UNIVERSITÀ DI PISA

Dipartimento di Informatica

Corso di Laurea in Informatica

Protocollo Diffie–Hellman su Curve Ellittiche

Studente: Francesco Pirro'

Matricola: 544539

Esperienze di Programmazione

Anno Accademico 2019/2020

Indice

Capitolo 1

Introduzione

Il protocollo di Diffie-Hellman è un protocollo crittografico per lo scambio di chiavi tra due utenti. E' un protocollo a chiave pubblica, il che vuol dire che gli utenti possiedono una chiave pubblica nota a tutti, e una chiave segreta, difficile da calcolare a partire da quella pubblica, ma che ha lo scopo di generare una back-door. [1]

Nella sua forma originaria il protocollo di Diffie-Hellman operava con interi in modulo p, numero primo, e generava chiavi pubbliche, a partire da quelle private, tramite delle operazioni di elevamento a potenza: la difficoltà computazionale del problema inverso, ovvero il logartimo discreto, ci garantisce che ottenere il segreto a partire dalla chiave pubblica sia difficile.

Lo stesso protocollo è stato adattato per funzionare anche su curve ellittiche in campi finiti.

1.1 Curve Ellittiche

1.1.1 Definizione

Una curva ellittica è una curva algebrica, definita su di un campo K, che nella sua forma generica è rappresentata l'equazione:

$$y^2 + axy + by = x^3 + cx^2 + dx + e$$
$$a, b, c, d, e \in K$$

Se la curva è definita su di un campo K con caratteristica diversa da 2 e 3, essa può essere ridotta nella *forma normale di Weierstrass*:

$$y^2 = x^3 + ax + b$$
$$a, b \in K$$

La caratteristica k di un campo algebrico K è il più piccolo numero per cui sommando k volte l'elemento neutro della moltiplicazione si ottiene l'elemento neutro dell'addizione.[1]

All'insieme dei punti $(x, y) \in K^2$, che soddisfano l'equazione, si può attribuire la struttura algebrica di un *gruppo Abeliano additivo*.
Un *gruppo Abeliano additivo* è un gruppo algebrico in cui esiste un'operazione binaria che gode della proprietà commutativa. [2] Un'operazione binaria è una funzione che prende due elementi di un insieme e ne restituisce un terzo che fa parte dello stesso insieme.[3] Quello di cui si necessita è quindi un'operazione di composizione di due punti di una della curva, che generi un terzo punto sempre sulla curva, e che questa sia commutativa. Questa operazione verrà chiamata con il nome di Somma di Punti.

1.1.2 Somma di Punti

Prima di considerare l'operazione di somma su curve ellittiche in campi finiti, consideriamo il suo significato su curve nel campo dei numeri reali $\mathbb{R}$.
Definiamo l'insieme E come l'insieme dei punti appartenenti ad una curva:

$$E(a,b) = \{(x,y) \in \mathbb{R}^2 \mid y^2 = x^3 + ax + b\}$$
$$a, b \in \mathbb{R}$$

Si nota che questo insieme contiene un punto speciale, definito come "punto all'infinito" O: ovvero il punto all'infinito sull'asse delle ordinate.[1] Il nostro punto all'infinito sarà l'elemento neutro della somma tra punti.
Si nota anche come la curva ellittica è simmetrica rispetto l'asse delle x: infatti

$$(x,y) \in E(a,b) \implies (x,-y) \in E(a,b)$$

Se $P = (x,y)$, possiamo definire il suo inverso $-P = (x,-y)$.
Consideriamo due punti P e Q, tracciamo la retta che passa passante per essi e valutiamo le sue intersezioni con la curva ellittica $E(a,b)$: se sostituiamo l'equazione della retta $y = mx + q$ nella y di quella della curva otteniamo un'equazione di terzo grado per x:

$$(mx+q)^2 = x^3 + ax + b$$

che può avere o una soluzione reale e due soluzioni complesse e coniugate, o tre soluzioni reali (in alcuni casi con due soluzioni coincidenti). [1]

