografia detto *"Crittografia post-quantistica"* [9].

1.4.2 Attacco per modulo comune

Come dimostrato in [6], vale il seguente fatto:

> *"Sia (n,e) una chiave pubblica RSA. Data una chiave privata d, è possibile fattorizzare, in modo efficiente, il modulo $n = pq$. Viceversa, data la fattorizzazione del modulo n, è possibile recuperare, in modo efficiente, la chiave privata d."*

Detto ciò, ipotizziamo che in una organizzazione, per semplificare la manutenzione dell'infrastruttura a chiave pubblica, si decida di utilizzare un unico modulo n

e assegnare invece, ad ogni impiegato, un personale esponente pubblico e ed un corrispondente esponente privato d. Ogni impiegato *"i"* avrà quindi:

- *Chiave pubblica:* (n, e_i);
- *Chiave privata:* (n, d_i).

Supponiamo ora che Alice e Bob siano due impiegati della suddetta organizzazione. Per il fatto sopracitato Bob, ad esempio, potrebbe utilizzare la sua chiave privata d_{B} per determinare la fattorizzazione del modulo comune n e quindi, ottenuti p e q, determinare la chiave privata d_{A} di Alice a partire dalla sua chiave pubblica (n, e_{A}).

Mantenendo invariato lo scenario sopra descritto, un'ulteriore ipotesi che possiamo considerare è che l'organizzazione decida di selezionare gli esponenti pubblici da un database di numeri primi, ad esempio per velocizzare il processo di selezione per il singolo impiegato. In questo modo si avrà che, presi due esponenti pubblici e_1 ed e_2, $\mathrm{MCD}(e_1, e_2) = 1$. Supponiamo ora che Alice, Bob e Carl siano tre impiegati dell'organizzazione e che Alice invii lo stesso messaggio m sia a Bob che a Carl. Avremo che:

- $c_{\mathrm{B}} = m^{e_{\mathrm{B}}} \ mod \ n$ è il messaggio codificato inviato a Bob;
- $c_{\mathrm{C}} = m^{e_{\mathrm{C}}} \ mod \ n$ è il messaggio codificato inviato a Carl.

A questo punto un *attacker* avrebbe a disposizione le due chiavi pubbliche di Bob e Carl, rispettivamente (n, e_{B}) e (n, e_{C}), più i due messaggi cifrati c_{B} e c_{C}. Poiché $\mathrm{MCD}(e_{\mathrm{B}}, e_{\mathrm{C}}) = 1$, l'attacker potrà utilizzare l'*algoritmo di Euclide esteso* per calcolare due interi x e y tali che $x \cdot e_{\mathrm{B}} + y \cdot e_{\mathrm{C}} = 1$ e decifrare il messaggio m nel seguente modo:

$c_{\mathrm{B}}^{\mathrm{x}} \cdot c_{\mathrm{C}}^{\mathrm{y}} \ mod \ n$	si effettuano le sostituzioni $c_{\mathrm{B}} = m^{e_{\mathrm{B}}} \ mod \ n$, $c_{\mathrm{C}} = m^{e_{\mathrm{C}}} \ mod \ n$
$m^{\mathrm{x} \cdot e_{\mathrm{B}}} \cdot m^{\mathrm{y} \cdot e_{\mathrm{C}}} \ mod \ n$	scrivendo diversamente
$m^{\mathrm{x} \cdot e_{\mathrm{B}} + \mathrm{y} \cdot e_{\mathrm{C}}} \ mod \ n$	per l'algoritmo di Euclide $x \cdot e_{\mathrm{B}} + y \cdot e_{\mathrm{C}} = 1$
m^1 mod n	l'attacker ha decifrato il messaggio.

Queste due ipotesi dimostrano che, in generale, non si dovrebbe usare uno stesso modulo n per più di una entità.

1.4.3 Attacco ad una trasmissione a molti

Consideriamo il seguente teorema:

Teorema 1.2 (Teorema cinese del resto)**.** *Siano $n_1, n_2, ..., n_k$ con $k > 1$ interi positivi a due a due coprimi, cioè $MCD(n_i, n_j) = 1$ per $i \neq j$, e siano $b_1, b_2, ..., b_k$ interi. Allora il sistema di congruenze lineari*

$$\begin{cases} x \equiv b_1 \ mod \ n_1 \\ x \equiv b_2 \ mod \ n_2 \\ \vdots \\ x \equiv b_k \ mod \ n_k \end{cases} \tag{1.1}$$

è risolubile ed ha un'unica soluzione $mod \ n_1 n_2 \dots n_k$.

Supponiamo ora, che Alice voglia inviare uno stesso messaggio m a più destinatari $d_1, d_2, \dots, d_k$, ciascuno dei quali ha una propria chiave pubblica RSA (n_i, e_i). Assumiamo che $m < n_i \ \forall i \in \{1, \dots, k\}$ e che $\text{MCD}(n_i, n_j) = 1$ per $i \neq j$ (questa condizione è molto probabile che in pratica sia verificata poiché gli n_i sono costruiti con il prodotto di grandi primi scelti in modo casuale [7]). Quindi Alice codifica ciascun messaggio con la rispettiva chiave pubblica (n_i, e_i) del destinatario i-esimo. Per semplicità, assumiamo tutti gli esponenti pubblici $e_i = e = 3$ (anche questa condizione è fondata, poiché , per migliorare l'efficienza della funzione di cifratura, è vantaggioso utilizzare degli esponenti pubblici piccoli come $e = 3$ o $e = 17$) . Un attacker, a questo punto, potrebbe aver intercettato i k messaggi cifrati:

$$c_1 = m^3 \ mod \ n_1, \ c_2 = m^3 \ mod \ n_2, \dots, \ c_k = m^3 \ mod \ n_k$$

se $k \geq e = 3$, allora l'attacker seleziona un sottoinsieme di $e = 3$ elementi e, applicando il *Teorema Cinese del resto* (Th. 1.2) a c_1,c_2 e c_3, può ottenere un'unica soluzione:

$$c \equiv m^3 \ mod \ n_1 n_2 n_3$$

ma poiché $m < n_i \ \forall i \in \{1, \dots, k\}$ allora $m^3 < n_1 n_2 n_3$ quindi $c = m^3$. Per ottenere m, all'attacker sarà quindi sufficiente calcolare la radice cubica intera di c.

Uno stesso messaggio, quindi, non dovrebbe essere inviato ad un numero di destinatari $k \geq e$ aventi lo stesso esponente pubblico. Ovviamente, questo tipo di attacco è efficiente nel caso in cui l'esponente pubblico e sia *"piccolo"*. Una possibile soluzione, per evitare questi tipi di attacco, consiste nell'inserire all'interno del messaggio una stringa di bit puramente casuale e diversa per ogni destinatario. Non darebbe invece alcun beneficio, come sostenuto in [6], l'applicazione di trasformazioni polinomiali a m.

1.4.4 Timing Attack

Questa tipologia di attacchi, a differenza dei precedenti, riguarda l'implementazione del sistema RSA. Infatti, come dimostrato da *Paul C. Kocher* in [8], misurando in modo preciso il tempo impiegato dal sistema RSA per l'operazione di decifratura, un attacker può ottenere velocemente l'esponente privato d.
L'attacco si basa su come, in alcune implementazioni, $m = c^{\mathrm{d}} \ mod \ n$ viene effettivamente calcolato, ovvero mediante il *"Repeated Squaring Algorithm"* che trova il suo fondamento nella seguente uguaglianza:

$$m = c^{\mathrm{d}} = c^{\sum_{\mathrm{i}=0}^{\mathrm{k}} 2^{\mathrm{i}} \mathrm{d_i}} = \prod_{i=0}^{k} c^{2^{\mathrm{i}} \mathrm{d_i}}$$

dove d_{i} è l'i-esima cifra della rappresentazione binaria di $d = d_{\mathrm{k}} d_{\mathrm{k}-1} \ldots d_0$.

Algorithm 1 Repeated Squaring Algorithm

INPUT: c, $d = d_{\mathrm{k}} d_{\mathrm{k}-1} \ldots d_0$
OUTPUT: $m = c^{\mathrm{d}}$

```
z = c
m = 1
for i = 0 to k do
    if d_i = 1 then
        m = m · z mod n
    end if
    z = z^2 mod n
end for
return m
```

Osserviamo che l'operazione $m = m \cdot z \ mod \ n$ viene eseguita soltanto se la i-esima cifra binaria di d è pari a 1. Inoltre, nel caso in cui venga eseguita, il tempo impiegato per la sua esecuzione è proporzionale all'iterazione nella quale si trova l'algoritmo. Ciò permetterebbe ad un attacker, adeguatamente preparato, di determinare d in base al tempo impiegato dal destinatario di un messaggio per decifrarlo.
Il modo più semplice per evitare attacchi di questo tipo consiste nell'introdurre un tempo di ritardo casuale prima di decifrare il messaggio ricevuto, in questo modo il *Repeated Squaring Algorithm* richiederebbe un tempo sempre diverso. Un secondo approccio, più utilizzato, è il *"blinding"*:

$c = m^{\mathrm{e}} \ mod \ n$	è il messaggio cifrato ricevuto dal destinatario
$c' = c \cdot r^{\mathrm{e}} \ mod \ n$	prima di decifrare c si applica e ad un numero random $r \in \mathbb{Z}_{\mathrm{n}}^*$
$m' = (c')^{\mathrm{d}} \ mod \ n$	si decifra c' applicando d
$m = m' \cdot r^{-1} \ mod \ n$	si ottiene m dividendo per r.

In questo modo viene decifrato un messaggio c' non conosciuto dall'attacker che quindi non può procedere con l'attacco.

1.5 Crittosistemi basati sul problema del logaritmo discreto

Il primo sistema basato sul problema del logaritmo discreto è stato il protocollo per lo scambio di chiavi di *Diffie-Hellman* del 1976. Da quel momento diverse varianti di questo schema sono state mano a mano proposte.
Prima di analizzare i singoli sistemi di crittografia, diamo alcune definizioni:

Definizione 1.3 (Logaritmo Discreto)**.** Sia G un gruppo ciclico finito di ordine n. Sia g un generatore di G e $\alpha \in G$. Allora il *Logaritmo Discreto* di α in base g, indicato con $\log_{\mathrm{g}} \alpha$, è l'unico intero x, $0 \leq x \leq n-1$, tale che $g^{\mathrm{x}} = \alpha$.

Gruppi di particolare interesse in crittografia per il Problema del Logaritmo Discreto sono i gruppi moltiplicativi $\mathbb{Z}_{\mathrm{p}}^*$ degli interi modulo p, con p primo. Quindi:

Definizione 1.4 (Problema del Logaritmo Discreto)**.** Dato un numero primo p, un generatore g di $\mathbb{Z}_{\mathrm{p}}^*$ determinare l'intero x, $0 \leq x \leq p-1$, tale che $g^{\mathrm{x}} \equiv \alpha \ mod \ p$.

L'utilizzo del problema del logaritmo discreto in crittografia è dovuto al fatto che non esistono algoritmi efficienti per calcolarlo. Al contrario, l'operazione inversa di *esponenziazione discreta* è "facilmente" calcolabile (appunto per questo si parla di funzione *one-way*).

1.5.1 Protocollo di Diffie-Hellman

Il protocollo è stato proposto nel 1976 da *Whitfield Diffie* e *Martin E. Hellman* in [1]. Tuttavia, come ammesso dallo stesso Hellman in [10], per la definizione del protocollo furono fondamentali il suggerimento di *John Gill* di utilizzare il problema del logaritmo discreto per creare funzioni *one-way* e il lavoro di *Ralph Merkle* sulla distribuzione delle chiavi pubbliche.

Il protocollo di Diffie-Hellman consente a due entità di stabilire una chiave segreta condivisa tramite un canale di comunicazione non sicuro e senza la necessità che le due parti si siano precedentemente incontrate o scambiate informazioni. La chiave così determinata potrà poi essere utilizzata mediante un sistema di crittografia simmetrica.

Vediamone nel dettaglio il funzionamento:

1. Alice e Bob scelgono un numero primo p grande (tipicamente al minimo 512 cifre binarie) e una base α (che deve essere un generatore modulo p). Questi dati possono essere resi pubblici.

2. Alice e Bob scelgono rispettivamente un x e un $y \in \{1, \ldots, p-1\}$ segreti.

3. Alice invia a Bob $A = \alpha^{\mathrm{x}} \ mod \ p$, Bob invia ad Alice $B = \alpha^{\mathrm{y}} \ mod \ p$ (questo scambio avviene su un canale insicuro).

4. Alice e Bob calcolano la chiave segreta:

   - Alice calcola: $S_{\mathrm{A}} = B^{\mathrm{x}} \ mod \ p = \alpha^{\mathrm{y}\cdot\mathrm{x}} \ mod \ p$;
   - Bob calcola: $S_{\mathrm{B}} = A^{\mathrm{y}} \ mod \ p = \alpha^{\mathrm{x}\cdot\mathrm{y}} \ mod \ p$.

   $\Rightarrow S_{\mathrm{A}} = S_{\mathrm{B}} = S$.

È da notare che le informazioni rese pubbliche sono quindi: p, α, A e B. Il problema di determinare S essendo note tali informazioni è detto *"Problema di Diffie-Hellman"* (DHP). È possibile dimostrare [11] che:

$$DHP \leq DLP$$

cioè se si è in grado di risolvere il problema del logaritmo discreto allora si è grado di risolvere anche il problema di Diffie-Hellman (ad esempio si potrebbe ottenere x da A per poi calcolare $S = B^{\mathrm{x}} \; mod \; p$). Il viceversa è invece uno dei problemi aperti della crittografia, cioè non è noto se sapendo risolvere il problema di Diffie-Hellman si sia anche in grado di risolvere il problema del logaritmo discreto.

1.5.2 ElGamal

Nel 1985 *Taher ElGamal* propose un sistema di crittografia che semplificava quello proposto dal duo Diffie-Hellman. Infatti il sistema originale richiedeva l'interazione di entrambe le parti coinvolte per il calcolo di una chiave privata comune. Questo poneva dei problemi se il sistema era utilizzato in contesti nei quali entrambe le parti non sono in grado di interagire in tempi ragionevoli a causa di ritardi nella trasmissione o indisponibilità di una di esse. ElGamal ha quindi introdotto un esponente casuale k che va a sostituire l'esponente privato dell'entità ricevente [12]. Grazie a tale modifica il sistema può essere utilizzato da una delle entità comunicanti nonostante l'inattività della controparte.

Vediamone nel dettaglio il funzionamento:

1. Alice sceglie un numero primo p di dimensione pari a L bit e un generatore g di $\mathbb{Z}_{\mathrm{p}}^*$ di ordine q (cioè $g^{\mathrm{q}} = 1$), dove q è un divisore primo di $p-1$ di dimensione pari a N bit (i valori di L e N sono stabiliti da enti governativi).

2. Alice sceglie un valore $x \in \{1, \ldots, q-1\}$. La chiave privata di Alice sarà quindi $K_{\mathrm{A}}^- = x$.

3. Alice calcola $A = g^{\mathrm{K_A^-}} \; mod \; p$. La sua chiave pubblica sarà $K_{\mathrm{A}}^+ = (p, q, g, A)$.

4. Bob, che desidera inviare un messaggio m ad Alice, recupera la sua chiave pubblica K_{A}^+ e sceglie un valore $k \in \{1, \ldots, q-1\}$.

5. Bob calcola:

   - $c_1 = g^{\mathrm{k}} \; mod \; p$;
   - $c_2 = m \cdot A^{\mathrm{k}} \; mod \; p$;

   ed invia ad Alice (c_1, c_2).

6. Alice calcola:

   - $y = c_1^{\mathrm{K_A^-}} \; mod \; p$ e il suo inverso y^{-1};
   - $m' = y^{-1} \cdot c_2 \; mod \; p$.

   $\Rightarrow m' = m$.

Dimostriamo che il messaggio ottenuto da Alice sia proprio il messaggio m cifrato da Bob:

Dimostrazione.

$y = c_1^{\mathrm{K_A^-}} \; mod \; p$ effetuiamo la sostituzione $c_1 = g^{\mathrm{k}} \; mod \; p$

$y = (g^{\mathrm{k}})^{\mathrm{K_A^-}} \; mod \; p$ scriviamo diversamente

$y = (g^{\mathrm{K_A^-}})^{\mathrm{k}} \; mod \; p$ ma $g^{\mathrm{K_A^-}} \; mod \; p = A$

$y = A^{\mathrm{k}} \; mod \; p$ calcoliamo y^{-1} e facciamo il prodotto con $c_2 = m \cdot A^{\mathrm{k}} \; mod \; p$

$m' = m$.

□

È da notare che per ottenere il messaggio in chiaro un attacker dovrebbe essere in grado di calcolare $A^{\mathrm{k}} \; mod \; p$ a partire dai dati noti: c_1, $K_{\mathrm{A}}^{+} = (p, q, g, A) \Rightarrow$ *Problema di Diffie-Hellman.*

1.5.3 Firma Digitale e DSA

Nel paragrafo 1.1 abbiamo indicato tra gli obiettivi da perseguire al fine di rendere una comunicazione sicura quelli di: *autenticazione*, *integrità* e *non ripudiabilità*. Tali obiettivi possono essere raggiunti mediante il meccanismo della *Firma digitale*

a crittografia asimmetrica che, ad esempio, nell'ordinamento giuridico italiano è equiparata a tutti gli effetti di legge alla firma autografa su carta. La titolarità della firma digitale è garantita dai *"certificatori"* o (*Certification Authority*), cioè soggetti con particolari requisiti di onorabilità (solitamente sono grandi società, ad esempio Poste Italiane) [13]. In particolare i certificatori hanno il compito di tenere i registri delle chiavi pubbliche. L'idea alla base della firma digitale infatti è quella di utilizzare la chiave privata per generare la firma stessa e la chiave pubblica per verificarne l'autenticità.

La nozione di schema di firma digitale è stata descritta anch'essa, nel 1976, da *W. Diffie* e *M.E. Hellman*, sebbene si limitarono a ipotizzare soltanto l'esistenza di tali schemi. Nel 1977 con la pubblicazione di RSA, si è avuto un primo esempio concreto poiché l'algoritmo può essere utilizzato per ottenere delle firme digitali [14]. Nel 1991 fu presentato dal *National Institute of Standards and Technology* (NIST) il *Digital Signature Algorithm* (DSA). Il sistema, inventato da *David Kravitz*, è basato sullo schema di ElGamal e deve quindi la sua sicurezza alla difficoltà di calcolare i logaritmi discreti.

Prima di vedere nel dettaglio il funzionamento dello schema DSA, diamo la seguente definizione:

Definizione 1.5 (Funzione Hash)**.** Una *funzione hash* h è una funzione che prende in input un messaggio di lunghezza variabile e come output restituisce una stringa di lunghezza fissa detta *message digest*. Esempi di funzioni hash sono *MD5* e *SHA*.

Una nota problematica, seppure estremamente rara, delle funzioni hash è quella delle *collisioni*. Si ha una collisione hash quando due diversi input producono uno stesso output. Una funzione che prende un input a lunghezza variabile e restituisce un output a lunghezza fissa, deve avere necessariamente delle collisioni, poiché il numero degli output possibili è finito a fronte di un numero infinito di possibili input.

Tornando al DSA, così come tutti gli schemi di firma digitale, si compone di due algoritmi:

- un *algoritmo di generazione della firma*;

- un *algoritmo di verifica della firma.*

Supponiamo che Alice voglia firmare un messaggio m da inviare a Bob. Siano $K_{\mathrm{A}}^{+} = (p, q, g, A)$ e $K_{\mathrm{A}}^{-} = x$, rispettivamente la chiave pubblica e la chiave privata di Alice, allora la generazione della firma avviene nel seguente modo:

1. viene scelto un valore $k \in [1, \ldots, q-1]$.
2. si calcola $r = (g^{\mathrm{k}} \; mod \; p) \; mod \; q$. Se $r = 0$ si torna al passo 1.
3. si calcola $h = H(m)$ dove H è una funzione hash e h il *message digest.*
4. si calcola $s = k^{-1}(h + xr) \; mod \; q$. Se $s = 0$ si torna al passo 1.

   $\Rightarrow$ la firma è la coppia (r, s).

Bob ricevuti il messaggio m e la firma (r, s) può verificarla, attraverso i parametri pubblici $K_{\mathrm{A}}^{+} = (p, q, g, A)$ e H, nel seguente modo:

1. si verifica che r ed s appartengano all'intervallo $[1, \ldots, q-1]$. Se tale condizione non è vera, la verifica termina senza successo.
2. si calcola $h = H(m)$.
3. si calcola $\omega = s^{-1} \; mod \; q$.
4. si calcola $u_1 = h \cdot \omega \; mod \; q$ e $u_2 = r \cdot \omega \; mod \; q$.
5. si calcola $v = (g^{\mathrm{u_1}} \cdot A^{\mathrm{u_2}} \; mod \; p) \; mod \; q$.
6. se $v = r$ la verifica termina con successo, se $v \neq r$ viceversa.

Dimostriamo che, se la firma è autentica ed è stata correttamente ricevuta insieme al messaggio e ai parametri pubblici, $v = r$.

Dimostrazione.

$v = (g^{\mathrm{u_1}} \cdot A^{\mathrm{u_2}} \ mod\ p) \ mod\ q$	effettuiamo la sostituzione $A = g^{\mathrm{x}} \ mod\ p$
$v = (g^{\mathrm{u_1}} \cdot g^{\mathrm{x \cdot u_2}} \ mod\ p) \ mod\ q$	sostituiamo $u_1 = \cdot\omega \ mod\ q$, $u_2 = r \cdot \omega \ mod\ q$
$v = (g^{\mathrm{h} \cdot \omega} g^{\mathrm{x \cdot r} \cdot \omega} \ mod\ p) \ mod\ q$	scriviamo diversamente
$v = (g^{\omega \cdot (\mathrm{h+x \cdot r})} \ mod\ p) \ mod\ q$	ma $(h + x \cdot r) = s \cdot k \ mod\ q$
$v = (g^{\omega \cdot \mathrm{s \cdot k}} \ mod\ p) \ mod\ q$	sostituiamo $\omega = s^{-1} \ mod\ q$
$v = (g^{\not{s}^{-1} \cdot \not{s} \mathrm{k}} \ mod\ p) \ mod\ q$	dopo le cancellazioni ciò che rimane è proprio r.