Dati tre punti P, Q, R della curva E definiamo l'operazione di addizione su una curva ellittica a partire dalla seguente proprietà: se P, Q, R sono disposti

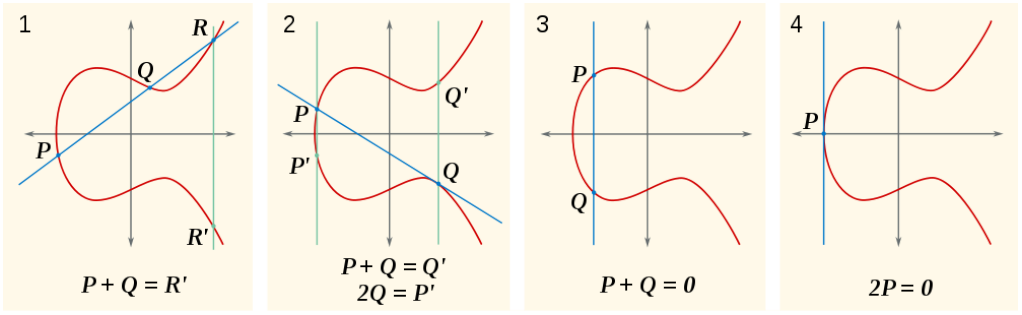


Figura 1.1: Somma di Punti (https://i.stack.imgur.com/iwBHR.png)

su una retta, la loro somma equivale al punto all'infinito.

$$P + Q + R = O$$

$$P + Q = -R$$

La somma di due punti P e Q è quindi l'inverso del terzo punto di intersezione della retta passante per P e Q. Se P e Q sono coincidenti, la retta è la tangente alla curva.

Per permettere che la tangente sia calcolabile in ogni punto si pone un limite ai valori di a e b:

$$4a^3 + 27b^2 \neq 0$$

in modo che l'equazione $x^3 + ax + b$ non abbia radici multiple e la curva sia priva di punti singolari.[1]

Se P e Q hanno la stessa coordinata x, $P + Q = P + (-P) = O$.

Questa operazione è binaria rispetto alla curva, in quanto genera sempre punti contenuti nell'insieme E, ed è banalmente commutativa: di conseguenza ci permette di attribuire all'insieme E la struttura algebrica di gruppo abeliano. Ottenere le coordinate del punto somma è immediato, bisogna risolvere il sistema tra le equazioni della retta passate per P e Q e la curva ellittica. Se $P = Q$ allora si utilizza la tangente.

La somma $P + Q$ genera un punto S di coordinate:

$$x_s = \lambda^2 - x_p - x_q$$

$$y_s = -y_p + \lambda(x_p - x_s)$$

$$\text{con } \lambda = \begin{cases} (y_q - y_p)/(x_q + x_p) \text{ se } P \neq Q \\ (3x_p^2 + a)/2y_p \text{ se } P = Q \wedge y_p \neq 0 \end{cases}$$

Se $y_p = 0$ allora $2P = O$

Se $y_p = -y_q$ allora $P + Q = O$

Se utilizziamo l'accortezza di utilizzare le operazioni in modulo, tutte le considerazioni effettuare sulle curve ellittiche sul campo dei numeri reali si possono adattare al caso dell'algebra modulare.[1] Le curve ellittiche su campi $\mathbb{Z}_p$, p numero primo, sono dette *curve ellittiche prime*.

1.1.3 Moltiplicazione scalare di Punti

Definiamo la moltiplicazione scalare di un punto della curva ellittica come:

$$Q = kP = P + P + ... + P \ (k \text{ volte})$$

Defiano come ordine n di un punto P il più piccolo intero per cui:

$$nB = O$$

L'operazione inversa consiste nel trovare il più piccolo intero k tale che $Q = kP$. Questo problema prende il nome di *logaritmo discreto per le curve ellittiche*, e risolverlo tramite algoritmi noti ha complessità esponenziale se i parametri della curva sono scelti opportunamente.

La moltiplicazione scalare è la nostra funzione one-way da utilizzare per la crittografia a chiave pubblica.

1.2 Protocollo Diffie-Hellman

Il protocollo di Diffie-Hellman è un protocollo che consente lo scambio di una chiave tra due utenti tramite un canale non sicuro: entrambi gli utenti generano un'informazione segreta e a partire da questa ottengono una chiave pubblica, in seguito si scambiano le chiavi pubbliche e combinano il proprio segreto con la chiave pubblica ricevuta dall'altro utente, generando entrambi la stessa chiave privata, usata successivamente per un cifrario simmetrico.
Nella sua forma originaria, il protocollo di Diffie-Hellman usava l'elevamento a potenza in modulo come funzione per generare la chiave pubblica a partire da un esponente, che diventava il segreto, sfruttando quindi la difficoltà di risolvere il problema del logaritmo discreto. E' stata anche sviluppata una versione dello stesso protocollo che funziona su curve ellittiche perchè promette una sicurezza maggiore a parità di bit della chiave.
Il NIST definisce sia le modalità di generazione della chiave pubblica e del segreto, sia della creazione della chiave privata comune. [4] Per prima cosa viene definito un dominio dei parametri comune a tutti gli utenti che vogliono utilizzare il protocollo, per semplicità eviteremo di citare quelli usati per tipi di curve diversi da quelle ellittiche prime. Per prima cosa si ha un numero primo p, che specifica il gruppo $\mathbb{Z}_p$, i parametri a, b della curva sempre in modulo p, e un punto G della curva di ordine n. Per prima cosa entrambi gli utenti generano un intero c, utilizzando un opportuno generatore di numeri randomici crittograficamente sicuro e ricavano da esso il proprio segreto d

$$d = (c \mod (n-1)) + 1$$

e la propria chiave pubblica Q

$$Q = dG$$

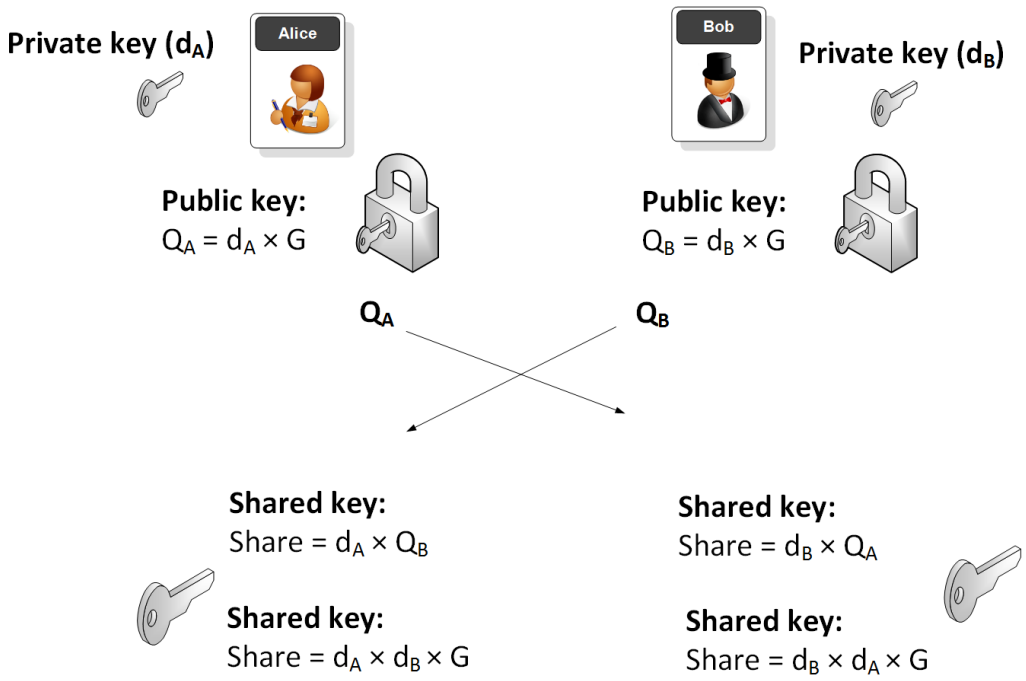


Figura 1.2: Diffie-Hellman (https://asecuritysite.com/public/ecdh.png)