□

Capitolo 2

Curve ellittiche

In questo capitolo è fornita una descrizione matematica delle curve ellittiche, nonché della loro aritmetica (*Legge di Gruppo*, *Point Multiplication*). Nella parte finale del capitolo sono trattate le curve ellittiche su campi finiti che sono alla base dei sistemi crittografici che verrano analizzati nel capitolo successivo. Per le nozioni di algebra che verranno utilizzate nel corso della trattazione delle curve ellittiche, si faccia riferimento all'*Appendice A*.

2.1 Definizione ed equazioni di Weierstrass

Definizione 2.1 (Curva Ellittica [15])**.** Una curva ellittica E su un campo K è una curva cubica di grado 3 e genere 1 con un unico punto sulla *linea all'infinito*, cioè il *punto all'infinito* $\mathcal{O} = [0 : 1 : 0]$ e definita nel piano proiettivo $\mathbb{P}^2$ dall'*equazione di Weierstrass*:

$$E : Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3 \tag{2.1}$$

con a_1,a_2,a_3,a_4 e $a_6 \in \bar{K}$.

Dimostriamo che ogni curva ellittica ha uno e un solo punto all'infinito.

Dimostrazione. Imponiamo nella (2.1) $Z = 0$. Si ottiene $X^3 = 0 \rightarrow X = 0$. Calcoliamo la Y imponendo, sempre nella (2.1) $X = 0$, ottenendo $Y^2Z + a_3YZ^2 = a_6Z^3$. Rimanendo $Z = 0$ è chiaro che la Y può assumere qualsiasi valore ma, in uno spazio proiettivo tutte le terne $[0 : \alpha Y : 0]$, dove α è una costante, sono equivalenti. □

Per facilitare la notazione, generalmente si usa l'equazione di Weierstrass usando le coordinante non omogenee $x = X/Z$ e $y = Y/Z$:

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{2.2}$$

con a_1,a_2,a_3,a_4 e $a_6 \in K$.
Peraltro affinché la curva sia *"liscia"* o *"non singolare"* deve valere $\Delta \neq 0$, dove Δ è il *"discriminante"* di E ed è definito come:

$$\begin{cases} \Delta = -d_2^2d_8 - 8d_4^3 - 27d_6^2 + 9d_2d_4d_6 \\ d_2 = a_1^2 + 4a_2 \\ d_4 = 2a_4 + a_1a_3 \\ d_6 = a_3^2 + 4a_6 \\ d_8 = a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2 \end{cases} \tag{2.3}$$

Quindi l'insieme dei punti *K-razionali* della curva E sarà dato da:

$$E(K) = \{(x, y) \in K \times K \mid y^2 + a_1xy + a_3y - x^3 - a_2x^2 - a_4x - a_6 = 0, \Delta \neq 0\} \cup \{\mathcal{O}\}$$

Se K è un campo finito, allora la curva E ha un numero finito di punti denotato con $\#E(K)$.

Definizione 2.2 (Curve ellittiche isomorfe)**.** Date due curve E_1 e E_2 definite su un campo K:

$$E_1 : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

$$E_2 : y^2 + \bar{a_1}xy + \bar{a_3}y = x^3 + \bar{a_2}x^2 + \bar{a_4}x + \bar{a_6}$$

sono dette *isomorfe* sul campo K se esistono u, r, s, $t \in K$, $u \neq 0$, tale che il cambio di variabili

$$(x, y) \to (u^2x + r, u^3y + u^2sx + t) \tag{2.4}$$

transformi l'equazione E_1 nell'equazione E_2.

Detto ciò, è possibile semplificare l'equazione (2.2) applicando un cambio di variabili. In particolare se $char(K) \neq 2, 3$ il cambio di variabili

$$(x, y) \to (\frac{x - 3a_1^2 - 12a_2}{36}, \frac{y - 3a_1x}{216} - \frac{a_1^3 + 4a_1a_2 - 12a_3}{24})$$

trasforma (2.2) nell'equazione della curva isomorfa

$$y^2 = x^3 + ax + b \tag{2.5}$$

con a, $b \in K$. Il discriminante di (2.5) è $\Delta = -16(4a^3 + 27b^2)$ quindi, affinché la curva sia non singolare, è necessario che $4a^3 + 27b^2 \neq 0$. Otteniamo di conseguenza:

$$E(K) = \{(x, y) \in K \times K \mid y^2 = x^3 + ax + b,\ 4a^3 + 27b^2 \neq 0\} \cup \{\mathcal{O}\}. \tag{2.6}$$

Esempio 2.1 (Curve ellittiche su $\mathbb{R}$)**.** Consideriamo le due curve ellittiche

$$E_1 : y^2 = x^3 - x$$

$$E_2 : y^2 = x^3 + \frac{1}{4}x + \frac{5}{4}$$

definite sul campo dei numeri reali $\mathbb{R}$.

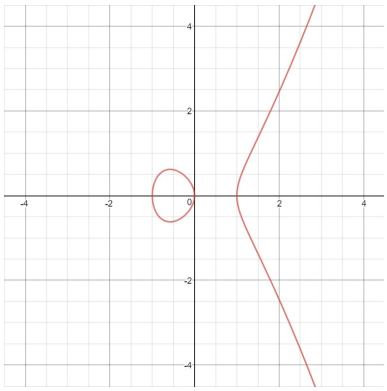

Figura 2.1: $y^2 = x^3 - x$

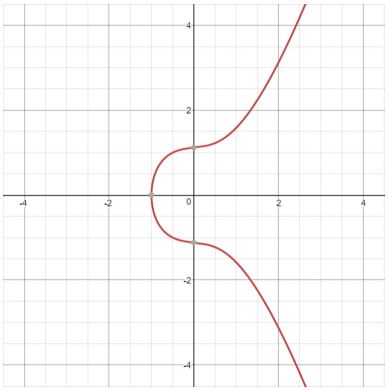

Figura 2.2: $y^2 = x^3 + \frac{1}{4}x + \frac{5}{4}$

2.2 Legge di Gruppo

Sia E una curva ellittica su un campo K e sia $E(K)$ l'insieme dei punti della curva. Allora è possibile definire un'operazione di addizione (*Point Addition*) "+" di due punti $P, Q \in E(K)$ che restituisce un terzo punto $R \in E(K)$. La regola di addizione può essere esposta in maniera intuitiva dal punto di vista geometrico. Siano quindi $P = (x_1, y_1)$ e $Q = (x_2, y_2)$ due punti distinti di una curva ellittica E, si traccia la retta L passante per P e Q, questa retta intercetta un terzo punto R sulla curva

E. Sia L' la retta passante per R e $\mathcal{O}$. Allora L' attraversa la curva E in R, $\mathcal{O}$ e nel simmetrico di R rispetto all'asse x, $-R = P + Q = (x_3, y_3)$ [16]. Nel caso in cui, invece, $P = Q$ (*Point Doubling*) si disegna prima la retta L tangente alla curva E nel punto P. Questa retta attraversa la curva ellittica in un secondo punto R. Sia L' la retta passante per R e $\mathcal{O}$. Allora L' attraversa la curva E in R, $\mathcal{O}$ e nel simmetrico di R rispetto all'asse x, $-R = P + Q = P + P$.

Esempio 2.2. Data la curva $E : y^2 = x^3 - 7x + 10$ su $\mathbb{R}$ consideriamo i due seguenti casi:

1. Siano $P = (1, 2)$ e $Q = (3, 4)$ allora la somma $P + Q = -R = (-3, 2)$, come si vede in figura (2.3).

2. Sia $P = (1, 2)$ allora la somma $P + P = -R = (-1, -4)$, come si vede in figura (2.4).

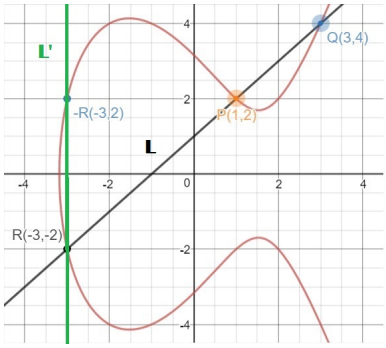

Figura 2.3: Point Addition

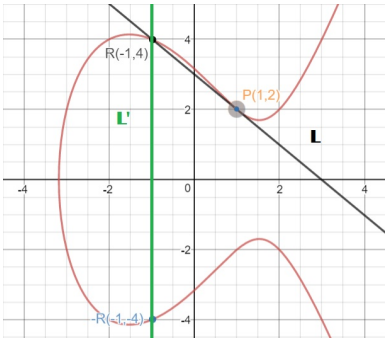

Figura 2.4: Point Doubling

L'operazione di addizione appena descritta gode peraltro delle seguenti proprietà:

1. Se una retta L interseca una curva ellittica E in tre punti (non necessariamente distinti) P, Q, R allora $P + Q + R = \mathcal{O}$;

2. $P + \mathcal{O} = P \ \ \forall P \in E$ (*esistenza dell'elemento identità*);

3. $P + Q = Q + P \ \ \forall P,\ Q \in E$ (*proprietà commutativa*);

4. Sia $P \in E$, allora esiste un punto in E, denotato con $-P$ tale che $P + (-P) = \mathcal{O}$ (*esistenza dell'elemento inverso*);

5. Siano $P,\ Q,\ R \in E$, allora vale $(P+Q)+R = P+(Q+R)$ (*proprietà associativa)*).

Dimostrazione.

1. La prima proprietà risulta dalla stessa definizione dell'operazione di addizione. Scrivendo infatti $(P+Q)+R$, essendo $P+Q=-R$, si ha $-R+R$. Ma $-R$ è il simmetrico rispetto all'asse x di R quindi la retta L passante per questi due punti è parallela all'asse y e il terzo punto della curva ellittica che interseca è il punto all'infinito $\mathcal{O}$. Perciò $-R+R=\mathcal{O}=P+Q+R$ (ciò si può capire anche guardando la figura (2.3)).

2. Se $Q=\mathcal{O}$ la retta L e la retta L' vengono a coincidere. La prima interseca la curva in P, $\mathcal{O}$ e R; la seconda in R, $\mathcal{O}$ e $P+\mathcal{O}$. Quindi possiamo scrivere $P+\mathcal{O}=P$.

3. Anche la terza proprietà risulta dalla definizione dell'operazione di addizione: l'ordine con il quale prendiamo i punti non è rilevante, la retta che li attraversa è sempre la stessa.

4. La quarta proprietà può essere provata attraverso la prima e la seconda. Consideriamo, infatti, una retta L che interseca la curva ellittica E in tre punti P, Q e R. Allora per la prima proprietà $P+Q+R=\mathcal{O}$. Poniamo ora $Q=\mathcal{O}$. Avremo $(P+\mathcal{O})+R=\mathcal{O}$. Ma per la seconda proprietà $P+\mathcal{O}=P$, quindi risulta $P+R=\mathcal{O}$ con $R=-P$, cioè il simmetrico di P rispetto all'asse x.

5. La quinta proprietà può essere provata facendo ricorso alla seguente proposizione:

   Proposizione 2.1. Sia C una cubica irriducibile e C',C" due cubiche. Supponiamo che C' intersechi C in P_i per $i=1,\dots,9$ e C" intersechi C in Q_i per $i=1,\dots,8$ e in H. Se $P_\text{i}=Q_\text{i}\ \forall i\in\{1,\dots,8\}$ allora $H=P_9$ (per la dimostrazione si veda [17]).

   Data una curva ellittica E e tre punti P, Q e $R\in E$. Consideriamo tre rette:

   - L_1 che interseca la curva ellittica E in P, Q e $-(P+Q)$;

- M_1 che interseca la curva ellittica E in $\mathcal{O}$, $-(P+Q)$ e $P+Q$;
- L_2 che interseca la curva ellittica E in $P+Q$, R e $-((P+Q)+R)$

e altre tre rette:

- M_2 che interseca la curva ellittica E in Q, R e $-(Q+R)$;
- L_3 che interseca la curva ellittica E in $\mathcal{O}$, $-(Q+R)$ e $Q+R$;
- M_3 che interseca la curva ellittica E in P, $Q+R$ e $-(P+(Q+R))$.

Applichiamo ora la proposizione (2.1) a $C = E$, $C' = L_1L_2L_3$ e C" $= M_1M_2M_3$. Si noti che otto dei punti di intersezione delle due cubiche C' e C" con la curva E sono uguali, quindi, per quanto affermato dalla proposizione (2.1) si ottiene:

$$-((P+Q)+R) = -(P+(Q+R))$$

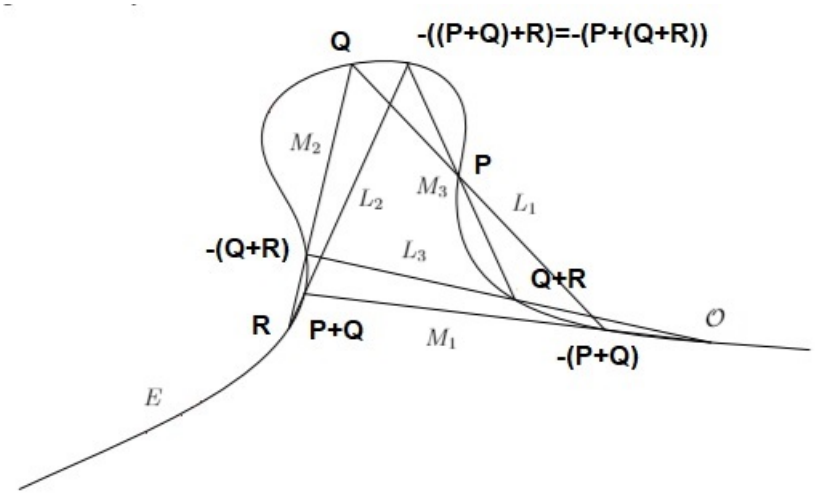

Figura 2.5: Dimostrazione associatività

□

A questo punto, essendo rispettati gli assiomi di gruppo e la proprietà commutativa, è facile constatare che $(E(K), +)$ è un *gruppo Abeliano*.

Trasformiamo adesso la regola di addizione, descritta finora dal punto di vista geometrico, in un set di equazioni per ottenere direttamente il risultato della somma dati due punti P e Q.

Consideriamo una curva $E : y^2 = x^3 + ax + b$. Si possono distinguere due casi:

1. $P \neq Q$. Siano $P = (x_1, y_1)$, $Q = (x_2, y_2)$ due punti della curva E e sia $L : y = mx + q$ la retta passante per P e Q. Si possono distinguere due sottocasi:

   (a) $x_1 \neq x_2$. La retta L ha equazione

   $$y = m(x - x_1) + y_1 \;\; con \;\; m = \frac{(y_2 - y_1)}{(x_2 - x_1)}$$

   e l'intersezione con la curva E restituisce il polinomio di terzo grado

   $$x^3 - m^2x^2 + (2m^2x_1^2 - 2m^2y_1 + a)x + b - (mx_1 - y_1)^2 = 0$$

   del quale conosciamo già le due radici x_1 e x_2. Quindi ponendo

   $$x^3 - m^2x^2 + (2m^2x_1^2 - 2m^2y_1 + a)x + b - (mx_1 - y_1)^2 = (x - x_1)(x - x_2)(x - x_3)$$

   si ottiene il sistema

   $$\begin{cases} x^3 = x^3 \\ m^2x^2 = (x_1 + x_2 + x_3)x^2 \\ (2m^2x_1^2 - 2m^2y_1 + a)x = (x_1x_2 + x_2x_3 + x_1x_3)x \\ b - (mx_1 - y_1)^2 = -x_1x_2x_3 \end{cases} \tag{2.7}$$

   che risolto ci restituisce la coordinata x del terzo punto d'intersezione R della retta L con la curva E e dall'equazione della retta possiamo trovare la coordinata y. Quindi

   $$\begin{cases} x_3 = m^2 - x_1 - x_2 \\ y_3 = m(x_3 - x_1) + y_1 \end{cases} \tag{2.8}$$

   Il punto $P + Q = -R$, cioè il simmetrico rispetto all'asse x di R, lo otteremo cambiando il segno di y_3. Quindi

   $$\begin{cases} x_3 = m^2 - x_1 - x_2 \\ -y_3 = m(x_1 - x_3) - y_1 \end{cases} \tag{2.9}$$

Riassumendo, dati due punti distinti $P = (x_1, y_1)$ e $Q = (x_2, y_2)$ con $x_1 \neq x_2$, allora il punto $P + Q = -R$ è quello avente le coordinate ottenute risolvendo il seguente sistema:

$$\begin{cases} m = \frac{(y_2 - y_1)}{(x_2 - x_1)} \\ x_3 = m^2 - x_1 - x_2 \\ -y_3 = m(x_1 - x_3) - y_1 \end{cases} \tag{2.10}$$

(b) $x_1 = x_2$ e quindi $y_1 = -y_2$. In tal caso $Q = -P$, la retta L interseca P,Q e $\mathcal{O}$. Pertanto $P + Q = \mathcal{O}$.

2. $P = Q$ (*Poin Doubling*). In tal caso L è la retta tangente la curva nel punto $P = Q$. Si possono distinguere due sottocasi:

   (a) $y_1 = y_2 = 0$. La tangente L è verticale e attraversa la curva in $P = Q$ e $\mathcal{O}$ quindi vale $P + Q = P + P = \mathcal{O}$.

   (b) $y_1 \neq 0$. L'equazione della tangente L è

   $$y = m(x - x_1) + y_1 \;\; con \;\; m = \frac{\partial E/\partial x}{\partial E/\partial y} = \frac{\partial y}{\partial x} = \frac{3x_1^2 + a}{2y_1}$$

   come fatto precedentemente, dall'intersezione della retta L con la curva E si ottiene un polinomio di terzo grado del quale si conosce la sola radice x_1 che però è doppia

   $$x^3 - m^2x^2 + (2m^2x_1^2 - 2m^2y_1 + a)x + b - (mx_1 - y_1)^2 = (x - x_1)^2(x - x_3)$$

   e si ottiene il sistema

   $$\begin{cases} x^3 = x^3 \\ m^2x^2 = (2x_1 + x_3)x^2 \\ (2m^2x_1^2 - 2m_2y_1 + a)x = x_1(x_1 - 2x_3)x \\ b - (mx_1 - y_1)^2 = -x_1^2x_3 \end{cases} \tag{2.11}$$

   Con la x_3 ricavata dal sistema e l'equazione della retta L troviamo le coordinate del punto R e invertendo il segno di y_3, come prima, troviamo le coordinate del punto $-R = P + Q = P + P$.

Riassumendo, dato un punto $P = (x_1, y_1)$ con $y_1 \neq 0$, allora il punto $P + P = -R$ è quello avente le coordinate ottenute risolvendo il seguente sistema:

$$\begin{cases} m = \frac{3x_1^2 + a}{2y_1} \\ x_3 = m^2 - 2x_1 \\ -y_3 = m(x_1 - x_3) - y_1 \end{cases} \tag{2.12}$$

È da notare che mediante queste formule è possibile calcolare anche la differenza tra due punti. In particolare se $P + Q = R$, allora possiamo indicare la differenza con $Q = R - P$ dove $-P$, come già detto in precedenza, è il simmetrico rispetto all'asse x di P. Quindi la differenza può essere considerata come un caso particolare della *Point Addition* in cui $Q = R - P = R + (-P)$.

2.3 Moltiplicazione scalare

Dato un punto P di una curva ellittica e un intero k allora:

- $kP = P + P + \ldots + P$ se $k > 0$;
- $kP = (-P) + (-P) + \ldots + (-P)$ se $k < 0$;
- $kP = \mathcal{O}$ se $k = 0$.

Tale operazione prende il nome di *"moltiplicazione scalare"* o *"Point Multiplication"*. Calcolare kP attraverso $k - 1$ Point addiction è ovviamente inefficiente, in particolare quando $|k|$ è molto grande. Peraltro, nella crittografia ellittica la Point Multiplication è un'operazione largamente utilizzata. Appunto per tali motivi si utilizzano degli algoritmi più efficienti come il *Double & Add*, il *Montgomery Ladder* e il *Left-to-Right NAF*.

2.3.1 Double & Add

Data una curva ellittica E e un punto $P \in E(K)$ vogliamo calcolare kP dove k è un numero intero ≥ 2. Sia $b_{n-1} \cdot 2^{n-1} + e_{n-2} \cdot 2^{n-2} + \ldots + e_1 \cdot 2 + e_0$ l'espansione binaria di

k con n numero delle cifre binarie, $e_{\mathrm{i}} \in \{0, 1\}$ e $e_{\mathrm{n}-1} = 1$. Allora l'algoritmo *Double & Add* opera nel modo seguente:

Algorithm 2 Double & Add Algorithm

INPUT: $e_0, \ldots, e_{\mathrm{n}-1}$, P, n

OUTPUT: kP

```
Q = P
if e_0 = 0 then
    R = O
else
    R = P
end if
for i = 1 to n − 1 do
    Q = 2Q {Point Doubling}
    if e_i = 1 then
        R = R + Q {Point Addition}
    end if
end for
return R
```

Dimostriamo che il risultato ottenuto è proprio kP.

Dimostrazione. Durante l'i-esima iterazione del ciclo, il valore di Q è pari a $2^{\mathrm{i}} \cdot P$. Poiché R viene incrementato, sommandolo a Q, soltanto quando $e_{\mathrm{i}} = 1$, il valore finale di R è dato da:

$$\sum_{i=0}^{n-1}(e_{\mathrm{i}}2^{\mathrm{i}}) \cdot P = (\sum_{i=0}^{n-1} e_{\mathrm{i}}2^{\mathrm{i}}) \cdot P = kP$$

□

Rispetto alle $k - 1$ *Point Addition* viste prima, il Double & Add, richiede $\lfloor \log_2 k \rfloor$ *Point Doubling* e, sia $t \leq \lfloor \log_2 k \rfloor$ il numero delle cifre binarie poste a "1", un numero di *Point Addition* pari a t.