Una volta ottenuta la chiave pubblica Q_u dell'altro utente si puo' proseguire al calcolo della chiave condivisa Z

$$Z = dQ_u$$

La correttezza del protocollo è data dal fatto che, se chiamiamo (d_1, Q_1, Z_1) e (d_2, Q_2, Z_2) le triple di chiave pubblica e privata dell'Utente 1 e 2

$$Z_1 = d_1 Q_2 = d_1 d_2 G$$

$$Z_2 = d_2 Q_1 = d_2 d_1 G$$

$$Z_1 = Z_2 = (x_z, y_z)$$

Un utente esterno, che non conosce nessuno dei due segreti d_1 e d_2, e, non potendo calcolarseli a partire da Q_1 e Q_2 data la difficoltà di invertire l'operazione di moltiplicazione scalare, non ha modo di ottenere il segreto condiviso.

Table 24: Appoved elliptic curves for ECC key-agreement.

Referenced in:	FIPS 186-4 SP 800-56A	TLS (RFC 4492) (SP 800-52)	IPsec w/ IKE v2 (RFC 5903)	Targeted Security Strengths that can be Supported
Specified in:	SP 800-186[29]	SEC 2	RFC 5903	
	P-224	secp224r1	-	$s = 112$
	P-256	secp256r1	secp256r1	$112 \le s \le 128$
	P-384	secp384r1	secp384r1	$112 \le s \le 192$
	P-521	secp521r1	secp521r1	$112 \le s \le 256$
	K-233	sect233k1	-	$112 \le s \le 128$
	K-283	sect283k1	-	$112 \le s \le 128$
	K-409	sect409k1	-	$112 \le s \le 192$
	K-571	sect571k1	-	$112 \le s \le 256$
	B-233	sect233r1	-	$112 \le s \le 128$
	B-283	sect283r1	-	$112 \le s \le 128$
	B-409	sect409r1	-	$112 \le s \le 192$
	B-571	sect571r1	-	$112 \le s \le 256$

Figura 1.3: Curve accettate dal NIST in [4],[5]

Il NIST consiglia l'utilizzo di alcune curve ellittiche, sulle quali garantisce che il problema del logaritmo discreto su curve ellittiche sia computanzionalmente complesso, e per ognuna fornisce anche un punto G generatore. [5]

Capitolo 2

Algoritmi

Per rendere il protocollo di Diffie-Hellman utilizzabile, si necessita che che la moltiplicazione scalare sia facile da calcolare. Sommare k volte il punto P non è una soluzione possibile perchè in applicazioni reali si parla di un numero ad almeno 128 bit, c'è bisogno quindi di algortimi efficienti.
Vediamo due algoritmi per il calcolo di

$$Q = dP$$

2.1 Double-and-Add

L'algoritmo Double-and-Add è la versione su curve ellittiche dell'algortimo delle esponenziazioni successive per il calcolo di una potenza, ed è quello base per il calcolo delle moltiplicazione scalare.
Per prima cosa scriviamo il nostro scalare d come somma di potenze di 2:

$$d = d_0 + 2d_1 + 2^2 d_2 + 2^3 d_3 + ... + 2^m d_m$$

$$\text{con } d_0, d_1, ...d_m \in 0, 1 \text{ e } m = \lfloor \log_2 d \rfloor$$

Partendo da P, calcoliamo ogni volta il raddoppio $2P, 4P, 8P, ..., 2^m P$ e sommiamo tra di loro i punti $2^i P$ per cui $d_i = 1$.
Osservando la rappresentazione di d, e ricordando che la proprietà commutativa, ma anche quella associativa, vale per la somma di punti su curve ellittiche, l'algoritmo è corretto e ha complessità $\mathcal{O}(\log d)$[1]. Per la precisione si tratta di esattamente m operazioni di raddoppio, e di al più $m + 1$ operazioni di somma.
Da [6] l'algortimo in pseudo codice, nella variante con indici crescenti:

Algorithm 2.1: Double-and-Add

```
1  begin
2  N ← P
3  Q ← 0
4  for i from 0 to m do:
5      if d_i = 1 then
6          Q ← point_add(Q,N)
7      N ← point_double(N)
8  return Q
```

2.2 wNAF

Il methodo wNAF utilizza la rappresentazione in forma non adiacente di d, da cui prende appunto il nome. Sia $w \geq 2$ un intero positivo, allora l'intero d è descritto da questa espressione:

$$d = \sum_{i=0}^{l-1} k_i 2^i$$

$$k_i \neq 0 \implies (i \mod 2 = 1)$$

$$| k_i | \leq 2^{w-1}$$

Le proprietà di questa rappresentazione è che è unica, e che per ogni w cifre consecutive almeno una è diversa da 0. Il numero di cifre non zero nella rappresentazione w-NAF è $\frac{l}{w+1}$, con $l \leq \lfloor \log_2 d \rfloor$.[7]
Possiamo pre-calcolare i punti $k_i P$ con $-2^{w-1} \leq k_i \leq 2^{w-1}$ e utilizzare la rappresentazione di d nella forma w-NAF per calcolare $Q = dP$.
L'algoritmo richiede quindi $\frac{l}{w+1}$ addizioni, e l raddoppi nell'algoritmo principale e 2^{w-1} somme nella fase di pre-computazione. Ricordando che $l \leq \lfloor \log_2 d \rfloor$, l'algoritmo che sfrutta w-NAF effettua meno operazioni su punti rispetto al double-and-add. [7]

Algorithm 2.2: Calcolo w-NAF

```
1   begin
2   l ← 0
3   while d ≥ 1 do
4       if d mod 2 ≠ 0 then
5           k_l ← 2 − (d mod 2^w)
6           d ← d − k_l
7       else
8           k_l ← 0
9       d ← d/2
10      l ← l+1
11  return (k_{l−1}, ... ,k_1 ,k_0)
```

Algorithm 2.3: Precalcolo Punti

```
1   begin
2   P_1 ← P
3   P_2 ← point_double(P)
4   foreach( i = 3 ,5 .... 2^{w−1})
5       P_i ← point_sum(P_{i−2} ,P_2)
6       P_{−i} ← −P_i
7   return (P_{−2^{w−1}}, ... ,P_{−3} ,P_{−1} ,P_1 ,P_3 , ... P_{2^{w−1}})
```

Algorithm 2.4: Moltiplicazione tramite w-NAF

```
1  begin
2  Q ← 0
3  for j from l−1 downto 0 do
4      Q ← point_double(Q)
5      if d_j ≠ 0 then
6          Q ← point_add(Q, P_{d_j})
7  return Q
```

Capitolo 3

Implementazione in Python

Il Protocollo è implementato in Python, rispettando i passaggi e i controlli richiesti dal NIST in [4]. Inoltre per la generazione di bit randomici è stata utilizzata la funzione urandom, che secondo la documentazione di Python è adatta per applicazioni crittografiche.[8]
La classe "DH-ECC", una volta inizializzata leggendo i dati di una curva ellittica da un file .xml,permette la creazione del segreto e della chiave pubblica tramite una funzione chiamata "key_pair_generator" e la chiave segreta condivisa tramite la funzione "shared_key_generator". Se inizializzata con il parametro "wNAF=1", la curva ellittica utilizzerà la forma adiacente per la moltiplicazione scalare. La classe che implementa il protocollo è molto semplice, e tutta l'aritmetica è nascosta dalle classi che implementano le curve ellittiche.
Ci sono due classi che implementano le Curve Ellittiche: "EllipticCurve" e "WNAF": la prima implementa le operazioni elementari di somma, raddoppio e la moltiplicazione tramite Double-and-Add, più alcune funzioni ausiliarie come l'algoritmo di euclide esteso per il calcolo dell'inverso in modulo; mentre

la seconda estende la prima classe e sovrascrive la funzione di moltiplicazione per implementare l'algoritmo w-NAF. Python non pone un limite agli interi, quindi possiamo avere numeri di qualsiasi bit, il che ci permette di lavorare molto tranquillamente con i numeri di grandi dimensioni di un algortimo crittografico. L'implementazione degli algortimo riprende la descrizione in pseudocodice precedentemente data, mentre per la somma e il raddoppio le formule sono quelle ricavate e descritte nell'introduzione.