Un'altra possibile versione del *Double & Add*, nel quale si parte dalla seconda cifra più significativa fino alla cifra meno significativa e non viceversa, è la seguente:

Algorithm 3 Double & Add Algorithm (alternative)

INPUT: $e_0, \ldots, e_{n-1}$, P, n

OUTPUT: kP

```
R = P
for i = n − 2 downto 0 do
    R = 2R {Point Doubling}
    if e_i = 1 then
        R = R + P {Point Addition}
    end if
end for
return R
```

2.3.2 Montgomery Ladder

L'algoritmo *Montgomery Ladder* si differenzia dal *Double & Add* poiché, sia nel caso in cui l'i-esima cifra binaria sia pari 0 o a 1, esegue una *Point Addition* e una *Point Doubling*. Vediamo nel dettaglio il suo funzionamento.

Algorithm 4 Montgomery Ladder

INPUT: $e_0, \ldots, e_{n-1}$, P, n

OUTPUT: kP

```
R_0 = P
R_1 = 2P
for i = n − 2 to 0 do
    if e_i = 1 then
        R_0 = R_0 + R_1 {Point Addition}
        R_1 = 2R_1 {Point Doubling}
    else
        R_1 = R_0 + R_1 {Point Addition}
        R_0 = 2R_0 {Point Doubling}
    end if
end for
return R_0
```

Il numero delle operazioni effettuate è quindi pari a $2\lfloor \log_2 k \rfloor$, poiché vengono eseguite $\lfloor \log_2 k \rfloor$ *Point Addition* e $\lfloor \log_2 k \rfloor$ *Point Doubling*. Un modo per ottimizzare l'algoritmo è la *parallelizzazione* delle operazioni, cioè eseguire su un processore le *Point Addition* e su un altro le *Point Doubling*. Peraltro il fatto di eseguire entrambe le operazioni per tutte le cifre binarie permette all'algoritmo di avere una naturale resistenza ad attacchi di tipo *Simple Power Analysis* (SPA), che vedremo nei capitoli successivi.

2.3.3 Left-to-Right NAF

Questo algoritmo, a differenza dei precedenti due, fa uso di una rappresentazione detta *"Non-Adjacent-Form"* (NAF).

Definizione 2.3 (Non-Adjacent-Form [15])**.** La *Non-Adjacent-Form* di un numero intero positivo k è data da una sequenza di cifre di lunghezza n, $e_{\mathrm{n}-1}e_{\mathrm{n}-2}\dots e_0$ con $e_{\mathrm{i}} \in \{0, \pm 1\}$, $e_{\mathrm{n}-1} \neq 0$ e senza due cifre e_{i} consecutive diverse da 0, tale che $k = \sum_{i=0}^{n-1} e_{\mathrm{i}} 2^{\mathrm{i}}$.

Teorema 2.1 (Proprietà della rappresentazione NAF [15])**.** *Sia k un intero positivo.*

1. *k ha un'unica rappresentazione NAF denotata $NAF(k)$.*
2. *$NAF(k)$ ha il minor numero di cifre diverse da zero di qualsiasi rappresentazione segnata di k.*
3. *$NAF(k)$ è al massimo più lunga di un'unità rispetto alla rappresentazione binaria di k.*
4. *Se la lunghezza di $NAF(k)$ è n, allora $\frac{2^{\mathrm{n}}}{3} < k < \frac{2^{\mathrm{l}+1}}{3}$.*
5. *La densità media di cifre diverse da 0 in tutte le $NAF(k)$ di una stessa lunghezza n è approssimativamente $\frac{1}{3}$.*

Esempio 2.3. La rappresentazione NAF di 7 è $(100-1)_{\mathrm{NAF}} \rightarrow \sum_{i=0}^{3} e_{\mathrm{i}} 2^{\mathrm{i}} = 8 - 1 = 7$.

La rappresentazione NAF può essere efficientemente calcolata attraverso il seguente algoritmo:

Algorithm 5 Calcolo di $NAF(k)$

INPUT: k

OUTPUT: $NAF(k)$

```
i = 0
while k ≥ 1 do
  if k mod 2 ≠ 0 then
    e_i = 2 − (k mod 4)
    k = k − e_i
  else
    e_i = 0
  end if
  k = k/2
  i = i + 1
end while
return e_{i−1}, e_{i−2}, ..., e_0
```

Come è possibile notare, l'algoritmo consiste nel dividere ripetutamente k per 2. Nel caso in cui k è dispari l'i-esima cifra e_i sarà data dalla differenza tra 2 e il resto della divisione tra k e 4, che essendo k dispari può essere pari a 1 o a 3, quindi $e_\mathrm{i} \in \{-1, 1\}$. Si fa poi la differenza tra k ed e_i in modo che k nel ciclo successivo sia divisibile per 2, così da non avere due cifre diverse da 0 consecutive. Se k è pari infatti $e_\mathrm{i} = 0$.

Detto questo, l'algoritmo *Left-to-Right NAF* non è altro che un *Double & Add* modificato per essere utilizzato con $NAF(k)$ invece della rappresentazione binaria di k. L'idea alla base è che, come detto in precedenza, la differenza di due punti è efficiente come la somma, non essendo che un caso particolare di quest'ultima. Per quanto riguarda il numero delle operazioni richieste per il calcolo di kP possiamo effettuare una stima tenendo conto, in particolare, di quanto detto dalle proprietà (3) e (5) del teorema (2.1). L'algoritmo infatti esegue una *Point Doubling* per ogni cifra di $NAF(k)$ e una *Point Addition* per ogni cifra diversa da 0. Dalla proprietà

(3) deriva che, nel caso peggiore, il numero di cifre di $NAF(k)$ è pari a $(\lfloor \log_2 k \rfloor + 1)$. Invece, dalla proprietà (5), deriva che il numero di cifre diverse da 0 è, in media, pari a $(\lfloor \log_2 k \rfloor + 1)/3$. Quindi approssimativamente vengono effettuate $(\lfloor \log_2 k \rfloor + 1)$ *Point Doubling* e $(\lfloor \log_2 k \rfloor + 1)/3$ *Point Addition*.

Algorithm 6 Left-To-Right NAF

INPUT: k, P

OUTPUT: kP

si usa l'algoritmo 5 per calcolare $NAF(k)$
$R = \mathcal{O}$
for $i = n - 1$ *to* 0 **do**
  $R = 2R$
  if $e_{\mathrm{i}} = 1$ **then**
    $R = R + P$
  end if
  if $e_{\mathrm{i}} = -1$ **then**
    $R = R - P$
  end if
end for
return R

2.4 Curve ellittiche su campi finiti

Dato un campo finito $\mathbb{F}_{\mathrm{p}}$ con p primo e sia E una curva ellittica definita su $\mathbb{F}_{\mathrm{p}}$. Allora, se la caratteristica del campo $char(\mathbb{F}_{\mathrm{p}}) \neq 2, 3$, l'insieme dei punti della curva sarà dato da:

$$E(\mathbb{F}_{\mathrm{p}}) = \{(x, y) \in (\mathbb{F}_{\mathrm{p}})^2 \mid y^2 = x^3 + ax + b \ (mod\ p),\ 4a^3 + 27b^2 \neq 0 \ (mod\ p)\} \cup \{\mathcal{O}\}$$

con a, b interi in $\mathbb{F}_{\mathrm{p}}$.

Per quanto riguarda la *Point Addition* rimangono valide le formule viste nel paragrafo 2.2, ma anch'esse assumono carattere modulare:

$$P \neq Q \begin{cases} m = \frac{(y_2 - y_1)}{(x_2 - x_1)} \ (mod\ p) \\ x_3 = m^2 - x_1 - x_2 \ (mod\ p) \\ -y_3 = m(x_1 - x_3) - y_1 \ (mod\ p) \end{cases} \qquad P = Q \begin{cases} m = \frac{3x_1^2 + a}{2y_1} \ (mod\ p) \\ x_3 = m^2 - 2x_1 \ (mod\ p) \\ -y_3 = m(x_1 - x_3) - y_1 \ (mod\ p) \end{cases}$$

Un'importante quantità legata ad una curva ellittica E su un campo finito è il numero dei punti della curva stessa. Questa quantità è detta *ordine della curva* E ed è denotata con $\#E(\mathbb{F}_p)$. A tal proposito citiamo il seguente teorema che ci permette di registreggere ad un intervallo i possibili valori di $\#E(\mathbb{F}_p)$:

Teorema 2.2 (Teorema di Hasse)**.** *Sia E una curva ellittica definita su un campo $\mathbb{F}_q$. Allora*

$$q + 1 - 2\sqrt{q} \leq \#E(\mathbb{F}_q) \leq q + 1 + 2\sqrt{q}.$$

Tuttavia dal punto di vista crittografico questo teorema è poco utile, poiché in tale contesto è richiesta la determinazione esatta del valore di $\#E(\mathbb{F}_p)$. Un algoritmo utilizzato a tale scopo è l'*Algoritmo di Schoof*, che con un tempo polinomiale è in grado di restituire univocamente $\#E(\mathbb{F}_p)$ data una curva $E(\mathbb{F}_p)$, sfruttando il *Teorema di Hasse* e il *Teorema Cinese del Resto*, come mostrato in [19].

Sia invece P un punto di $E(\mathbb{F}_p)$. Allora l'insieme dei multipli di P è un sottogruppo ciclico H di $E(\mathbb{F}_p)$ e P è il generatore del sottogruppo. L'ordine della curva $\#E(\mathbb{F}_p)$ e l'ordine di un suo sottogruppo $\#H$ (che essendo ciclico è anche l'ordine del generatore), come detto nell'Appendice A, sono legati dal *Teorema di Lagrange*, cioè $\#H$ è un divisore di $\#E(\mathbb{F}_p)$ e la quantità $h = \#E(\mathbb{F}_p)/\#H$ con $h \in \mathbb{Z}$ è detta *cofattore*.

Lo standard *FIPS 186-2* raccomanda l'utilizzo di cinque curve ellittiche, diverse per livello di sicurezza. Queste curve sono denominate P_k, dove k è la massima potenza di 2 utilizzata per esprimere la cardinalità del campo $\mathbb{F}_p$:

$P_{192} = 2^{192} - 2^{64} - 1$

$P_{224} = 2^{224} - 2^{96} + 1$

$P_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$

$P_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$

$P_{521} = 2^{521} - 1$

Capitolo 3

Crittografia su curve ellittiche

3.1 Crittosistemi su curve ellittiche

3.1.1 Parametri di dominio

Nei crittosistemi basati su curve ellittiche sono utilizzati una serie di parametri che permettono di definire una specifica curva ellittica su un campo finito $E(\mathbb{F}_\mathrm{p})$. Questi parametri sono detti *parametri di dominio* e devono essere scelti in modo opportuno, al fine di rendere l'ECDLP resistente ad un maggior numero possibile di attacchi.

Definizione 3.1 (Parametri di dominio)**.**

- L'ordine del campo p.
- Due coefficienti a e $b \in \mathbb{F}_\mathrm{p}$ che definiscono l'equazione della curva (es. $y^2 = x^3 + ax + b$).
- il generatore G del sottogruppo H.
- l'ordine del generatore $n = \#H$, che è anche l'ordine del sottogruppo H ed è un numero primo.
- il cofattore $h = \#E(\mathbb{F}_\mathrm{p})/n$.

$$D = (p, a, b, G, n, h).$$

3.1.2 Codifica e decodifica del messaggio

In quasi tutti i sistemi crittografici, abbiamo bisogno di un metodo per mappare il testo in chiaro del messaggio in un valore numerico sul quale eseguire delle operazioni matematiche. Nel caso della crittografia ellittica, invece, abbiamo bisogno di un metodo per mappare il testo in chiaro su una serie di punti di una curva ellittica. In questo modo potranno essere eseguite delle operazioni su tali punti per cifrare il messaggio. Tuttavia, non esistono algoritmi che in tempo polinomiale sono in grado di trovare un elevato numero di punti su una curva arbitraria. È da notare, che non cerchiamo dei punti a caso, ma un metodo sistematico per trovare dei punti su una curva elittica che sono in qualche modo collegati al testo in chiaro del messaggio. Appunto per questo motivo, si utilizza un algoritmo probabilistico con basse possibilità di fallimento. In particolare, il metodo proposto è stato ideato da *Neal Koblitz*. Vediamone il funzionamento.

Consideriamo una curva ellittica su un campo finito $E(\mathbb{F}_{\mathrm{p}})$. Supponiamo che il nostro alfabeto sia composto dalle cifre $0, \ldots, 9$ e dalle lettere A,B,...,X,Y,Z. Ogni carattere c del nostro alfabeto potrà essere quindi mappato, ad esempio, con un numero intero compreso tra 0 e 35.

1. Scegliamo un numero k. Questo valore inciderà sulla probabilità di fallimento nell'associare un carattere ad un punto della curva, che è pari a $1/2^{\mathrm{k}}$ [20]. Valori realistici possono essere scelti nell'intervallo $[30, 50]$.

2. Per ogni carattere c del messaggio si cerca di risolvere il seguente sistema:

$$\begin{cases} x = ck + j \ (mod\ p) \\ y^2 = x^3 + ax + b \ (mod\ p) \end{cases}$$

   dove j varia all'interno dell'intervallo $[1, k-1]$. Questo passo viene ripetuto, incrementando j, finchè non si riesce a risolvere il sistema, ottenendo le coordinate del punto $P = (x, y)$ che codifica il carattere c. Nel caso in cui non fosse possibile risolvere il sistema $\forall j \in [1, k-1]$, si avrebbe il fallimento dell'algoritmo.

3. Se il passo 2 è andato a buon fine, ogni carattere del nostro messaggio è stato codificato in un punto della curva ellittica.

Per quanto riguarda invece la decodifica, preso un punto $P = (x, y)$ il carattere c è dato dal più grande intero minore di $(x - 1)/k$.

Osserviamo che i parametri della curva devono essere scelti adeguatamente. Infatti, supponendo che ogni carattere sia rappresentato dal corrispondente valore decimale ASCII, il massimo valore di c sarà 127. Scegliendo ad esempio $k = 40$. Allora, la coordinata $x = 127 \cdot 40 + j$ con $j \in [1, k - 1] = [1, 39]$. Quindi nel peggiore dei casi $x = 5119$. Ora, affinché la coordinata x possa essere pari a 5119 bisogna selezionare una curva con un valore di $p > 5119$, essendo le coordinate dei punti limitate nell'intervallo $[1, p - 1]$.

3.1.3 Validazione della chiave pubblica

Nei prossimi paragrafi vedremo diversi sistemi di crittografia ellittica a chiave pubblica. É da considerare che in tali sistemi, in particolare nell'ECDH (nel quale a partire dalla chiave pubblica della controparte si costruisce la chiave segreta condivisa), una chiave pubblica adeguatamente costruita potrebbe rivelare delle informazioni circa la chiave privata dell'altro. Vediamo perciò un algoritmo per verificare che, la chiave pubblica ricevuta dall'altro interlocutore, possegga determinate proprietà e quindi sia valida. É da premettere che l'esito positivo non garantisce che una chiave privata, dal quale poi è stata creata la pubblica, sia stata calcolata ed esista.

Supponiamo quindi che Alice riceva la chiave pubblica di Bob $K_{\mathrm{B}}^{+} = Q = (x_{\mathrm{Q}}, y_{\mathrm{Q}})$ e, dati i parametri di dominio $D = (p, a, b, G, n, h)$, voglia verificarne la validità.

Algorithm 7 Algoritmo di validazione della chiave pubblica

INPUT: $D = (p, a, b, G, n, h)$, chiave pubblica $K_{\mathrm{B}}^{+} = Q = (x_{\mathrm{Q}}, y_{\mathrm{Q}})$.

OUTPUT: Valida o non valida.

si verifica che $Q \neq \mathcal{O}$

si verifica che x_{Q} e $y_{\mathrm{Q}} \in [0, p - 1]$, cioè che siano elementi del campo $\mathbb{F}_{\mathrm{p}}$

si verifica che il punto Q soddisfi l'equazione della curva definita da a e b

si verifica che $n \cdot Q = \mathcal{O}$

return se tutte le verifiche sono andate a buon fine *Valida*, altrimenti *Invalida*.

3.1.4 ElGamal su curve ellittiche

Abbiamo visto nel paragrafo (1.5.2) il crittosistema *ElGamal* basato sul problema del logaritmo discreto. Vediamo ora, invece, l'analogo nella crittografia ellittica.

Supponiamo che Alice voglia inviare un messaggio $M = m_0 \dots m_{l-1}$ a Bob e che i due interlocutori siano già d'accordo sulla curva ellittica E da utilizzare (sono quindi fissati i parametri di dominio $D = (p, a, b, G, n, h)$).

1. Alice e Bob calcolano le loro chiavi pubbliche e private:
   - Alice: $K_A^- = \alpha$ con $\alpha \in [1, n-1]$, $K_A^+ = \alpha \cdot G$.
   - Bob: $K_B^- = \beta$ con $\beta \in [1, n-1]$, $K_B^+ = \beta \cdot G$.
2. Alice codifica l'i-esimo carattere del messaggio m_i in un punto della curva, che denotiamo M_i (come visto, ad esempio, nel paragrafo precedente).
3. Alice sceglie un intero $k \in [1, n-1]$, calcola $C = M_i + k \cdot K_B^+$ e invia a Bob la coppia (kG, C).
4. Bob ricevuto (kG, C) può ottenere M_i calcolando prima, $\beta \cdot kG = k \cdot K_B^+$ e poi, $C - \beta \cdot kG = M + k \cdot K_B^+ - \beta \cdot kG = M_i$.
5. Bob può ora ottenere il carattere m_i applicando un algoritmo di decodifica al punto M_i.

È chiaro che un attacker potrebbe ottenere la chiave privata $K_A^- = \alpha$ dalla chiave pubblica $K_A^+ = \alpha \cdot G$, se fosse in grado di risolvere l'ECDLP.

3.1.5 ECDH

L'*Elliptic Curve Diffie-Hellman* è la versione ellittica del protocollo visto nel paragrafo (1.5.1) e anch'esso viene utilizzato da due entità comunicanti per stabilire una chiave segreta condivisa K^S su un canale non sicuro.

Per descrivere il protocollo partiamo dallo stesso scenario del paragrafo precedente. Ovvero i due interlocutori Alice e Bob si sono già accordati sulla curva E e quindi sono fissati i parametri pubblici $D = (p, a, b, G, n, h)$.

1. Alice e Bob calcolano le loro chiavi pubbliche e private:
   - Alice: $K_{\mathrm{A}}^{-} = d_{\mathrm{A}}$ con $d_{\mathrm{A}} \in [1, n-1]$, $K_{\mathrm{A}}^{+} = d_{\mathrm{A}} \cdot G$.
   - Bob: $K_{\mathrm{B}}^{-} = d_{\mathrm{B}}$ con $d_{\mathrm{B}} \in [1, n-1]$, $K_{\mathrm{B}}^{+} = d_{\mathrm{B}} \cdot G$.
2. Alice e Bob si scambiano le rispettive chiavi pubbliche e ne verificano la validità con l'algoritmo (7) visto nel paragrafo (3.1.3).
3. Alice e Bob calcolano la chiave segreta condivisa:
   - Alice: $K_{\mathrm{A}}^{\mathrm{S}} = d_{\mathrm{A}} \cdot K_{\mathrm{B}}^{+} = d_{\mathrm{A}} \cdot d_{\mathrm{B}} \cdot G$.
   - Bob: $K_{\mathrm{B}}^{\mathrm{S}} = d_{\mathrm{B}} \cdot K_{\mathrm{A}}^{+} = d_{\mathrm{B}} \cdot d_{\mathrm{A}} \cdot G$.
     $\Rightarrow K_{\mathrm{A}}^{\mathrm{S}} = K_{\mathrm{B}}^{\mathrm{S}} = K^{\mathrm{S}}$.

La chiave segreta K^{S} così calcolata potrà, o essere utilizzata direttamente come chiave in un algoritmo di crittografia simmetrica, oppure a sua volta per generare una chiave di cifratura, ad esempio dandola in input ad una funzione hash. Nel primo caso, se Alice volesse inviare un messaggio M a Bob:

1. dopo aver codificato l'i-esimo carattere m_{i} in un punto della curva M_{i}, Alice calcola $C = M_{\mathrm{i}} + K^{\mathrm{S}}$.
2. Alice invia C a Bob.
3. Bob calcola $M_{\mathrm{i}} = C - K^{\mathrm{S}}$ e decodifica M_{i} ottenendo il carattere m_{i}.

Un attacker per decifrare C dovrebbe quindi venire a conoscenza, date $K_{\mathrm{A}}^{+} = d_{\mathrm{A}}G$ e $K_{\mathrm{B}}^{+} = d_{\mathrm{B}}G$, della chiave segreta $K^{\mathrm{S}} = d_{\mathrm{A}}d_{\mathrm{B}}G$. Questo problema è anche noto come *Elliptic Curve Diffie-Hellman Problem* (ECDHP) e si può dimostrare che:

$$ECDHP \leq ECDLP$$

cioè, se si è in grado di risolvere l'ECDLP si è in grado di risolvere anche l'ECDHP (infatti sapendo risolvere l'ECDLP si potrebbe, ad esempio, ricavare la chiave privata di Alice $K_{\mathrm{A}}^{-} = d_{\mathrm{A}}$ dalla sua chiave pubblica e poi moltiplicare la chiave privata di Alice con la chiave pubblica di Bob, ottenendo K^{S}). Per la tesi inversa non esiste invece alcuna dimostrazione.