Particolare attenzione è stata posta ai casi limiti nella somma e nel raddoppio, che riguardano soprattuto il punto all'infinito e la somma di punti opposti. Vista la complessità dell'operazione di modulo, si è preferito ridurre al minimo il suo utilizzo e applicarlo sono quando strettamente necessario.

Per il calcolo dell'inverso in modulo si è applicato l'algortimo di Euclide esteso, su cui però non si spende molto a riguardo visto che è molto semplice e non è il nostro focus.

Per l'algortimo w-NAF si utilizza un array per contenere i vari k_i della rappresentazione, e un dizionario per la raccolta dei punti precomputati.

Capitolo 4

Test e Analisi Complessità

Dall'analisi dell'algoritmo, è facilmente raggiungibile la conclusione che l'algortimo w-NAF effettua meno somme e raddoppi della sua controparte double and add, ma la sua efficienza è influenzata soprattutto dalla scelta di w
Utilizzando le curve consigliate dal NIST, si è utilizzata la nostra implementazione del protocollo per generare un tot numero di chiavi, e si è misurato il tempo impegato per generarle. Si passa da 1000 chiavi generate per la curva P-192, fino a 200 per la P-521, questo per permettere la raccolta di dati in tempi ragionevoli, visto che la curva P-521 utilizza numeri da 521 bit.
Per ogni curva, e per valori di w da 2 a 6 eseguiremo delle analisi temporali per trovare il parametro migliore per w, ed una volta trovato, lo paragoneremo utilizzando lo stesso test contro l'algoritmo base e valuteremo in percentuale l'incremento in performance.

Tabella 4.1: Media dei tempi di esecuzione (in secondi) di WNAF

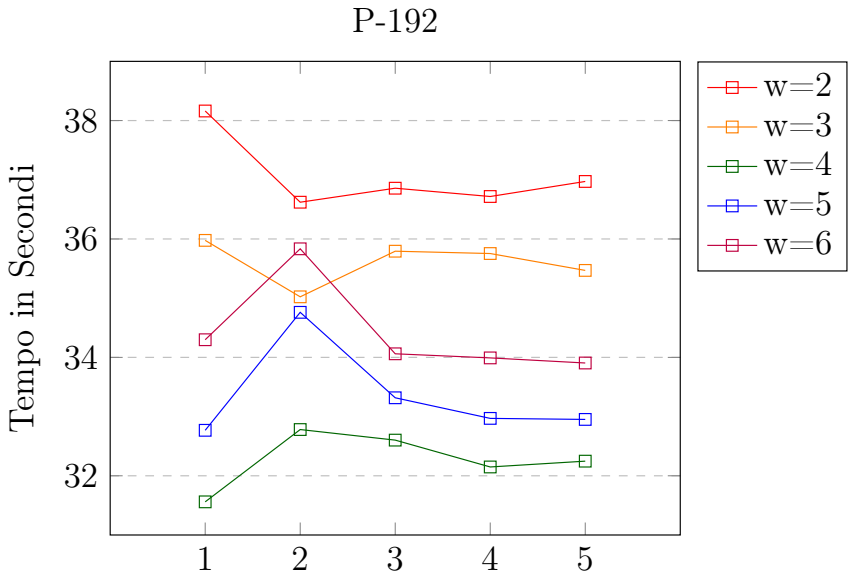
curva	nr. chiavi	w=2	w=3	w=4	w=5	w=6
P-192	1000	37,0658	35,6032	32.2679	33,3537	34,4177
P-224	800	41,1389	39,3827	35,9550	36,4019	37,2664
P-256	600	39,4041	38,1113	35,3145	37,1548	37,7814
P-384	400	64,2393	61,1978	57,2765	57,2308	57,9385
P-521	200	63,9713	61,1595	58,0838	58,1239	58,9967