Osserviamo che, senza un metodo di autenticazione degli interlocutori, l'ECDH è soggetto ad attacchi di tipo *man in the middle*. In questa tipologia di attacco un terzo soggetto ritrasmette o altera la comunicazione tra due parti che credono di comunicare direttamente tra loro. Un esempio di questo attacco è mostrato in figura (3.1). Come si vede, Carl intercetta la chiave pubblica di Alice e invia la propria chiave pubblica a Bob. Quest'ultimo quindi crederà di stabilire una chiave segreta condivisa con Alice ma in realtà ciò sta accadendo con Carl, che utilizzerà questa chiave per decifrare i messaggi inviati da Bob ad Alice e volendo potrà anche alterarli.

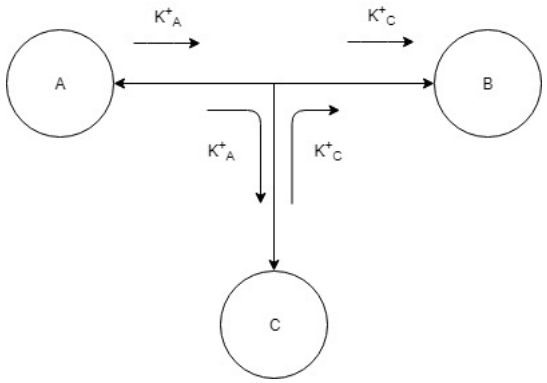

Figura 3.1: Attacco Man In The Middle

Come abbiamo già detto, per evitare questa tipopologia di attacchi è necessario un metodo di autenticazione. A tal proposito, bisogna effettuare la seguente distinzione:

- *Chiave statica*: se una chiave è utilizzata per un periodo di tempo relativamente lungo, o meglio per più istanze del meccanismo di scambio delle chiavi.
- *Chiave Effimera*: se una chiave viene generata e utilizzata per una sola istanza dello stesso meccanismo.

Le chiavi pubbliche statiche sono solitamente autenticate mediante dei *Certificati digitali* rilasciati da enti terzi riconosciuti come *autorità di certificazione*; le chiavi effimere, invece, non sono di norma autenticate, quindi l'identità degli interlocutori dev'essere verificata con altre modalità. Se da un lato le chiavi statiche permettono

una più facile autenticazione, dall'altro non garantiscono la *segretezza in avanti* o *Perfect Forward Secrecy* (PFS). Infatti, in caso di compromissione di una chiave pubblica statica, l'attacker potrebbe calcolare la chiave segreta è decifrare tutte le conversazioni passate e future tra i due interlocutori. Al contrario la PFS è garantita con le chiavi effimere poiché valide soltanto per una sessione. Una soluzione ibrida che soddisfa sia il requisito di autenticazione che quello di PFS è la seguente:

1. Alice e Bob hanno ciascuno una coppia di chiavi statica, con le chiavi pubbliche autenticate da certificati digitali.

2. Ogniqualvolta Alice e Bob vogliono comunicare generano ciascuno una coppia di chiavi effimere che utilizzano per creare la chiave segreta condivisa della sessione.

3. Sia Alice che Bob firmano digitalmente la propria chiave pubblica effimera appena generata utilizzando le chiavi statiche e la inviano alla controparte.

4. Entrambi gli interlocutori verificano la firma digitale dell'altro e il certificato, così da accertarne l'identità. Se la verifica va a buon fine utilizzano la chiave pubblica effimera dell'altro e la propria chiave privata effimera per generare la chiave segreta condivisa della sessione.

In questo caso, quindi, le chiavi statiche vengono utilizzate soltanto per l'autenticazione mentre la chiave segreta condivisa viene generata mediante delle chiavi effimere diverse per ogni sessione.

3.1.6 ECMQV

L'ECMQV, da *Menezes-Qu-Vanstone*, è un protocollo basato sullo schema di *Diffie-Hellman* che consente a due entità, le quali già possiedono una copia autenticata della chiave pubblica l'una dell'altra, di stabilire una chiave segreta condivisa [21]. Lo schema è molto simile a quello ibrido, chiavi statiche-effimere, visto nel paragrafo precedente. Infatti l'ECMQV prevede che i due interlocutori generino una coppia di chiavi effimere ogniqualvolta debbano comunicare. Questa coppia di chiavi viene utilizzata per il calcolo della *firma implicita*, necessaria per verificare l'identità della controparte. Vediamo in dettaglio lo schema.

Allo stato iniziale Alice e Bob hanno ciascuno una coppia di chiavi statiche, rispettivamente $(K_{\mathrm{A}}^{+} = xG, K_{\mathrm{A}}^{-} = x)$ e $(K_{\mathrm{B}}^{+} = yG, K_{\mathrm{B}}^{-} = y)$, della quale la chiave pubblica è nota all'altro. Sono inoltre fissati i parametri di dominio $D = (p, a, b, G, n, h)$.

1. Alice e Bob calcolano le chiavi effimere:
   - Alice: $K_{\mathrm{A,E}}^{-} = \alpha$ con $\alpha \in [1, n-1]$, $K_{\mathrm{A,E}}^{+} = \alpha G$.
   - Bob: $K_{\mathrm{B,E}}^{-} = \beta$ con $\beta \in [1, n-1]$, $K_{\mathrm{B,E}}^{+} = \beta G$.
2. Alice e Bob si scambiano le chiavi pubbliche effimere.
3. Alice e Bob validano la chiave pubblica effimera ricevuta dall'altro con l'algoritmo (7) visto nel paragrafo (3.1.3).
4. Alice e Bob calcolano la *firma implicita*:
   - Alice: $S_{\mathrm{A}} = (K_{\mathrm{A,E}}^{-} + \bar{A} \cdot K_{\mathrm{A,S}}^{-})\ (mod\ n)$
     con $\bar{A} = (\bar{x}\ mod\ 2^{\lfloor(\log_2 \mathrm{n})/2\rfloor}) + 2^{\lfloor(\log_2 \mathrm{n})/2\rfloor}$ e $\bar{x}$ rappresentazione intera della coordinata x di $K_{\mathrm{A,E}}^{+}$.
   - Bob: $S_{\mathrm{B}} = (K_{\mathrm{B,E}}^{-} + \bar{B} \cdot K_{\mathrm{B,S}}^{-})\ (mod\ n)$
     con $\bar{B} = (\bar{x}\ mod\ 2^{\lfloor(\log_2 \mathrm{n})/2\rfloor}) + 2^{\lfloor(\log_2 \mathrm{n})/2\rfloor}$ e $\bar{x}$ rappresentazione intera della coordinata x di $K_{\mathrm{B,E}}^{+}$.
5. Alice e Bob calcolano la chiave segreta condivisa:
   - Alice: $K_{\mathrm{A}}^{\mathrm{S}} = hS_{\mathrm{A}}(K_{\mathrm{B,E}}^{+} + \bar{B} \cdot K_{\mathrm{B,S}}^{+})$
     $\Rightarrow K_{\mathrm{A}}^{\mathrm{S}} = hS_{\mathrm{A}}(\beta G + \bar{B} \cdot yG) = hS_{\mathrm{A}}(\beta + \bar{B} \cdot y)G = hS_{\mathrm{A}}S_{\mathrm{B}}G$
   - Bob: $K_{\mathrm{B}}^{\mathrm{S}} = hS_{\mathrm{B}}(K_{\mathrm{A,E}}^{+} + \bar{A} \cdot K_{\mathrm{A,S}}^{+})$
     $\Rightarrow K_{\mathrm{B}}^{\mathrm{S}} = hS_{\mathrm{B}}(\alpha G + \bar{A} \cdot xG) = hS_{\mathrm{B}}(\alpha + \bar{A} \cdot x)G = hS_{\mathrm{B}}S_{\mathrm{A}}G$

$\Rightarrow K_{\mathrm{A}}^{\mathrm{S}} = K_{\mathrm{B}}^{\mathrm{S}}$

Come si vede, Alice e Bob posso verificare la validità della firma indirettamente attraverso le seguenti eguaglianze:

$$S_{\mathrm{A}} \cdot G = (K_{\mathrm{A,E}}^{+} + \bar{A} \cdot K_{\mathrm{A,S}}^{+})\ \ e\ \ S_{\mathrm{B}} \cdot G = (K_{\mathrm{B,E}}^{+} + \bar{B} \cdot K_{\mathrm{B,S}}^{+})$$

appunto per questo è detta *"implicita"*.

3.1.7 ECDSA

Vediamo infine la versione in crittografia ellittica del *Digital Signature Algorithm* (DSA), visto nel paragrafo (1.5.3). Ricordiamo che uno schema di firma digitale si compone di due algoritmi, uno per la generazione della firma e uno per la sua verifica.

Supponiamo che Alice voglia firmare un messaggio m da inviare a Bob. Come al solito, va prima stabilita una curva E su un campo finito $\mathbb{F}_p$ e quindi i parametri di dominio $D = (p, a, b, G, n, h)$ e va generata la coppia di chiavi $(K_A^- = \alpha, K_A^+ = \alpha G)$.

Generazione della firma.

1. Alice sceglie un valore $k \in [1, n-1]$ e calcola $Q = k \cdot G$.
2. Alice calcola $r = \bar{x} \ mod \ n$ con $\bar{x}$ rappresentazione intera della coordinata x di Q. Se $r = 0$ si ripete dal passo 1.
3. Alice calcola $s = k^{-1}(H(m) + K_A^- \cdot r) \ mod \ n$ con H funzione di hash. Se $s = 0$ si ripete dal passo 1.

   $\Rightarrow$ la firma è la coppia (r, s).
4. Alice invia a Bob il messaggio m e la firma digitale (r, s).

Verifica della firma.

1. Bob verifica che r e s siano interi nell'intervallo $[1, n-1]$. Se tale condizione non vale, la verifica termina senza successo.
2. Bob calcola:

   - $\omega = s^{-1} \ mod \ n$;
   - $u_1 = H(m) \cdot \omega \ mod \ n$;
   - $u_2 = r \cdot \omega \ mod \ n$;
   - $P = u_1 \cdot G + u_2 \cdot K_A^+$.

   Se $P = \mathcal{O}$ la verifica termina senza successo.

3. Bob calcola $v = \bar{x}\ mod\ n$ con $\bar{x}$ rappresentazione intera della coordinata x di P.

4. se $v = r$ la verifica termina con successo, se $v \neq r$ viceversa.

Dimostriamo che se la firma è autentica $v = r$.

Dimostrazione.

$$P = u_1 \cdot G + u_2 \cdot K_{\mathrm{A}}^{+} \qquad \text{ma } K_{\mathrm{A}}^{+} = K_{\mathrm{A}}^{-} \cdot G$$
$$P = u_1 \cdot G + u_2 \cdot K_{\mathrm{A}}^{-} \cdot G \qquad \text{raccogliamo } G$$
$$P = (u_1 \cdot + u_2 \cdot K_{\mathrm{A}}^{-}) \cdot G \qquad \text{sostituiamo } u_1 = H(m) \cdot \omega\ mod\ n \text{ e } u_2 = r \cdot \omega\ mod\ n$$
$$P = (H(m) \cdot \omega + r \cdot \omega \cdot K_{\mathrm{A}}^{-}) \cdot G \qquad \text{raccogliamo } \omega$$
$$P = (H(m) + r \cdot K_{\mathrm{A}}^{-}) \cdot \omega \cdot G \qquad \text{sostituiamo } \omega = s^{-1}\ mod\ n$$
$$P = (H(m) + r \cdot K_{\mathrm{A}}^{-}) \cdot s^{-1} \cdot G \qquad \text{sostituiamo } s^{-1} = k \cdot (H(m) + K_{\mathrm{A}}^{-} \cdot r)^{-1}\ mod\ n$$
$$P = k \cdot G = Q \qquad \text{quindi la coordinata x di } P \text{ è uguale a quella di } Q.$$

□

Un attacker per fingersi Alice dovrebbe riuscire ad ottenere k a partire da $Q = k \cdot G$, cioè dovrebbe risolvere l'ECDLP. Notiamo inoltre che se in qualche modo l'attacker riuscisse ad ottenere k, egli potrebbe con molta facilità ottenere la chiave privata di Alice da $s = k^{-1}(H(m) + K_{\mathrm{A}}^{-} \cdot r)\ mod\ n$. Ciò dimostra quanto sia fondamentale mantenere segreto k. A tal proposito è altresì importante che lo stesso valore k non sia utilizzato più di una volta. Infatti nel caso contrario è possibile dimostrare che un attacker potrebbe ottenere il valore di k senza troppe difficoltà.

Dimostrazione. Supponiamo che uno stesso valore k sia utilizzato per generare le due firme (r, s_1) e (r, s_2) su due messaggi m_1 e m_2. Allora avremo che:

$$s_1 = k^{-1}(H(m_1) + K_{\mathrm{A}}^{-} \cdot r)\ mod\ n \ \ e \ \ s_2 = k^{-1}(H(m_2) + K_{\mathrm{A}}^{-} \cdot r)\ mod\ n.$$

Moltiplicando entrambi i membri di s_1 e s_2 per k si ottiene:

$$s_1 \cdot k = (H(m_1) + K_{\mathrm{A}}^{-} \cdot r)\ mod\ n \ \ e \ \ s_2 \cdot k = (H(m_2) + K_{\mathrm{A}}^{-} \cdot r)\ mod\ n.$$

Sottraendo la seconda espressione alla prima:

$$k \cdot (s_1 - s_2) = H(m_1) - H(m_2) \; mod \; n \; \rightarrow k = \frac{H(m_1) - H(m_2)}{(s_1 - s_2)} \; mod \; n$$

per $s_1 \neq s_2 \; mod \; n$, che accade quasi certamente. □

3.2 ECDLP

La sicurezza dei sistemi crittografici appena visti è dovuta, come già accennato, all'intrattabilità dell'*Elliptic Curve Discret Logarithm Problem* (ECDLP).

Definizione 3.2 (Elliptic Curve Discret Logarithm Problem [15])**.** Data una curva ellittica E su un campo finito $\mathbb{F}_p$ e un punto $P \in E(\mathbb{F}_p)$ di ordine n. Sia $Q = kP$ con $k \in [1, n-1]$. Allora k è il logaritmo discreto di Q in base P.

É da notare che non esiste alcuna prova matematica dell'intrattabilità dell'ECDLP. Tuttavia, nonostante le numerose ricerche fin da quando la crittografia ellittica è stata proposta, nessun algoritmo che risolve in ogni caso il problema in tempo sub-esponenziale è stato trovato.

Vediamo alcuni metodi per la risoluzione dell'ECDLP.

3.2.1 Baby Step, Giant Step

Il primo metodo che vediamo per la risoluzione dell'ECDLP è quello proposto da *Daniel Shanks*, noto come *Baby Step, Giant Step* o appunto *Algoritmo di Shanks*. Questo metodo richiede approssimativamente $\sqrt{n}$ operazioni e una complessità spaziale pari a $\sqrt{n}$, quindi risulta efficiente soltanto per valori relativamente piccoli di n, poiché per n grandi il quantitativo di memoria necessario diviene eccessivo. Vediamo il suo funzionamento.

1. Si sceglie un intero $m \geq \sqrt{n}$.
2. *Baby Step*: si memorizza una lista di punti iP per $i \in [1, m-1]$. Il punto i-esimo è calcolato sommando P a $(i-1)P$.

3. *Giant Step*: si calcola $Q - jmP$ per $j \in [0, m-1]$ affinché non c'è una corrispondenza con uno dei punti memorizzati al passo precedente. Il calcolo di $Q - jmP$ viene effettuato aggiungendo $-mP$ a $Q - (j-1)mP$.

4. Se si trova una corrispondenza allora $iP = Q - jmP \rightarrow Q = (i + jm)P \rightarrow k = i + jm \ (mod\ n)$.

Facciamo un esempio.

Esempio 3.1. Consideriamo la curva ellittica E descritta dall'equazione $y^2 = x^3 + 2x + 1$ sul campo $\mathbb{F}_{41}$. Siano $P = (0, 1)$ e $Q = (30, 40)$.

1. Dal *Teorema di Hasse* (2.2) sappiamo che $n \leq 54$, quindi scegliamo $m = 8$.

2. Calcoliamo tutti i punti iP per $i \in [1, 7]$:

$m = 1$	$m = 2$	$m = 3$	$m = 4$	$m = 5$	$m = 6$	$m = 7$
$(0, 1)$	$(1, 39)$	$(8, 23)$	$(38, 38)$	$(23, 23)$	$(20, 28)$	$(26, 9)$

3. Calcoliamo $Q - jmP$ per $j \in [0, 7]$ fino a che non troviamo una corrispondenza:

$m = 0$	$m = 1$	$m = 2$
$(30, 40)$	$(9, 25)$	$(26, 9)$

4. Per $m = 2$ si ottiene una corrispondenza con $m = 7$ del passo precedente. Quindi abbiamo che:

$$7P = Q - 2 \cdot 8P \rightarrow 23P = Q \rightarrow k = 23.$$

Vediamo perché troviamo una corrispondenza. Sappiamo, per definizione, che $m^2 > n$, quindi possiamo assumere che la soluzione k soddisfi $0 \leq k < m^2$. Poniamo $k = k_0 + mk_1$ con $k_0 = k \ (mod\ m)$ e $0 \leq k_0 < m$. Poiché $k_1 = (k - k_0)/m$, allora $0 \leq k_1 < m$. Se $i = k_0$ e $j = k_1$, allora abbiamo

$$Q - k_1 mP = kP - k_1 mP = k_0 P.$$

3.2.2 Metodi di Pollard

Il principale svantaggio del *Baby Step Giant Step*, come già detto, è la sua complessità spaziale. I metodi di Pollard hanno una complessità temporale approssimativamente uguale a quella del *Baby Step Giant Step*, ma richiedono un quantitativo di memoria molto minore. Vedremo prima il metodo ρ e poi la sua generalizzazione al metodo λ.

Metodo ρ di Pollard.

Sia E una curva ellittica su un campo finito $\mathbb{F}_{\mathrm{P}}$. Allora consideriamo una funzione $f : E(\mathbb{F}_{\mathrm{p}}) \to E(\mathbb{F}_{\mathrm{p}})$ che si comporta in modo pseudo-casuale. Partendo da un punto casuale P_0 calcoliamo iterativamente $P_{\mathrm{i}+1} = f(P_{\mathrm{i}})$. Poiché $E(\mathbb{F}_{\mathrm{p}})$ è un insieme finito di punti, per qualche indice $i_0 < j_0$ si avrà $P_{\mathrm{i}_0} = P_{\mathrm{j}_0}$. Quindi:

$$P_{\mathrm{i}_0+1} = f(P_{\mathrm{i}_0}) = f(P_{\mathrm{j}_0}) = P_{\mathrm{j}_0+1}$$

e cosi anche $P_{\mathrm{i}_0+l} = P_{\mathrm{j}_0+l}$ $\forall l \geq 0$. Si ha quindi una periodicità dei P_{i} pari a $j_0 - i_0$. La figura (3.2) rappresenta il processo appena descritto e come si vede assomiglia alla lettera grega ρ, da ciò deriva il nome del metodo.

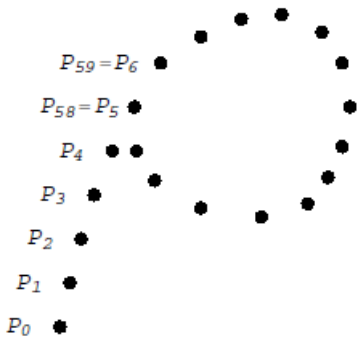

Figura 3.2: ρ di Pollard

Non abbiamo ancora parlato, tuttavia, di come scegliere una funziona adatta f tale che abbia un comportamento pseudo-casuale e che ci permetta di trarre delle informazioni utili da una corrispondenza. Un modo suggerito in [19] è il seguente:

- Dividiamo $E(\mathbb{F}_\mathrm{p})$ in r sottoinsiemi disgiunti $S_1, S_2, \ldots, S_\mathrm{r}$ approssimativamente della stessa dimensione.
- Scegliamo $2r$ interi casuali $a_\mathrm{i}, b_\mathrm{i}$ e sia $M_\mathrm{i} = a_\mathrm{i}P + b_\mathrm{i}Q$.
- Definiamo
$$f(g) = g + M_\mathrm{i}$$
con $g \in S_\mathrm{i}$.

La funzione f può essere immaginata come un cammino casuale in $E(\mathbb{F}_\mathrm{p})$ con punto di partenza $P_0 = a_0P + b_0Q$ con a_0 e b_0 scelti a caso e con dei possibili passaggi per i punti M_i. A partire da P_0 si calcolano quindi $P_\mathrm{j} = u_\mathrm{j}P + v_\mathrm{j}Q$, $P_{\mathrm{j}+1} = P_\mathrm{j} + M_\mathrm{i} = u_\mathrm{j}P + v_\mathrm{j}Q + a_\mathrm{i}P + b_\mathrm{i}Q = (u_\mathrm{j} + a_\mathrm{i})P + (v_\mathrm{j} + b_\mathrm{i})Q$ e cosi via, memorizzando i "coefficienti" di P e Q. Quando viene trovata una corrispondenza $P_{\mathrm{j}_0} = P_{\mathrm{i}_0}$ avremo:

$$u_{\mathrm{j}_0}P + v_{\mathrm{j}_0}Q = u_{\mathrm{i}_0}P + v_{\mathrm{i}_0}Q \rightarrow (u_{\mathrm{i}_0} - u_{\mathrm{j}_0})P = (v_{\mathrm{j}_0} - v_{\mathrm{i}_0})Q$$

quindi:

$$k = (v_{\mathrm{j}_0} - v_{\mathrm{i}_0})^{-1}(u_{\mathrm{i}_0} - u_{\mathrm{j}_0}) \ (mod\ n).$$

La complessità temporale richiesta dal metodo per trovare una corrispondenza è al massimo pari a $O(\sqrt{\frac{\pi n}{2}})$. Per quanto riguarda la complessità spaziale, un'implementazione "banale" richiede una quantità di memoria simile a quella richiesta dal *Baby Step Giant Step*, dovendo memorizzare tutti i punti P_i finché non si trova una corrispondenza. Tuttavia, è possibile fare di meglio sfruttando l'*Algoritmo di Floyd*. L'idea alla base è che una volta che c'è stata una corrispondenza tra due punti i cui indici si differenziano per una costante d, allora tutti i successivi indici distanti d l'uno dall'altro, generano una corrispondenza per la periodicità detta prima. Pertanto, possiamo calcolare le coppie $(P_\mathrm{i}, P_{2\mathrm{i}})$ per $i = 1, 2, \ldots,$ ma memorizzare soltanto la coppia corrente e non quelle precedenti. Queste coppie possono essere calcolate attraverso le seguenti regole:

$$P_\mathrm{i} = f(P_\mathrm{i}) \ \ e \ \ P_{2(\mathrm{i}+1)} = f(f(P_{2\mathrm{i}})).$$

Trovata quindi una coppia $(P_\mathrm{i}, P_{2\mathrm{i}})$ con $P_\mathrm{i} = P_{2\mathrm{i}}$ tutte le coppie con indici alla stessa distanza d, genereranno una corrispondenza.

Esempio 3.2. Consideriamo la curva E sul campo finito $\mathbb{F}_{1093}$ descritta dall'equazione $y^2 = x^3 + x + 1$. Siano $P = (0, 1)$ e $Q = (413, 959)$, vogliamo trovare k tale che $kP = Q$. É possibile dimostrare che l'ordine di P è 1067.

Suddividiamo $E(\mathbb{F}_{1093})$ in tre sottoinsiemi disgiunti S_1, S_2 e S_3, quindi $r = 3$. Scegliamo $2r = 6$ interi casuali a_i e b_i con $M_\mathrm{i} = a_\mathrm{i}P + b_\mathrm{i}Q$. Ad esempio:

$$M_0 = 4P + 3Q, \quad M_1 = 9P + 17Q, \quad M_2 = 19P + 6Q.$$

Definiamo la funzione $f : E(\mathbb{F}_{1093}) \to E(\mathbb{F}_{1093})$:

$$f(x, y) = (x, y) + M_\mathrm{i} \;\; se \;\; x \equiv i \; mod \; 3.$$

Scegliamo un punto di partenza del nostro cammino $P_0 = a_0P + b_0Q$ con a_0 e b_0 casuali. Ad esempio $P_0 = 3P + 5Q$.

Calcoliamo i punti, P_0, $P_1 = f(P_0)$, $P_2 = f(P_1), \ldots$, ottenendo una corrispondenza per $P_5 = P_{58}$ con $P_5 = 88P + 46Q$ e $P_{58} = 685P + 620Q$.Quindi:

$$88P + 46Q = 685P + 620Q \Rightarrow 597P = -574Q \Rightarrow k = -574^{-1} \cdot 597 \; mod \; 1067 = 499.$$

Perciò $499P = Q$. É chiaro che per giungere a questo risultato abbiamo dovuto memorizzare tutti i punti calcolati fino a che non si è trovata una corrispondenza. Utilizzando invece l'*Algoritmo di Floyd* si otterrebbe una corrispondenza per $i = 53$, cioè $P_{53} = P_{106}$, ottenendo il medesimo risultato ma con un notevole risparmio in termini di complessità spaziale, dovendo memorizzare soltanto la coppia corrente.

Un ulteriore modo per velocizzare il metodo, consiste nell'eseguirlo su più processori contemporaneamente, cioè parallelizzarlo. Supponendo quindi di disporre di T processori, un approccio banale sarebbe quello di eseguire l'algoritmo ρ di Pollard su ciascun processore indipendentemente, ottenendo una riduzione delle operazioni di un fattore pari a $\sqrt{T}$. Un diverso approccio, proposto da *Van Oorschot* e *Wiener*, consente invece di ottenere un miglioramento di un fattore pari a T quando T processori sono impiegati. L'idea è quella di far collidere le sequenze di punti prodotte da diversi processori. Più precisamente, per ogni processore viene scelto un diverso punto di partenza P_0, ma la funzione f utilizzata per calcolare i punti successivi è la stessa. Quindi, se le sequenze prodotte da due diversi processori collidono in un punto, allora a partire da quest'ultimo saranno identiche [15].

Metodo λ di Pollard.
Questo algoritmo è solitamente descritto nei termini di un *canguro addestrato* che cerca di catturare un *canguro selvatico*. Infatti è stato lo stesso Pollard a definirlo in [22] come *un metodo λ per catturare canguri* (anche in questo caso λ deriva dalla rappresentazione grafica del metodo. Figura 3.3.). L'idea alla base è di far partire prima il canguro addestrato T da un punto P_0 e memorizzare tutti i punti nei quali arriva, per un numero prefissato di salti N. Ad ogni salto, il canguro addrestato, arriva in un punto P_i e "piazza una trappola". Dopo N salti T sarà arrivato al punto che chiamiamo P_N. Poi viene fatto partire il canguro selvatico W da un punto ignoto Q_0. Se durante il suo percorso arriva in uno dei punti P_i in cui è passato T, il canguro selvatico viene catturato e l'ECDLP può essere risolto come visto nel metodo ρ. Ovviamente l'algoritmo si basa sul fatto che la "legge di salto", cioè la funzione f che calcola il prossimo punto in cui saltare, sia la stessa per i due canguri. Il costo computazionale del metodo λ è nell'ordine di $O(\sqrt{\frac{\pi n}{2}})$, tuttavia, a differenza del metodo ρ, bisogna conoscere un intervallo $[a, b] \subset [0, n]$ nel quale cercare k. Appunto per tale motivo è ritenuto più efficiente il metodo ρ.

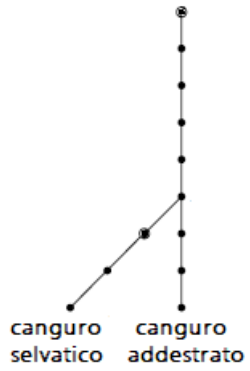

Figura 3.3: λ di Pollard

Peraltro, anche il metodo λ può essere parallelizzato nel caso in cui si abbiano a disposizione T processori. L'idea è di far partire prima T canguri addestrati da punti di partenza noti $P_0^1, P_0^2, \ldots, P_0^\text{T}$ e poi T canguri selvatici da punti di partenza sconoscuti $W_0^1, W_0^2, \ldots, W_0^\text{T}$. Quando uno dei canguri selvatici viene a trovarsi su

un punto in cui sono passati uno o più canguri addestrati l'algoritmo si conclude ed è possibile risolvere l'ECDLP [23].

É chiara quindi la differenza tra i due metodi di Pollard: il metodo ρ utilizza un solo cammino casuale aspettando che si formi un ciclo; il metodo λ utilizza due cammini casuali aspettando che si incrocino.

3.2.3 Metodo Pohling-Hellman

Sia E una curva ellittica su un campo finito $\mathbb{F}_{\mathrm{P}}$. Come al solito, dati due punti P e Q della curva, vogliamo trovare l'intero k tale che $Q = kP$. Supponiamo che P abbia un ordine pari a n, del quale conosciamo la fattorizzazione

$$n = \prod_{i}^{m} q_{\mathrm{i}}^{\mathrm{e_i}}.$$

L'idea alla base del metodo è di calcolare $k_{\mathrm{i}} = k \ mod \ q_{\mathrm{i}}^{\mathrm{e_i}}$ per ogni i e poi utilizzare il *Teorema Cinese del Resto* (Th. 1.2)

$$\begin{cases} k = k_1 \ mod \ q_1^{\mathrm{e_1}} \\ k = k_2 \ mod \ q_2^{\mathrm{e_2}} \\ \vdots \\ k = k_{\mathrm{m}} \ mod \ q_{\mathrm{m}}^{\mathrm{e_m}} \end{cases}$$

per ottenere k.
Per calcolare i singoli k_{i} per $1 \leq i \leq m$ scriviamo k_{i} nella sua rappresentazione in base p_{i}:

$$k_{\mathrm{i}} = z_0 + z_1 p_{\mathrm{i}} + z_2 p_{\mathrm{i}}^2 + \ldots + z_{\mathrm{e_i}-1} p_{\mathrm{i}}^{\mathrm{e_i}-1} \tag{3.1}$$

con $z_{\mathrm{t}} \in [0, p_{\mathrm{i}} - 1]$. I coefficienti z_{t} sono calcolati uno alla volta nel seguente modo:

- si determina un insieme di punti $T_{\mathrm{i}} = \{j(\frac{n}{p_{\mathrm{i}}})P \mid 0 \leq j \leq p_{\mathrm{i}}\}$.
- si calcola $Q_0 = (\frac{n}{p_{\mathrm{i}}})Q$ e si va a cercare una corrispondenza in T_{i} e quindi il coefficiente z_0. Infatti:
  $Q_0 = (\frac{n}{p_{\mathrm{i}}})Q = (\frac{n}{p_{\mathrm{i}}})k_{\mathrm{i}}P = (\frac{n}{p_{\mathrm{i}}})(z_0 + z_1 p_{\mathrm{i}} + z_2 p_{\mathrm{i}}^2 + \ldots + z_{\mathrm{e_i}-1} p_{\mathrm{i}}^{\mathrm{e_i}-1})P$
  $Q_0 = z_0(\frac{n}{p_{\mathrm{i}}})P + n(z_1 + z_2 p_{\mathrm{i}} + z_3 p_{\mathrm{i}}^2 + \ldots + z_{\mathrm{e_i}-1} p_{\mathrm{i}}^{\mathrm{e_i}-2})P$

$Q_0 = z_0(\frac{n}{p_i})P + (z_1 + z_2p_i + z_3p_i^2 + \ldots + z_{e_i-1}p_i^{e_i-2})nP$
$Q_0 = z_0(\frac{n}{p_i})P + (z_1 + z_2p_i + z_3p_i^2 + \ldots + z_{e_i-1}p_i^{e_i-2})\mathcal{O}$
$Q_0 = z_0(\frac{n}{p_i})P.$

- si calcola $Q_1 = (\frac{n}{p_i^2})(Q - z_0P)$ e si cerca una corrispondenza in T_i e quindi il coefficiente z_1. Infatti:
  $Q_1 = (\frac{n}{p_i^2})(Q - z_0P) = (\frac{n}{p_i^2})(k_iP - z_0P)$
  $Q_1 = (\frac{n}{p_i^2})[(z_0 + z_1p_i + z_2p_i^2 + \ldots + z_{e_i-1}p_i^{e_i-1})P - z_0P]$
  $Q_1 = (\frac{n}{p_i^2})(z_1p_i + z_2p_i^2 + \ldots + z_{e_i-1}p_i^{e_i-1})P$
  $Q_1 = z_1(\frac{n}{p_i})P + (\frac{n}{p_i^2})(z_2 + z_3p_i + \ldots + z_{e_i-1}p_i^{e_i-3})P$
  $Q_1 = z_1(\frac{n}{p_i})P + (z_2 + z_3p_i + \ldots + z_{e_i-1}p_i^{e_i-3})nP$
  $Q_1 = z_1(\frac{n}{p_i})P + (z_2 + z_3p_i + \ldots + z_{e_i-1}p_i^{e_i-3})\mathcal{O}$
  $Q_1 = z_1(\frac{n}{p_i})P.$

- Si ripete quindi il procedimento per ogni coefficiente z_j di k_i. In generale il termine Q_t sarà quindi dato da:

$$Q_t = \frac{n}{p^{t+1}}(Q - z_0P - z_1p_iP - z_2p_i^2P - \ldots - z_{e-2}p_i^{t-1}P).$$

- Trovati tutti i coefficienti z_j si può calcolare k_i attraverso la (3.1).

Determinati tutti i $k_i = k \; mod \; q_i^{e_i}$ si utilizza, come già detto, il teorema cinese del resto per determinare k e risolvere l'ECDLP.

Esempio 3.3. Consideriamo la curva $y^2 = x^3 + 1001x + 75$ definita sul campo $\mathbb{F}_{7919}$. Siano $P = (4023, 6036)$ e $Q = (4135, 3169)$ con P che ha un ordine pari a 7889. Vogliamo determinare k tale che $Q = kP$.

La fattorizzazione di n è

$$7889 = 7^3 \cdot 23.$$

Dobbiamo quindi calcolare $k_1 = k \; mod \; 7^3$ e $k_2 = k \; mod \; 23$.

Calcolo k_1.

- Calcoliamo $T_1 = \{j(\frac{7889}{7}P \mid 0 \leq j \leq 6\}$
  $T_1 = \{\mathcal{O}, (7801, 2071), (2516, 2309), (7285, 14), (7285, 7905), (2516, 5610), (7801, 5848)\}$

- Scriviamo k_1 nella sua rappresentazione in base 7: $k_1 = z_0 + z_1 7 + z_2 7^2$.
- Calcoliamo $Q_0 = \frac{n}{7}P = \frac{7^3 \cdot 23}{7}P = 7^2 \cdot 23P = (7801, 2071)$.
- Dalla corrispondenza in T_1 troviamo che $z_0 = 1$.
- Calcoliamo $Q_1 = \frac{7^3 \cdot 23}{7^2}(Q - P) = 7 \cdot 23(Q - P) = (7285, 14)$.
- Dalla corrispondenza in T_1 troviamo che $z_1 = 3$.
- Calcoliamo $Q_2 = \frac{7^3 \cdot 23}{7^3}(Q - P) = 7 \cdot 23(Q - P - 3 \cdot 7P) = (7285, 7905)$.
- Dalla corrispondenza in T_1 troviamo che $z_1 = 4$.
- Abbiamo quindi che $k_1 = 1 + 3 \cdot 7 + 4 \cdot 7^2 = 218$.

Calcolo k_2.

- Scriviamo k_1 nella sua rappresentazione in base 23: $k_2 = z_0$
- Calcoliamo $T_2 = \{j(\frac{7889}{23}P \mid 0 \leq j \leq 22\}$

  $T_2 = \{\mathcal{O}, (7190, 7003), (1579, 7204), (3633, 1339), (4703, 1815), (2708, 7505),$
  $(7507, 6510), (5414, 3375), (3405, 7560), (5770, 1387), (2599, 759),$
  $(6777, 1921), (6777, 5998), (2599, 7160), (5770, 6532), (3405, 359),$
  $(5414, 4544), (7507, 1409), (2708, 414), (4703, 6104), (3633, 6580),$
  $(1579, 715), (7190, 916)\}$.

- Calcoliamo $Q_0 = \frac{7^3 \cdot 23}{23}P = 7^3 P = (2599, 759)$.
- Dalla corrispondenza in T_2 troviamo che $z_0 = 10$.
- Abbiamo quindi che $k_2 = 10$.

Ora che abbiamo sia k_1 che k_2, attraverso il *Teorema Cinese del Resto*, risolviamo il sistema di congruenze:

$$\begin{cases} k = 218 \ mod \ 7^3 \\ k = 10 \ mod \ 23 \end{cases}$$

ottenendo $k = 4334$.

Il costo computazionale di tale metodo è nell'ordine di $\sqrt{p'}$ dove p' è il più grande divisore primo di n [24]. Ciò vuol dire che il metodo è inefficiente quando n ha un divisore primo "grande". Quindi, ancora una volta, per evitare questo tipo di attacchi è fondamentale scegliere adeguatamente i parametri della curva.

3.2.4 Conclusioni

I metodi per la risoluzione dell'ECDLP visti nei paragrafi precedenti sono anche detti *"Attacchi generici"* in quanto possono essere utilizzati in qualsiasi circostanza a prescindere dalle proprietà della curva in oggetto. Riportiamo in seguito una tabella riassuntiva dei costi computazionali dei metodi trattati.

Metodo	Costo computazionale	Versione Migliorata
Baby Step, Giant Step	$O(\sqrt{n})$	//
ρ di Pollard	$O(\sqrt{\frac{\pi n}{2}})$	$O(\frac{1}{T}\sqrt{\frac{\pi n}{2}})$
λ di Pollard	$O(\sqrt{\frac{\pi n}{2}})$	$O(\frac{1}{T}\sqrt{\frac{\pi n}{2}})$
Pohling-Hellman	$O(\sqrt{p'})$	//

Tabella 3.1: Tabella riassuntiva "attacchi generici"

É chiaro che il metodo più efficiente, tra quelli trattati, è il metodo ρ di Pollard nella sua versione migliorata. Infatti, nonostante il metodo λ abbia la stessa complessità temporale, bisogna ricordarsi che in tal caso è necessario conoscere un intervallo nel quale cercare k (circostanza piuttosto rara). Anche il metodo di Pohling-Hellman può essere efficiente quando l'ordine del sottogruppo n è dato dal prodotto di numeri primi "piccoli". Altre tipologie di attacchi, detti *Attacchi specializzati*, sfruttano invece proprietà delle curve ellittiche per semplificare la risoluzione dell'ECDLP. In particolare si possono citare tre casi:

1. *Curve supersingolari*, cioè quando $\#E(\mathbb{F}_p) = p + 1$. In tal caso l'ECDLP può essere ridotto alla risoluzione del DLP sul gruppo moltiplicativo del campo finito $\mathbb{F}_{p^k}$ [15].

2. *Curve anomale*, cioè quando $\#E(\mathbb{F}_p) = p$. In tal caso è stato dimostrato che è possibile realizzare un isomorfismo tra $E(\mathbb{F}_p)$ e il gruppo additivo di

$\mathbb{F}_p$. Questo permette di avere diversi algoritmi, ideati da *Semaev*, *Smart* e *Satoh-Araki*, che risolvono l'ECDLP su $E(\mathbb{F}_p)$ in tempo polinomiale [25].

3. Se $\#E(\mathbb{F}_p)$ è divisibile soltanto da numeri primi "piccoli". Allora si può utilizzare il metodo di Pohlig-Hellman che, come sappiamo, risolve il problema con complessità $O(\sqrt{p'})$, dove p' è il piu grande divisore primo di $\#E(\mathbb{F}_p)$ [23].

É da notare, tuttavia, che questi attacchi possono essere facilmente elusi in fase di costruzione del sistema crittografico, andando a verificare che la curva prescelta non rispetti alcuna delle condizioni richieste per la loro applicabilità. Quindi notevole importanza assumono i metodi per la generazione di curve sicure. Il più semplice di questi metodi consiste nel generare delle curve in modo casuale e poi verificare, con appositi test, che posseggano determinate proprietà. A tal proposito mostriamo un algoritmo, tratto da [15], che consente di generare una curva in modo casuale su un campo finito $\mathbb{F}_p$.

Algorithm 8 Algoritmo per la generazione di una curva random su $\mathbb{F}_p$

INPUT: un numero primo $p > 3$ e una funzione hash a l bit H.

OUTPUT: Un *seed* S e i coefficienti a,$b \in \mathbb{F}_p$

$t = \lfloor \log_2 p \rfloor$, $s = \lfloor (t-1)/l \rfloor$, $v = t - sl$

si sceglie una stringa arbitraria S di lunghezza $g \geq l$ bit

si calcola $h = H(S)$, e si pone r_0 uguale ai v bit meno significativi di h

si pone R_0 uguale a r_0 ma con bit più significativo posto a 0

si pone z uguale all'intero la cui rappresentazione binaria è data da S

for $i = 1$ *to* s **do**

  sia s_i la rappresentazione binaria dell'intero dato da $(z + i)$ *mod* 2^g

  si calcola $R_i = H(s_i)$

end for

$R = R_0||R_1||\ldots||R_s$ dove con "||" si intende la concatenazione

si pone r uguale all'intero la cui rappresentazione binaria è R

se $r = 0$ *or* $4r + 27 \equiv 0$ *mod* p si torna alla scelta di S

si scelgono a, $b \in \mathbb{F}_p$ non entrambi uguali a 0, tali che $r \cdot b^2 \equiv a^3$ *mod* p

return (S, a, b)

Supponendo ora di aver correttamente generato una curva E sul campo finito $\mathbb{F}_p$, bisogna verificare che la curva sia sicura attraverso alcuni test. Ad esempio, per escludere la vulnerabilità della curva agli attacchi sopra menzionati, si dovrebbe innanzitutto calcolare $\#E(\mathbb{F}_p)$ e poi verificare che sia diverso da p, $p+1$ o divisibile per numeri primi piccoli. Se questi test non hanno esito positivo, si provvede alla generazione di una nuova curva.

3.2.5 Attacchi Side-Channel

Gli *Attacchi a canale laterale*, o in inglese *Side-Channel*, sono attacchi che sfruttano le informazioni non intenzionalmente trapelate da un dispositivo apparentemente resistente alla manomissione. In particolare, queste informazioni vengono spesso ottenute attraverso l'analisi del consumo di energia e/o del tempo di esecuzione dell'algoritmo. Come abbiamo visto, nella crittografia basata sulle curve ellittiche, un'operazione fondamentale è quella della moltiplicazione di un punto P della curva per un intero k. Ebbene, alcune implementazioni "banali" della moltiplicazione scalare rivelano, ad esempio attraverso una differente traccia di potenza del processore, quando viene effettuata una *Point Add* e quando invece una *Point Doubling*, permettendo così di determinare l'intero k, cioè la chiave privata.

Esempio 3.4. Consideriamo come esempio il punto $Q = 9P$ ed eseguiamo la moltiplicazione scalare con la *Double & Add* vista al paragrafo 2.3.1. Essendo la rappresentazione binaria di $9 = (1001)_2$, l'algoritmo genera la sequenza di operazioni $DADDDA$, dove "A" sta per *Point Add* e "D" per *Point Doubling*. La sequenza DA corrisponde alla cifra binaria "1" e la sola D corrisponde alla cifra binaria "0". Quindi un malintenzionato avendo a disposizione la traccia di potenza del processore potrebbe riconoscere la sequenza delle operazioni e dedurre il valore di $k = 9$.

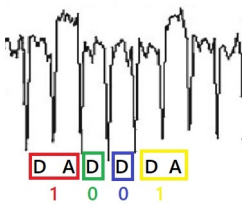

Figura 3.4: Traccia di potenza di una sequenza di Point Add e Point Doubling

Peraltro, gli attacchi a canale laterale non riguardano soltanto la *Double & Add* ma anche altre implementazioni della moltiplicazione scalare. In particolare, in alcuni casi, l'analisi della traccia di potenza e/o del tempo di esecuzione dell'algoritmo non permettono di dedurre direttamente il valore dello scalare, ma comunque di ridurre considerevolmente l'intervallo nel quale cercarlo [26].
Fortunatamente esistono diverse contromisure a questo tipo di attacchi. In generale, esse puntano a rendere indipendente la traccia di potenza dallo scalare, ma possiamo distinguerle in tre categorie:

1. contromisure che inseriscono in modo casuale delle *Point Add* così da inficiare la traccia di potenza;

2. contromisure che rendono univoco il passo base della moltiplicazione (Es. una *Point Add* e una *Point Doubling* per ogni cifra binaria) così che la traccia di potenza sia uguale per ogni cifra binaria sia se posta a 0 che a 1;

3. contromisure che eliminano le differenze nella traccia di potenza tra la *Point Add* e la *Point Doubling*.

Una delle contromisure che rientra nella seconda categoria è la *Montgomery Ladder* vista al paragrafo 2.3.2. Infatti in tale implementazione della moltiplicazione scalare per ogni cifra binaria viene eseguita sia una *Point Add* che una *Point Doubling* rendendo impossibile ad un attacker riconoscere dalla traccia di potenza lo scalare.

Capitolo 4

Applicazione Java

In questo capitolo saranno approfondite tutte le fasi che hanno portato alla realizzazione dell'applicazione Java che implementa il protocollo ECDH. Per quanto riguarda la parte teorica relativa al protocollo si faccia riferimento al capitolo 3 e in particolare al paragrafo 3.1.5.

4.1 Fasi dello sviluppo software

Il processo di sviluppo di software si compone solitamente di quattro fasi:

1. *Analisi*: fase nella quale sono definiti con precisione i compiti che devono essere eseguiti dal sistema software.

2. *Progettazione*: in questa fase vengono identificate le singole classi e le rispettive responsabilità. Vengono inoltre determinate le relazioni che legano ciascuna classe alle altre. Il risultato di questa fase comprende quindi: una descrizione testuale delle classi e delle loro principali responsabilità, diagrammi di relazione fra le classi, diagrammi di sequenza.

3. *Implementazione*: fase nella quale si scrive il codice dell'applicazione in base alla struttura e alle caratteristiche definite nella fase di progettazione.

4. *Test*: fase nella quale si verifica il corretto funzionamento dell'applicazione.

4.2 Analisi

4.2.1 Scopo del sistema software

Lo scopo del sistema software è quello di consentire a due processi distinti, che chiameremo rispettivamente *Client* e *Server*, di stabilire, secondo il protocollo ECDH, una chiave segreta condivisa. Più precisamente i due processi, che possono essere in esecuzione sulla stessa macchina o meno, devono poter generare ciascuno una coppia di chiavi, una pubblica ed una privata, e poi attraverso queste generare la chiave segreta condivisa.

4.2.2 Requisiti

- il sistema deve poter gestire delle curve ellittiche, nonché i singoli punti di una curva ellittica e quindi eseguire su questi ultimi operazioni come la *Point Add* e la *Point Multiplication*.
- il sistema deve permettere la generazione di una chiave pubblica ed una privata, nonché di una chiave segreta condivisa in base al protocollo ECDH.
- il sistema deve comporsi di due processi distinti: un *Server* che si pone in attesa su una porta TCP prestabilita della connessione del *Client* che, invece, effettua la richiesta di connessione.
- il sistema deve permettere lo scambio tra i due processi delle rispettive chiavi pubbliche attraverso la rete.
- il sistema deve permettere la validazione della chiave pubblica ricevuta da un altro processo (per il concetto di validazione della chiave pubblica si veda il paragrafo 3.1.3).
- il sistema deve permettere la connessione di più processi *Client* allo stesso processo *Server*.

4.3 Progettazione

4.3.1 Individuazione delle classi e delle responsabilità

Una regola molto semplice per l'individuazione delle classi consiste nel prendere in esame i *sostantivi* utilizzati nella descrizione del problema nella fase di analisi. In base a questo approccio le classi individuate sono:

Classe	**Responsabilità**
Curva	definire la rappresentazione della curva nel sistema e i metodi per la somma di due punti e di moltiplicazione scalare tra un punto della curva e un intero.
Punto	definire la rappresentazione di un punto di una curva ellittica
ECDH	definire i metodi per la generazione della chiave pubblica, della chiave privata e della chiave segreta condivisa, nonché del metodo per la validazione delle chiavi pubbliche.
Client	definire i metodi per effettuare la richiesta di connessione e per effettuare lo scambio della chiave pubblica con il server.
Server	definire i metodi per porre il server in ascolto su una certa porta TCP in attesa della connessione dei client e per effettuare lo scambio della chiave pubblica con il client.

4.3.2 Diagrammi UML

L'*Unified Modeling Language* è un linguaggio di modellizzazione basato sul paradigma orientato agli oggetti, definito nel 1996 da *Grady Booch*, *Jim Rumbaugh* e *Ivar Jacobson*. Esistono molte tipologie di diagrammi UML. Nell'ambito della nostra progettazione sono stati utilizzati due tipi di diagrammi UML:

- *Diagramma di classe*: mostra le classi e le relazioni esistenti tra di esse. Ogni classe è rappresentata da un "box" contente nome, attributi e metodi della classe (non serve indicarli tutti, ma solo i più importanti). Le relazioni tra le classi possono essere di diversa tipologia e sono esplicitate da specifici collegamenti. Alcuni tipi di relazione tra classi sono:

- la *Dipendenza*: una classe dipende da un'altra classe se manipola oggetti appartenti a quest'ultima. Peraltro, di norma, è più facile capire quando una classe non dipende da un'altra, ovvero quando riesce ad eseguire tutte le sue attività senza essere consapevole che l'altra classe esiste [27]. Simbolo: freccia tratteggiata (4.1 a).
- L'*Aggregazione*: una classe aggrega un'altra classe se gli oggetti della prima contengono oggetti della seconda. La relazione di aggregazione è tuttavia una relazione debole, cioè l'oggetto aggregato potrebbe in alcuni casi esistere indipendentemente dalle parti e le parti possono esistere indipendentemente dall'oggetto aggregato e addirittura possono far parte di più aggregati. Simbolo: linea terminata da un rombo vuoto (4.1 b).
- La *Composizione*: è una relazione più forte dell'aggregazione, in quanto l'esistenza del contenuto non è indipendente dal contenitore. Ciò comporta che ogni componente può far parte di un solo oggetto composto ed è legato al ciclo di vita di quest'ultimo: se viene distrutto il composto, vengono distrutti tutti i suoi componenti. Simbolo: linea terminata da un rombo pieno (4.1 c).

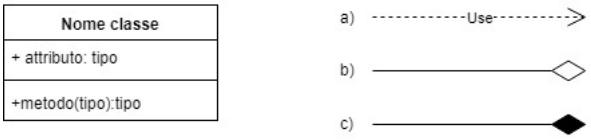

Figura 4.1: Simboli diagramma delle classi

- *Diagramma di sequenza*: descrive le interazioni tra gli oggetti nell'ambito di uno specifico scenario o caso d'uso, enfatizzando l'ordinamento temporale delle chiamate dei metodi. Ogni oggetto è rappresentato da un rettangolo contenente, di norma, nome dell'oggetto e classe dell'oggetto. Da ogni oggetto parte una linea tratteggiata, detta *"linea della vita"*, che rappresenta la vita dell'oggetto stesso. Lungo le linee di vita possono essere presenti delle *barre di attivazione* che mostrano quando l'oggetto ha il controllo, eseguendo un suo metodo. I

messaggi scambiati tra gli oggetti sono invece rappresentati da diversi tipi di frecce a seconda del tipo di messaggio:

- *Messaggio sincrono*: il mittente attende una risposta prima di continuare. Simbolo: freccia continua piena (4.2 a).
- *Messaggio di ritorno*: risposta al mittente di un messaggio sincrono. Simbolo: freccia tratteggiata (4.2 b).
- *Messaggio asincrono*: il mittente non richiede alcuna risposta prima di continuare. Sono ad esempio utilizzati per la creazione di nuovi thread. Simbolo: freccia continua aperta (4.2 c).
- *Messaggio di creazione*: crea un'istanza di una classe. Simbolo: lo stesso del messaggio sincrono, la differenza è che in tal caso troviamo il box di oggetto al termine della freccia (4.2 a).
- *Messaggio trovato*: è un messaggio asincrono da un oggetto sconosciuto o non specificato. Simbolo: freccia con all'inizio un tondo pieno (4.2 d).

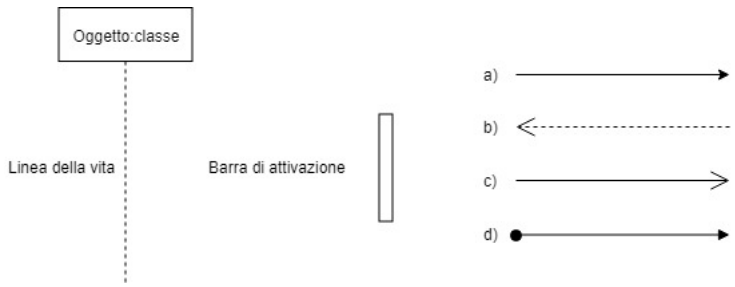

Figura 4.2: Simboli diagramma di sequenza

Dopo questa breve introduzione sui diagrammi UML, andiamo ad analizzare quelli realizzati nello sviluppo del sistema software.

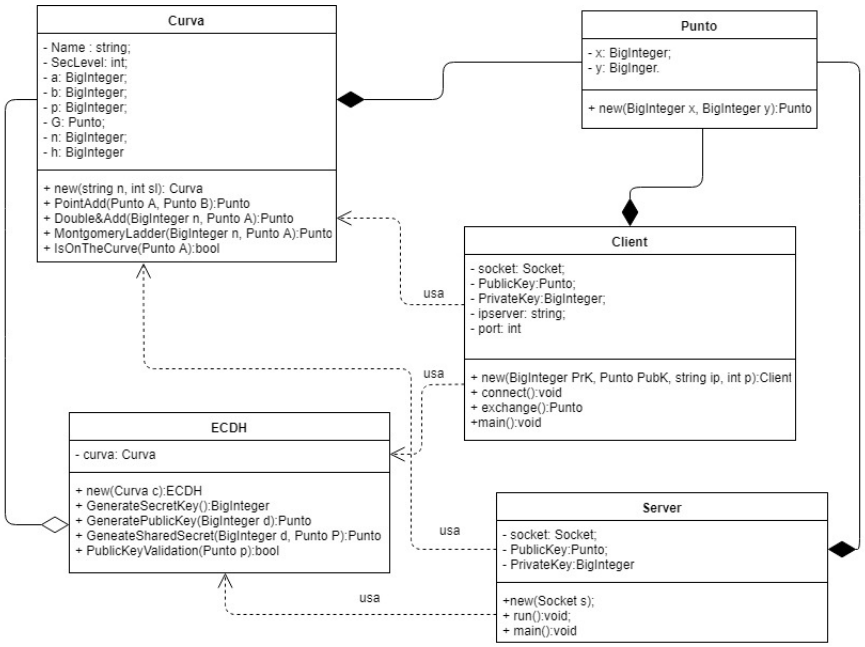

Figura 4.3: Diagramma delle classi

Il primo che vediamo è il diagramma delle classi riportato in figura 4.3. Come si vede sono presenti le classi rilevate in fase di analisi e in più sono state esplicitate le relazioni tra le stesse. In particolare:

- ***Curva - Punto***: nella rappresentazione realizzata per una curva ellittica, cioè mediante i *parametri di dominio* visti nel paragrafo 3.1.1, il punto G è il generatore del sottogruppo. La relazione che lega le due classi è quella di composizione, ciò comporta che un'istanza della classe punto potrà essere parte di una sola curva alla volta.

- ***Client/Server-Punto***: per il client/server la chiave pubblica è data da un punto di una curva ellittica, ottenuto attraverso la moltiplicazione scalare tra la propria chiave privata e il generatore del sottogruppo. É quindi ovvio il motivo della relazione di composizione, un'istanza della classe punto può essere la chiave pubblica di un solo client/server alla volta e la distruzione del client/server deve comportare la distruzione della rispettiva chiave pubblica.

- ***Curva - ECDH***: nella classe *ECDH* è presente un attributo della classe *Curva* che rappresenta la curva da utilizzare per la generazione delle chiavi. É

chiaro che ogni curva può essere utilizzata da più istanze della classe *ECDH* come base del protocollo; appunto per questo la relazione che lega le due classi è quella di aggregazione.

- ***Client/Server - Curva***: l'applicazione offre la possibilità di scegliere tra una molteplicità di curve ellittiche, che si differenziano per l'ente che ha standardizzato la curva e per il livello di sicurezza offerto. Partendo dal presupposto che client e server siano già d'accordo sulla curva da utilizzare, i due "interlocutori" creeranno un'istanza della classe *Curva*, per selezionare la specifica curva da utilizzare, e la daranno in input al costruttore della classe *ECDH*. Ciò spiega la relazione di dipendenza tra le due classi.

- ***Client/Server - ECDH***: la relazione di dipendenza tra le due classi è dovuta al fatto che i due "interlocutori" nei rispettivi metodi utilizzano oggetti e metodi della classe *ECDH* per la generazione delle rispettive chiavi pubbliche, private e segrete condivise, nonché per la validazione della chiave pubblica ricevuta l'uno dall'altro.

Dopo il diagramma delle classi sono stati realizzati due diagrammi di sequenza:

- il primo diagramma di sequenza, riportato in figura 4.4, mostra cosa avviene "lato server" quando viene stabilita la connessione tra un client e il server che è in ascolto su una specifica porta TCP. Come si vede, ogni qualvolta un client si connette, il server crea un nuovo thread per la gestione della connessione con il client. All'interno di ciascun thread vengono quindi

  - create la chiave pubblica e privata del server per quella connessione;
  - avviene lo scambio della chiave pubblica con il client;
  - viene generata la chiave segreta condivisa dal server con il client.

  É da notare l'architettura multithread del server che consente di gestire, come previsto in fase di analisi, più client "contemporaneamente". Infatti nel caso in cui il server fosse eseguito come un processo a singolo thread, esso sarebbe in grado di soddisfare un solo client alla volta e ciascuno di essi potrebbe dover aspettare molto tempo prima che la propria richiesta venga servita.

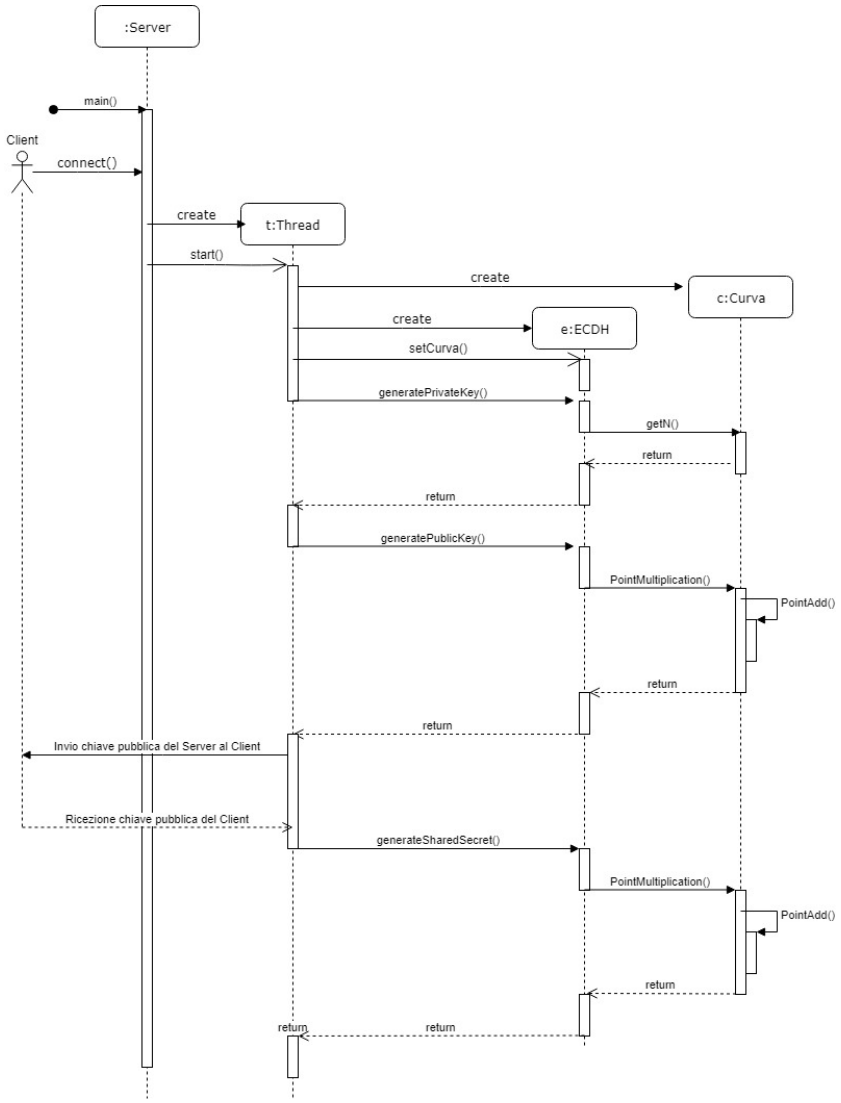

Figura 4.4: Diagramma di sequenza Server

- Nel secondo diagramma di sequenza, riportato in figura 4.5, viene mostrato l'analogo del precedente diagramma ma questa volta "lato client". Come si vede, dopo la generazione della coppia di chiavi, il client prova a stabilire una connessione con il server. Una volta connesso avviene lo scambio delle chiavi pubbliche e quindi la generazione delle chiave segreta condivisa.

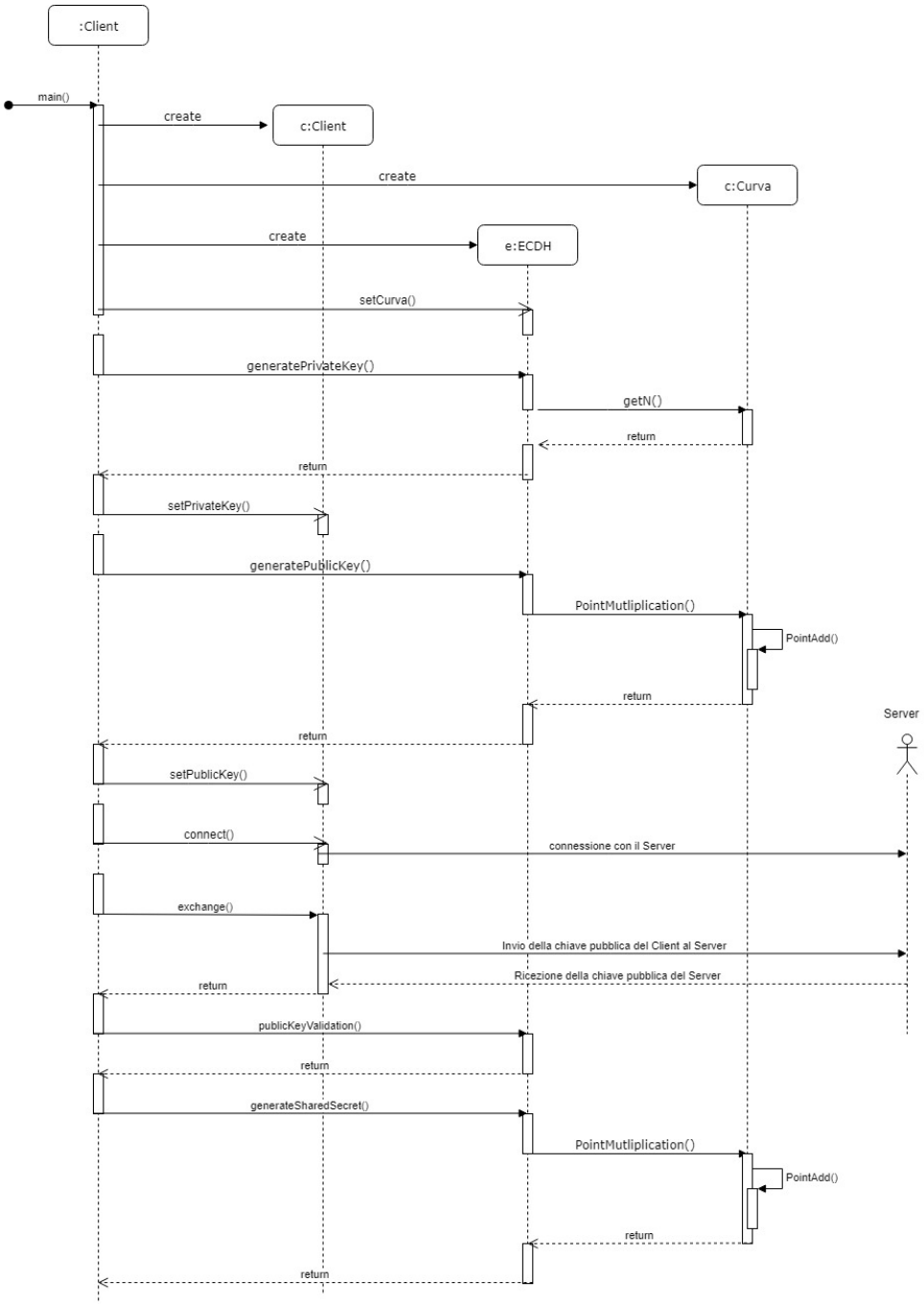

Figura 4.5: Diagramma di sequenza Client

4.4 Implementazione

In questa sezione viene analizzato il codice delle classi implementate in Java. Per ciascuna classe è riportato il codice relativo alla definizione degli attributi nonché dei principali metodi. Per ogni frammento di codice è presente una breve descrizione iniziale, nonché dei commenti inseriti in fase di scrittura del codice stesso per renderlo più comprensibile.

4.4.1 Classe *ECPoint*

Per l'implementazione in Java della classe *Punto* si è deciso di utilizzare la classe *ECPoint*, del Package *java.security.spec*, che rappresenta un punto di una curva ellittica. Il costruttore della classe riceve come argomenti due parametri appartenti alla classe *BigInteger* che rappresentano la coordinata x e la coordinata y del punto. É presente un attributo di classe POINT_INFINITY che definisce il punto all'infinito. La classe prevede inoltre due metodi di nostro particolare interesse:

- *getAffineX()*: che restituisce un oggetto della classe *BigInteger* che rappresenta la coordinata x del punto;
- *getAffineY()*: che restituisce un oggetto della classe *BigInteger* che rappresenta la coordinata y del punto.

4.4.2 Classe *Curva*

Nella porzione di codice che segue è presente la definizione degli attributi della classe.

```
public class Curva {

    private String Name; //nome della curva
    private int SecLevel; // livello di sicurezza
    private BigInteger a; // parametro a dell'equazione della curva
    private BigInteger b; // parametro b dell'equazione della curva
    private BigInteger p; // cardinalita' del campo Fp
    private ECPoint G; // generatore del sottogruppo
    private BigInteger n; // cardinalita' del sottogruppo
```

```
private int h; // cofattore del sottogruppo
```

Il costruttore della classe curva, in base al nome dell'agenzia e al livello di sicurezza ricevuti come argomenti, richiama lo specifico metodo che imposta i parametri della curva scelta (un esempio è il metodo successivo).

```
public Curva(String n, int sl) {
    if (n == "NIST") {
        if (sl == 112) {
            this.NISTP224();
        } else if (sl == 128) {
            this.NISTP256();
```

Il metodo seguente è un esempio di quelli che vengono richiamati ogni qualvolta viene invocato il costruttore della classe *Curva*. In tali metodi sono stati inseriti i parametri delle curve consigliate dal *NIST* nel *FIPS 186-2*.

```
public void NISTP256() {
    this.setName("NIST P256");
    this.setSecLevel(128);
    this.setA(BigInteger.valueOf(-3));
    this.setB(new BigInteger("410583637...");
    BigInteger Xg= new BigInteger("484395612...");
    BigInteger Yg= new BigInteger("361342509...");
    this.setG(new ECPoint(Xg,Yg));
    this.setP(new BigInteger("11579208921...");
    this.setN(new BigInteger("11579208921...");
    this.setH(1);
}
```

Il metodo ***IsOnTheCurve()*** dato un punto A determina se quest'ultimo si trova sulla curva verificando, prima se si tratta del punto all'infinito, in quanto in tal caso apparterebbe sicuramente alla curva e quindi si potrebbe immediatamente restituire *true*, poi se il punto soddisfa l'equazione della curva.

```
public Boolean IsOnTheCurve(ECPoint A) {
```

```
        if (A.equals(ECPoint.POINT_INFINITY)) { // se A èil punto
            all'infinito appartiene sicuramente alla curva per sua stessa
            definizione
            return true;
        } else {
            BigInteger x = A.getAffineX();
            BigInteger y = A.getAffineY(); //ricorda: y^2=x^3+ax+b (mod p)
                --> Eq. Weierstrass curva ellittica
            BigInteger leftmember = (y.pow(2)).mod(this.p); //y^2 (mod p)
            BigInteger rightmember =
                (((x.pow(3)).add((this.a).multiply(x))).add(this.b)).mod(this.p);
                //x^3+ax+b (mod p)
            if (leftmember.equals(rightmember)) { //si verifica
                l'eguaglianza tra membro destro e membro sinistro
                dell'equazione
                return true;
            } else {
                return false;
            }
        }
    }
```

Il metodo ***PointAdd()*** dati due punti A e B esegue la *Point Addition* tra di essi, restituendo un terzo punto. In particolare, viene prima verificata l'appartenenza dei due punti A e B alla curva, e nel caso non vi appartengano viene generata un'eccezione, e poi vengono applicate le formule viste nei paragrafi 2.2 e 2.4 , riconoscendo attraverso appositi controlli lo specifico caso in cui ci si trova (Es. $A = B$, A *o* $B = \mathcal{O}$ ecc.).

```
    public ECPoint PointAdd(ECPoint A, ECPoint B) throws
        IsntOnTheCurveException {
        BigInteger Xc; // coordinata x del punto risultante dalla somma di
            A e B
        BigInteger Yc; // coordinata y del punto risultante dalla somma di
            A e B
```

```
BigInteger Xa = A.getAffineX(); // coordinata x del punto A
BigInteger Ya = A.getAffineY(); // coordinata y del punto A
BigInteger Xb = B.getAffineX(); // coordinata x del punto B
BigInteger Yb = B.getAffineY(); // coordinata y del punto B
BigInteger m;
try {
    if (this.IsOnTheCurve(A) && this.IsOnTheCurve(B)) {
        if (A.equals(ECPoint.POINT_INFINITY)) { // Verifico se A èil
            punto all'infinito
            return B; // in tal caso B+0=B
        } else if (B.equals(ECPoint.POINT_INFINITY)) { // Verifico
            se B èil punto all'infinito
            return A; // in tal caso A+0=A
        } else if (((Xa.mod(this.p)).equals(Xb.mod(this.p)))
                &&
                    ((((Ya.negate())).mod(this.p)).equals(Yb.mod(this.p))))
                    { // Verifico A e B sono simmetrici
            return ECPoint.POINT_INFINITY; // in tal caso A+(-A)=0
        } else if (A.equals(B)) { // Verifico se i due punti sono
            uguali
            // in tal caso il coefficiente angolare m èdato da
                m=((3Xp^2+a)/(2Yp))(mod p)
            // -> Point Double
            m = (((BigInteger.valueOf(3).multiply(Xa.pow(2)))
                .add(this.a)).multiply((BigInteger.valueOf(2)
                .multiply(Ya)).modInverse(this.p))).mod(this.p);
            Xc = (m.pow(2).subtract(Xa.add(Xa))).mod(this.p);
            Yc =
                (((Xa.subtract(Xc)).multiply(m)).subtract(Ya)).mod(this.p);
            return new ECPoint(Xc, Yc);
        } else { // ci troviamo nel caso P!=Q
            // in tal caso il coefficiente angola m èdato da
                m=((Ya-Yb)/(Xa-Xb))(mod p)
            m = ((Ya.subtract(Yb))
                .multiply((Xa.subtract(Xb)).modInverse(this.p))).mod(this.p);
```

```
                Xc =
                    (((m.pow(2)).subtract(Xa)).subtract(Xb)).mod(this.p);
                Yc =
                    (((Xa.subtract(Xc)).multiply(m)).subtract(Ya)).mod(this.p);
                return new ECPoint(Xc, Yc);
            }
        } else {
            throw new IsntOnTheCurveException("Uno o entrambi i punti
                non sono sulla curva");
        }
    } catch (IsntOnTheCurveException ex) {
        System.out.println(ex);
        return null;
    }
```

Il metodo ***DoubleAndAdd()*** implementa l'algoritmo visto nel paragrafo 2.3.1. Quindi dati uno scalare e un punto il metodo restituisce il punto risultante dalla moltiplicazione scalare. Tuttavia, per i motivi menzionati nel paragrafo 3.2.5, non sarà utilizzato questo metodo per la *Point Multiplication*, bensì quello che implementa l'algoritmo *Montgomery Ladder*.

```
public ECPoint DoubleAndAdd(BigInteger n, ECPoint P) throws
    IsntOnTheCurveException {
    try {
        String nbinary = n.toString(2); // traduco lo scalare in una
            stringa binaria
        ECPoint result = P;
        for (int i = 1; i < (nbinary.length()); i++) { // a partire
            dalla seconda cifra binaria più significativa
            // fino alla meno significativa
            result = this.PointAdd(result, result); // faccio una Point
                Double
            if (nbinary.charAt(i) == '1') // e se la cifra binaria
                i-esima èuguale a 1
            {
```

```
                result = this.PointAdd(result, P); // faccio una Point
                    Add
            }
        }
        return result;
    } catch (IsntOnTheCurveException ex) {
        System.out.println(ex);
        return null;
    }
}
```

Il metodo ***MontgomeryLadder()*** implementa l'algoritmo visto nel paragrafo 2.3.2. Come già detto più volte, l'esecuzione della *Point Multiplication* mediante tale algoritmo garantisce la resistenza ad alcuni attacchi a canale laterale (per maggiori dettagli si veda 3.2.5).

```
public ECPoint MontgomeryLadder(BigInteger n, ECPoint P) throws
    IsntOnTheCurveException {
    try {
        ECPoint P1 = P; // si pone P1=P
        ECPoint P2 = this.PointAdd(P, P); // e P2=2P
        String nbinary = n.toString(2); // traduco lo scalare in una
            stringa binaria
        for (int i = 1; i < (nbinary.length()); i++) { // a partire
            dalla seconda cifra più significativa fino alla
            // cifra meno significativa
            if (nbinary.charAt(i) == '1') { // se la cifra binaria
                i-esima èuguale a 1
                P1 = this.PointAdd(P1, P2); // P1=P1+P2
                P2 = this.PointAdd(P2, P2); // P2=2P2
            } else { // se la cifra binaria i-esima èuguale a 0
                P2 = this.PointAdd(P1, P2); // P2=P1+P2
                P1 = this.PointAdd(P1, P1); // P1=2P1
            }
        }
```

```
            return P1; // P1 èil risultato
        } catch (IsntOnTheCurveException ex) {
            System.out.println(ex);
            return null;
        }
    }
```

4.4.3 Classe *ECDH*

Nella porzione di codice che segue è presente la definizione degli attributi della classe.

```
public class ECDH {

    private Curva curve; //curva sulla quale effettuare le operazioni.

    public ECDH() {}
```

Il metodo ***GeneratePrivateKey()*** è invocato per la generazione di una chiave privata. Nel dettaglio, il metodo prevede un ciclo al cui interno vengono generati degli interi random compresi nell'intervallo $[0, 2^{\text{bitlenght}} - 1]$, dove *bitlenght* è il numero di bit dell'ordine del sottogruppo n, finché non viene trovato un intero nell'intervallo $[1, n-1]$.

```
    public BigInteger GeneratePrivateKey() {
        Random rnd = new Random();
        BigInteger n = this.curve.getN();
        int bitlength = (n.toString(2)).length();
        BigInteger d;
        do {
            d = new BigInteger(bitlength, rnd);
        } while (d.compareTo(n) >= 0 || d.compareTo(BigInteger.ZERO) == 0);
        return d;
    }
```

Il metodo ***GeneratePublicKey()*** è invocato per la generazione di una chiave pubblica. In particolare, dato in input un intero d (chiave privata di chi invoca il metodo) esegue la *Point Multiplication*, con l'algoritmo *Montgomery Ladder*, tra il generatore del sottogruppo della curva G e l'intero d.

```
public ECPoint GeneratePublicKey(BigInteger d) throws
    Curva.IsntOnTheCurveException {
    ECPoint P = this.curve.MontgomeryLadder(d, this.curve.getG());
    return P;
}
```

Il metodo ***GenerateSharedSecret()*** è invocato per la generazione della chiave segreta condivisa. Fondamentalmente è analogo al metodo precedente, con la differenza che in tal caso la *Point Multiplication* è eseguita tra l'intero (chiave privata di chi invoca il metodo) e un punto dato anch'esso in input (chiave pubblica dell'altro interlocutore).

```
public ECPoint GenerateSharedSecret(BigInteger d, ECPoint OtherPubKey)
    throws Curva.IsntOnTheCurveException {
    ECPoint S = this.curve.MontgomeryLadder(d, OtherPubKey);
    return S;
}
```

Il metodo ***PublicKeyValidation()*** implementa l'algoritmo per la validazione di una chiave pubblica visto nel paragrafo 3.1.3. Come si vede, vengono effettuati tre controlli: che la chiave sia diversa dal punto all'infinito, che le coordinate del punto appartengano all'intervallo $[0, p-1]$, con p ordine del campo $\mathbb{F}_p$ sul quale è definita la curva, e che il punto appartenga alla curva.

```
public boolean PublicKeyValidation(ECPoint pubkey){
    BigInteger x=pubkey.getAffineX();
    BigInteger y=pubkey.getAffineY();
    BigInteger p=this.curve.getP();
    if(pubkey==ECPoint.POINT_INFINITY){ //si verifica che la chiave
        pubblica sia diversa dal punto all'infinito
        return false;
```

```
        }
        else if(!((x.compareTo(BigInteger.valueOf(-1))==1)&& // si verifica
            che le coordinate x e y della chiave pubblica appartengano al
            campo finito Fp
                (y.compareTo(BigInteger.valueOf(-1))==1) &&
                (x.compareTo(p)==-1) &&
                (y.compareTo(p)==-1))){
                    return false;
                }
        else if (!curve.IsOnTheCurve(pubkey)){ // si verifica che la chiave
            pubblica appartenga alla curva
            return false;
        }
        else return true;
    }
```

4.4.4 Classe *Client*

Nella porzione di codice che segue è presente la definizione degli attributi della classe.

```
public class Client {

    private Socket socket;
    private int port; //numero di porta del server
    private String ipserver; //indirizzo ip del server
    private BigInteger ClientPrivateKey; //chiave privata del client
    private ECPoint ClientPublicKey; //chiave pubblica del client
```

Il metodo ***connect()*** viene utilizzato per stabilire una connessione con il server che è in ascolto ad un certo indirizzo ip e ad un certa porta. Ciò viene realizzato creando una nuova istanza della classe *Socket* e passando come parametri al costruttore l'indirizzo ip del server e il numero di porta.

```
    public void Connect() throws UnknownHostException {
        try {
```

```
            System.out.println("Connessione al server in corso...");
            this.socket = new Socket(this.ipserver, this.port);
            System.out.println("Connessione avvenuta!");
        } catch (IOException ex) {
            Logger.getLogger(Client.class.getName()).log(Level.SEVERE,
                null, ex);
        }
    }
```

Il metodo ***Exchange()*** viene invece utilizzato per la trasmissione della chiave pubblica al server e per la ricezione della chiave pubblica dal server. L'apertura del canale di comunicazione con il server per lo scambio dei messaggi tramite array di byte, avviene mediante la creazione di istanze delle classi *DataInputStream* e *DataOutputStream* e dando in input ai rispettivi costruttori il risultato dei metodi *getInputStream()* e *getOutputStream* della classe *Socket*. Peraltro, è da notare che per il corretto funzionamento degli stream è necessario che siano "aperti" con ordine opposto nel client e nel server: se nel client viene creato prima il canale di input, nel server va creato prima quello di output e viceversa. Al termine della trasmissione vengono "chiusi" i canali di input e output e la connessione con il server. Ovviamente, nel caso di una reale comunicazione tra i due interlocutori, ciò di norma non accadebbre, poiché sarebbe proprio da questo momento che potrebbero iniziare a comunicare in "totale" sicurezza utilizzando la chiave appena calcolata come chiave di cifratura dei messaggi.

```
    public ECPoint Exchange() {
        try {
            DataInputStream in = new
                DataInputStream(this.socket.getInputStream());
            DataOutputStream out = new
                DataOutputStream(this.socket.getOutputStream());
            byte[] Xbytes =
                this.ClientPublicKey.getAffineX().toByteArray(); //la
                coordinata X della chiave pubblica viene tradotta in un
                array di byte
```

```
byte[] Ybytes =
    this.ClientPublicKey.getAffineY().toByteArray(); //la
    coordinata Y della chiave pubblica viene tradotta in un
    array di byte
int length = Xbytes.length + Ybytes.length; //somma della
    lunghezza dei due array
out.writeInt(length); // la lunghezza viene inviata al server
out.writeInt(Xbytes.length); // la lunghezza dell'array di byte
    della coordinata X viene inviata al server
// --> in questo modo sottraendo alla lunghezza totale quella
    della X il server èin grado
//di determinare la lunghezza della Y
byte[] XYbytes = new byte[length]; // vengono concatenati i due
    array
System.arraycopy(Xbytes, 0, XYbytes, 0, Xbytes.length);
System.arraycopy(Ybytes, 0, XYbytes, Xbytes.length,
    Ybytes.length);
out.write(XYbytes); // l'array risultante viene inviato al
    server
int othlength = in.readInt(); // si legge la lunghezza
    dell'array inviato dal server
int othXlength = in.readInt(); // si legge la lunghezza
    dell'array di byte della X inviato dal server
byte[] othXYbytes = new byte[othlength];
byte[] othXbytes = new byte[othXlength];
byte[] othYbytes = new byte[othlength - othXlength];
in.readFully(othXYbytes); // si legge l'array XY inviato dal
    server
System.arraycopy(othXYbytes, 0, othXbytes, 0, othXlength); //
    si scompone nei due array othXbytes
System.arraycopy(othXYbytes, othXlength, othYbytes, 0,
    othYbytes.length);// e othYbytes
ECPoint ServerPublicKey = new ECPoint(new
    BigInteger(othXbytes), new BigInteger(othYbytes)); // a
    questo punto èpossibile ottenere la chiave pubblica del
```

```
            server
        return ServerPublicKey;
    } catch (IOException ex) {
        Logger.getLogger(Client.class.getName()).log(Level.SEVERE,
            null, ex);
        return null;
    }
}
```

Il metodo ***main()*** della classe *Client* rappresenta il punto di inizio dell'esecuzione "lato Client". Come tutti i metodi *main()* in Java è dichiarato come *public* e *static*, accetta un solo argomento di tipo *String[]* e ha *void* come tipo di ritorno. Il metodo provvede:

- alla creazione di un'istanza della classe *Curva* della classe *ECDH*;
- alla creazione di un'istanza della classe *Client*, passando al costruttore come argomenti l'indirizzo ip del server e il numero di porta;
- alla generazione della chiave pubblica e di quella privata;
- allo scambio delle chiave pubblica con il server;
- alla verifica della validità della chiave pubblica ricevuta dal server;
- alla generazione della chiave segreta condivisa.

Precisiamo che il metodo *PrintKey()*, eseguito nella parte finale del *main()* che provvede alla stampa a display delle chiavi, è presente soltanto a scopo di test del sistema.

```
public static void main(String[] args) throws Exception {
    Curva c = new Curva("NIST", 128); // viene creata una nuova curva
    ECDH e = new ECDH();
    e.setCurva(c);
    Client cl = new Client("127.0.0.1", 6789); // viene creato un nuovo
        client
```

```
        BigInteger d = e.GeneratePrivateKey(); //viene generata la chiave
            privata
        ECPoint pubkey = e.GeneratePublicKey(d); // viene generata la
            chiave pubblica
        cl.setPrivateKey(d);
        cl.setPublicKey(pubkey);
        cl.Connect(); //viene effettuata la connessione al server
        ECPoint othPublicKey = cl.Exchange(); //viene effettuato lo scambio
            delle chiavi pubbliche con il server
        if (e.PublicKeyValidation(othPublicKey)) { //viene verificata la
            validitá della chiave pubblica ricevuta dal server
            ECPoint SharedSecret = e.GenerateSharedSecret(d, othPublicKey);
                // viene generata la Shared Secret
            cl.PrintKey(d, pubkey, othPublicKey, SharedSecret);// Stampa
                delle chiavi
        } else {
            System.out.println("La chiave pubblica ricevuta èinvalida!");
        }
    }
```

4.4.5 Classe *Server*

Il Server Multithread è stato realizzato implementando con la classe *Server* l'interfaccia *Runnable*. Infatti, questa interfaccia, andrebbe implementata ogni qualvolta le istanze di una classe devono essere eseguite da un thread. Peraltro l'implementazione di *Runnable* richiede che la classe definisca un metodo senza argomenti *run()* [28], che vedremo più in basso.

```
public class Server implements Runnable {

    private Socket socket;
    private ECPoint ServerPublicKey;
    private BigInteger ServerPrivateKey;
```

Il metodo ***run()*** definisce il "comportamento" del singolo thread. Nel nostro caso ogni thread provvederà:

- alla creazione di un'istanza della classe *Curva* e della classe *ECDH*;
- alla generazione della chiave pubblica e di quella privata del server per quella specifica connessione;
- allo scambio della chiave pubblica con il client;
- alla verifica della validità della chiave pubblica ricevuta dal client;
- alla generazione della chiave segreta condivisa per quella specifica connessione.

Per maggiori dettagli sull'apertura del canale di comunicazione con il client si veda il metodo *exchange()* della classe *Client*.

```
public void run() {
    try {
        Curva c = new Curva("NIST", 128); // viene creata una nuova
            curva
        ECDH e = new ECDH();
        e.setCurva(c);
        this.setPrivateKey(e.GeneratePrivateKey()); //viene generata la
            chiave privata
        this.setPublicKey(e.GeneratePublicKey(this.ServerPrivateKey));
            //viene generata la chiave pubblica
        DataOutputStream out = new
            DataOutputStream(this.socket.getOutputStream());
        DataInputStream in = new
            DataInputStream(this.socket.getInputStream());
        byte[] Xbytes =
            this.ServerPublicKey.getAffineX().toByteArray(); //la
            coordinata X della chiave pubblica viene tradotta in un
            array di byte
        byte[] Ybytes =
            this.ServerPublicKey.getAffineY().toByteArray(); //la
```

```
    coordinata Y della chiave pubblica viene tradotta in un
    array di byte
int length = Xbytes.length + Ybytes.length; //somma della
    lunghezza dei due array
out.writeInt(length); // la lunghezza viene inviata al client
out.writeInt(Xbytes.length); // la lunghezza viene dell'array
    di byte della coordinata X viene inviata al client
// --> in questo modo sottraendo alla lunghezza totale quella
    della X il client p in grado
//di determinare la lunghezza della Y
byte[] XYbytes = new byte[length]; // vengono concatenati i due
    array
System.arraycopy(Xbytes, 0, XYbytes, 0, Xbytes.length);
System.arraycopy(Ybytes, 0, XYbytes, Xbytes.length,
    Ybytes.length);
out.write(XYbytes); // l'array risultante viene inviato al
    client
int othlength = in.readInt(); // si legge la lunghezza
    dell'array inviato dal client
int othXlength = in.readInt(); // si legge la lunghezza
    dell'array di byte della X inviato dal client
byte[] othXYbytes = new byte[othlength];
byte[] othXbytes = new byte[othXlength];
byte[] othYbytes = new byte[othlength - othXlength];
in.readFully(othXYbytes); // si legge l'array XY inviato dal
    server
in.close(); // si chiude lo stream in input
out.close(); // si chiude lo stream in output
this.socket.close(); // si chiude la connessione con il client
System.arraycopy(othXYbytes, 0, othXbytes, 0, othXlength); //
    si scompone nei due array othXbytes
System.arraycopy(othXYbytes, othXlength, othYbytes, 0,
    othYbytes.length); // e othYbytes
ECPoint ClientPublicKey = new ECPoint(new
    BigInteger(othXbytes), new BigInteger(othYbytes)); // a
```

```
                questo punto èpossibile ottenere la chiave pubblica del
                client
            if (e.PublicKeyValidation(ClientPublicKey)) { //viene
                verificata la validitá della chiave pubblica ricevuta dal
                client
                ECPoint SharedSecret =
                    e.GenerateSharedSecret(this.ServerPrivateKey,
                    ClientPublicKey);
                this.PrintKey(this.ServerPrivateKey, this.ServerPublicKey,
                    ClientPublicKey,
                    SharedSecret,Thread.currentThread().getName());// Stampa
                    delle chiavi
            } else {
                System.out.println("La chiave pubblica ricevuta èinvalida!");
            }
        } catch (IOException ex) {
            Logger.getLogger(Server.class.getName()).log(Level.SEVERE,
                null, ex);
        } catch (Curva.IsntOnTheCurveException ex) {
            Logger.getLogger(Server.class.getName()).log(Level.SEVERE,
                null, ex);
        }
    }
```

Il metodo ***main()*** della classe *Server* rappresenta il punto di inizio dell'esecuzione "lato Server". Come si vede viene prima creata una socket, associata alla porta specificata nel costruttore, sulla quale il server rimane in attesa di una richiesta da parte di un client. Mediante il metodo *accept()* all'interno del ciclo "infinito", infatti, il server viene posto in attesa della connessione di un client. Quando un client si connette, viene creata una nuova istanza della classe *Server*, che ricordiamo implementa l'interfaccia *Runnable*, e viene passata come argomento al costruttore della classe *Thread*. Attraverso il metodo *start()* della classe *Thread* infine si invoca il metodo *run()* definito precedentemente.

```
    public static void main(String[] args) throws Exception {
```

```
        System.out.println("Server avviato! In attesa di client!");
        ServerSocket ss = new ServerSocket(6789); // viene avviato il
            server sulla porta indicata
        int i=1;
        while (true) { // viene fatto ciclare all'infinito
            Socket s = ss.accept(); // il server viene posto in attesa
                della connessione di un client
            System.out.println("Nuovo client connesso!");
            Server slave = new Server(s);
            Thread t = new Thread(slave,"Client"+i); //alla connessione di
                un client si crea un nuovo thread
            t.start();
            i++;
        }
    }
```

4.5 Test

La fase di "*testing*", come già detto, consiste nell'eseguire il sistema software realizzato e verificare se il suo comportamento rispecchia i requisiti preposti. Nel nostro caso il corretto funzionamento del sistema è sinomino del fatto che le chiavi segrete condivise generate dai due processi comunicanti siano uguali. Per verificare tale condizione è stata realizzata una classe ***Test*** contenente dei metodi che permettono di determinare, su un numero prefissato di tentativi, il numero di volte che le chiavi generate dal Server e dai Client sono uguali.

4.5.1 Classe *Test*

Per l'implementazione della classe *Test* si è ricorso al pattern *Singleton* allo scopo di garantire che di questa classe fosse creata una sola istanza. Infatti, poiché i metodi della classe *Test* operano su dei file, l'idea è stata quella di garantire un accesso unificato agli stessi.

```
public class Test { //Singleton
```

```
private static Test instance=null;
private static final int numerotest=1000;

private Test() {}

public static synchronized Test getInstance(){
    if(instance==null) instance = new Test(); // se non esiste
        un'istanza della classe ne crea una
    return instance; // altrimenti restituisce quella esistente
}
```

Il metodo ***NewServerSS()*** viene utilizzato per scrivere in un file di testo le chiavi segrete generate dal Server. Un metodo analogo, non riportato, è previsto per la scrittura su un ulteriore file delle chiavi generate dai molteplici Client.

```
public synchronized void NewServerSS(ECPoint ss) throws
    FileNotFoundException{
    PrintWriter writer = null;
    FileOutputStream fos = new FileOutputStream ("Server.txt", true);
    writer = new PrintWriter(fos);
    writer.println("X: "+ss.getAffineX()+" Y: "+ss.getAffineY());
    writer.close();
}
```

Il metodo ***Verify()*** viene utilizzato per leggere i file di testo contenenti le chiavi segrete generate dal Server e dai Client. Viene quindi verificata l'uguaglianza delle righe dei due file di testo e per ciascuna corrispondenza viene incrementato un contatore.

```
public static int Verify() throws FileNotFoundException, IOException{
    int corretti=0;
    BufferedReader readerS = new BufferedReader(new
        FileReader("Server.txt"));
    BufferedReader readerC = new BufferedReader(new
        FileReader("Client.txt"));
    String line=readerS.readLine();
```

```
        while(line!=null){
            if(line.equals(readerC.readLine())) corretti++; // si confronta
                se le righe sono uguali
            line=readerS.readLine();
        }
        readerS.close();
        readerC.close();
        return corretti;
    }
```

Il metodo ***DeleteFiles()*** è utilizzato per la cancellazione dei file di testo utilizzati per il testing del sistema.

```
    public static void DeleteFiles() {
        File fs=new File("Server.txt");
        File fc=new File("Client.txt");
        if(fs.exists()) fs.delete(); // se esiste viene cancellato
        if(fc.exists()) fc.delete();// se esiste viene cancellato
    }
```

Dal metodo ***main()*** è facile intuire quale sia la procedura del test effettuato:

- viene avviato il Server, che quindi si pone in ascolto sulla porta prestabilita;
- viene lanciata l'esecuzione del Test, che genera un numero di Client a sua volta prestabilito;
- ogni chiave generata dai Client e dal Server viene memorizzata su due file di testo;
- infine si confrontano i due file di testo, che vengono poi cancellati, e si mostra a display il numero di tentativi "corretti".

```
    public static void main(String[] args) throws Exception {
        for(int j=0; j<numerotest; j++){
            Client.main(args); //vengono generati n Client
        }
```

```
        int corretti = Test.Verify(); //si ottiene il numero dei tentativi
            corretti
        Test.DeleteFiles();
        System.out.println("Chiavi condivise uguali:
            "+corretti+"/"+numerotest); //si stampa il risultato
    }
```

4.5.2 Descrizione dei test effettuati

Utilizzando i metodi descritti nel paragrafo precedente è stato effettuato un primo test su 1000 tentativi in locale, cioè sia i Client che il Server erano in esecuzione sullo stesso dispositivo. La curva ellittica che è stata impiegata per la generazione delle chiavi è la *NIST P-256* e il tempo richiesto per il completamento del test è stato pari a 3 minuti e 36 secondi. Il risultato del test è stato di 1000 tentativi corretti sui 1000 effettuati.

Un secondo test è stato effettuato eseguendo i due processi Client e Server su due macchine distinte e sfruttando la rete Internet per la comunicazione. La curva utilizzata dai due processi per la generazione delle chiavi è la *NIST P-128* e anche in tal caso il test ha avuto esito positivo. I risultati del test sono mostrati nelle figure 4.6 e 4.7.

```
Server avviato! In attesa di client!
Client connesso!
Private Key: 112968616650448665331975834511054644091538839423734071993221708954702169692544
Public Key:
X: 104077853169275271587466352731269850362737761469749093629967577431310320157342
Y: 619697973088567152305951094742002200224270453975123756863329732546839536003 51
Client Public key:
X: 101121355645743893150229555607209958864380257727529850299754750857993793747628
Y: 715100169008271452653817307029225085373303037000903787428945942377406206058 46
Shared Secret:
X: 570843575276777486806305410969948397044503782666463695688980724501272750079 25
Y: 110624904641008285039299665640566188899289944163218172358242319128932346099393
```

Figura 4.6: Server

```
Connessione al server in corso...
Connessione avvenuta!
Private Key: 108028881868168687765649139426818170904250493286247572010376777054506079421303
Public Key:
X: 101121355645743893150229555607209958864380257727529850299754750857993793747628
Y: 715100169008271452653817307029225085373303037000903787428945942377406206058 46
Server Public Key:
X: 104077853169275271587466352731269850362737761469749093629967577431310320157342
Y: 619697973088567152305951094742002200224270453975123756863329732546839536003 51
Shared Secret:
X: 57084357527677748680630541096994839704450378266646369568898072450127275007925
Y: 11062490464100828503929966564056618889928994416321817235824231912893234609939 3
```

Figura 4.7: Client

Conclusioni

I criteri da considerare nella scelta di un sistema crittografico per una specifica applicazione sono molteplici. Due di questi sono sicuramente il *livello di sicurezza* e le *performance* offerte. La crittografia ellittica, grazie alla maggiore complessità del problema dell'ECDLP rispetto all'IFP e al DLP, è in grado di offrire lo stesso livello di sicurezza con chiavi di dimensione più piccola (si veda la tabella sotto) e quindi minori requisti di memoria e potenza di calcolo. Ciò rende la crittografia ellittica adatta per l'uso con smart card o in altri contesti nei quali risorse come spazio di memoria, tempo e potenza di calcolo sono limitate [30]. Peraltro, come si è visto nel capitolo 3, i possibili attacchi ai crittosistemi basati sulle curve ellittiche possono essere elusi, con relativa semplicità, facendo attenzione nella scelta della curva da utlizzare e nell'implementazione del sistema crittografico stesso.

Lunghezza della chiave (in bit)		Tempo per la generazione (in sec.)	
RSA	ECC	RSA	ECC
1024	163	0.16	0.08
2240	233	7.47	0.18
3072	283	9.80	0.27
7680	409	133.90	0.64
15360	571	679.06	1.44

Tabella 4.1: Confronto RSA, ECC [29]

Per quanto riguarda il sistema software realizzato, possibili lavori futuri potrebbero

essere volti a migliorare l'efficienza degli algoritmi implementati, ad inserire nuove funzionalità, come ad esempio un meccanismo di autenticazione basato su ECDSA, nonché a renderlo parte di un sistema più complesso.

Bibliografia

[1] Whitfield Diffie, Martin Hellman, *New Directions in Cryptography*, IEEE Transaction on information theory, Vol. IT-22, n. 6, pag. 644-654. `https://ee.stanford.edu/~hellman/publications/24.pdf`, 1976.

[2] Eric W. Weisstein, *Definizione di campo*, `http://mathworld.wolfram.com/Field.html`.

[3] T. Rowland, Eric W. Weisstein, *Definizione di gruppo* `http://mathworld.wolfram.com/Group.htm`.

[4] N. Bray, *Lagrange's Group Theorem*, Resource created by Eric W. Weisstein, `http://mathworld.wolfram.com/LagrangesGroupTheorem.html`.

[5] P. Gaudry, *Integer factorization and discrete logarithm problems*, Ottobre 2014.

[6] Dan Boneh, *Twenty Years of Attacks on the RSA Cryptosystem*, Notices of the AMS, 46, 1999. `https://crypto.stanford.edu/~dabo/papers/RSA-survey.pdf`.

[7] A. Languasco, A. Zaccagnini, *Introduzione alla crittografia*, Ulrico Hoepli Editore, Milano, 2004.

[8] Paul C. Kocher, *Cryptanalysis of Diffie-Hellman, RSA, DSS, and Other Systems Using Timing Attacks*, In Advances in cryptology, CRYPTO'95, pages 171–183. Springer-Verlag, 1995. `https://cdn.preterhuman.net/texts/cryptology/KOCHER_T.PDF`.

[9] J. McKeown, G. Page, B. Schoenfeld, *Attack on RSA*, http://d.umn.edu/~pagex266/RSA_Attacks/rsa-attacks-paper.pdf, 24 Gennaio 2017.

[10] Martin E. Hellman, *An Overview Of Public Key Cryptography*, IEEE Commun. Soc. Mag., vol. 16, Nov. 1978. https://ee.stanford.edu/~hellman/publications/31.pdf.

[11] Jonah Brown-Cohen, *Evidence that the Diffie-Hellman Problem is as Hard as Computing Discrete Logs*, http://theory.stanford.edu/~dfreeman/cs259c-f11/finalpapers/CDHandDLP.pdf.

[12] Andreas V.Meier, *The ElGamal Cryptosystem*, http://wwwmayr.in.tum.de/konferenzen/Jass05/courses/1/papers/meier_paper.pdf, 8 Giugno 2005.

[13] L. Alberini, *Sul documento informatico e sulla firma digitale*, in rivista "Giustizia Civile", tomo due, 1998.

[14] R. Rivest, A. Shamir, L.Adleman, *A Method for Obtaining Digital Signatures and Public Key Cryptosystems*, Communications of the ACM, vol.21 n.2, p.120-126, Feb. 1978. http://people.csail.mit.edu/rivest/Rsapaper.pdf.

[15] D. Hankerson, A. Menezes, S. Vanstone, *Guide to Elliptic Curve Cryptography*, Casa Editrice Springer-Verlag", 2004.

[16] Joseph H. Silverman *The Arithmetic of Elliptic Curvers*, Seconda edizione, Casa Editrice Springer.

[17] William Fulton, *An Introduction to Algebraic Geometry*, Benjamin-Cummings Publishing Co., Gennaio 2008.

[18] Raveen R. Goundar, Marc Joye, Atsuko Miyaji, Matthieu Rivain, Alexandre Venelli *Scalar Multiplication on Weierstraß Elliptic Curves from Co-Z Arithmetic*, http://www.matthieurivain.com/files/jcen11b.pdf.

[19] Lawrence C. Washington *Elliptic Curvers: Number Theory and Cryptography*, Chapman & Hall/CRC, 2008.

[20] Neal Koblitz, *A Course in Number Theory and Cryptography*, Springer-Verlag, 1994.

[21] William Lattin, *Efficient and authenticated key agreement: Meeting new government security requirements* `http://mil-embedded.com/pdfs/Certicom.Spr06.pdf`

[22] J. M. Pollard, *Monte Carlo Methods for Index Computation (mod p)*, Math. Comp. 32, pag. 918-924, 1978.

[23] Matthew Musson, *Attacking the Elliptic Curve Discrete Logarithm Problem.* `https://pdfs.semanticscholar.org/e164/b11488ba143de0ee94887db924a0574d2361.pdf`.

[24] Lawrence C. Washington, *Elliptic curves. Discrete Mathematics and its Applications*, Chapman & Hall/CRC, 2003.

[25] N. Koblitz, A. Menezes, S. Vanstone, *The State of Elliptic Curve Cryptography*, Designs, Codes and Cryptography, Vol. 19, Issue 2–3, pag. 173–193, 2000.

[26] Camille Vuillaume, *Side Channel Attacks on Elliptic Curve Cryptosystems*, Master's thesis, Technische Universitaet Darmstadt, Fachgebiet Informatik, Fachbereich Kryptographische Protokolle, 2004.

[27] Cay Horstmann, *Object-Oriented: Design & Patterns*, second edition, John Wiley & Sons, Inc.

[28] *Interface Runnable* `https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html`.

[29] R. Sinha, H. K. Srivastava, S. Gupta, *Performance Based Comparison Study of RSA and Elliptic Curve Cryptography* International Journal of Scientific & Engineering Research, Vol. 4, Issue 5, May-2013. `https://pdfs.semanticscholar.org/47c5/7bf7b1fce8600bf74bfb6825b2f707b8953e.pdf`.

[30] I. F. Blake, G. Seroussi, N. P. Smart, *Elliptic Curves in Cryptography*, Cambridge University Press, 1999.

Appendice A

Nozioni di algebra

Definizione A.1 (Campo [2])**.** Un campo K è un insieme non vuoto che soddisfa, per entrambe le operazioni di somma e prodotto, i seguenti assiomi:

	Somma	Prodotto
Associatività	$(a+b)+c = a+(b+c)$	$(ab)c = a(bc)$
Commutatività	$a+b = b+a$	$ab = ba$
Distributività	$a(b+c) = ab+ac$	$(a+b)c = ac+bc$
Identità	$a+0 = a = 0+a$	$a \cdot 1 = a = 1 \cdot a$
Inverso	$a+(-a) = 0 = (-a)+a$	$aa^{-1} = 1 = a^{-1}a$ se $a \neq 0$

Esempi di campi sono quelli dei numeri reali $\mathbb{R}$, dei numeri razionali $\mathbb{Q}$ e dei numeri complessi $\mathbb{C}$. Campi con un numero finito di elementi sono detti *"Campi finiti"* o *"Campi di Galois"*. Il numero di elementi di un campo è detto *"ordine"*. Esiste un campo finito $\mathbb{F}$ di ordine q se e solo se $q = p^{\mathrm{m}}$ con p numero primo detto *"caratteristica"* del campo $\mathbb{F}$, e m numero intero positivo. Se $m = 1$ allora il campo $\mathbb{F}$ è detto *"Campo primo"*. Se $m \geq 2$ allora il campo $\mathbb{F}$ è detto *"Campo estensione"*. La caratteristica di un campo $\mathbb{F}$ è pari al minimo numero di volte n tale che l'elemento identità della somma deve essere sommato a se stesso per ottenere l'elemento identità della moltiplicazione. Se non è possibile definire tale n si assume la caratteristica pari a zero. I campi finiti con ordine pari a 2^{m} sono detti *"Campi binari"*.

Definizione A.2 (Gruppo [3])**.** Un gruppo G è dato dall'abbinamento di un insieme finito o infinito di elementi, e una operazione binaria " $*$ ", detta *"Legge di gruppo"*, che soddisfa le seguenti proprietà:

- *Chiusura*: se $a, b \in G$ allora $a * b \in G$;
- *Associativa*: se $\forall a, b, c \in G$ allora $(a * b) * c = a * (b * c)$;
- *Identità*: $\exists! e \in G$ tale che $e * a = a * e = a, \forall a \in G$;
- *Inverso*: $\forall a \in G\ \exists b = a^{-1}$ tale che $a * a^{-1} = a^{-1} * a = e$.

I gruppi per i quali vale la *proprietà commutativa*, cioè $\forall a, b \in G$, $a * b = b * a$, sono detti *"Gruppi Abeliani"*.
Un gruppo G con un numero finito di elementi è detto *"Gruppo finito"* e il numero degli elementi è detto *"ordine"* del gruppo e si indica con $\#G$. È detto, invece, *"sottogruppo"*, un sottoinsieme non vuoto di un gruppo G per il quale valgono i requisiti di gruppo. Una importante relazione, esistente tra l'ordine di un gruppo finito G e quello di un suo sottogruppo S, è definita dal *"Teorema di Lagrange"*.

Teorema A.1 (Teorema di Lagrange [4])**.** *Dato un gruppo finito G con ordine $\#G$, sia S un sottogruppo di G con ordine $\#S$, allora $\#S$ è un divisore di $\#G$.*

Il numero intero dato da $h = \#G/\#S$ è detto *"cofattore"*.

Definizione A.3 (Gruppo ciclico [3])**.** Un gruppo G è detto *"ciclico"* se può essere generato a partire da un singolo elemento g detto *"generatore"*.

Ciò significa che ogni elemento del gruppo può essere ottenuto applicando ripetutamente la Legge di gruppo al generatore g. Il generatore g ha un ordine n che corrisponde al più piccolo valore tale che: $g^{\mathrm{n}} = e$ con e elemento identità. Per i gruppi ciclici l'ordine del generatore corrisponde con l'ordine del gruppo.

Definizione A.4 (Coordinate proiettive)**.** Sia K un campo e c, d due interi positivi. È possibile definire una relazione di equivalenza " $\sim$ " sull'insieme $K^3 \backslash \{(0,0,0)\}$:

$$(X_1, Y_1, Z_1) \sim (X_2, Y_2, Z_2) \text{ se } X_1 = \lambda^{\mathrm{c}} X_2,\ Y_1 = \lambda^{\mathrm{d}} Y_2,\ Z_1 = \lambda Z_2 \text{ per qualche } \lambda \in K.$$

La classe di equivalenza contenente $(X, Y, Z) \in K^3 \backslash \{(0,0,0)\}$ è

$$(X : Y : Z) = \{(\lambda^{\mathrm{c}} X, \lambda^{\mathrm{d}} Y, \lambda Z) : \ \lambda \in K\}.$$

$(X : Y : Z)$ prende il nome di *"Punto proiettivo"*, (X, Y, Z) è detta *rappresentazione* della classe di equivalenza e l'insieme dei punti proiettivi è denotato con $\mathbb{P}(K)$. Notiamo che se $(X', Y', Z') \in (X : Y : Z)$ allora $(X' : Y' : Z') = (X : Y : Z)$. In particolare, se $Z \neq 0$ allora $(X/Z^{\mathrm{c}}, Y/Z^{\mathrm{d}}, 1)$ è una rappresentazione della classe di equivalenza del punto proiettivo $(X : Y : Z)$, più precisamente è l'unica rappresentazione avente la coordinata Z uguale a 1.

L'insieme dei punti proiettivi dato da:

$$\mathbb{P}(K)^0 = \{(X : Y : Z) : X, Y, Z \in K, Z = 0\}$$

è chiamato *linea all'infinito*. I punti proiettivi appartenenti a tale insieme sono detti *punti all'infinito*.