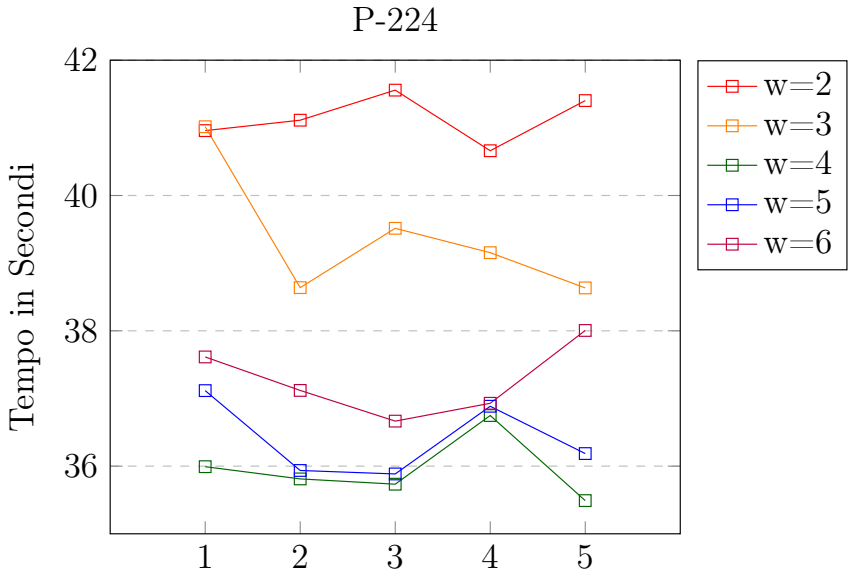
4.1 Influenza del parametro w

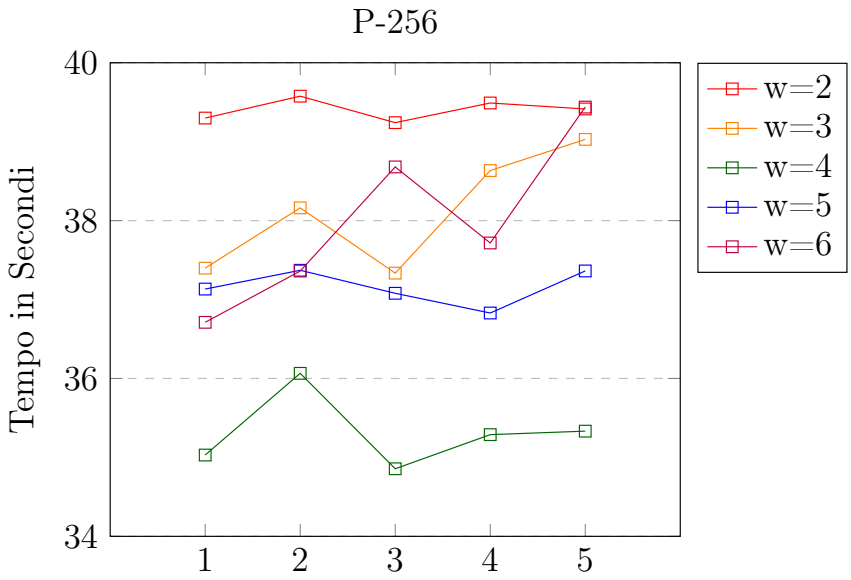
Per ogni curva è stato calcolato il tempo impiegato a generare un numero x di chiavi utilizzando $w = 2, 3, 4, 5, 6$; ripetendo il tutto per cinque volte: quello che abbiamo ottenuto per ogni curva sono 5 tempi per ogni valore del parametro da 2 a 6. Analizzando i dati è palese che la scelta peggiore sia $w = 2$, in quanto ha sempre generato i tempi più alti per ogni tipo di curva, ma è altrettanto palese come quella migliore sia $w = 4$. In ogni misurazione nelle curve P-192, P-224 e P-256 $w = 4$ è sempre quello con i tempi minori, in alcuni punti avvicinandosi molto ai valori registrati da $w = 5$ ma rimanendo sempre costantemente sotto ad esso.

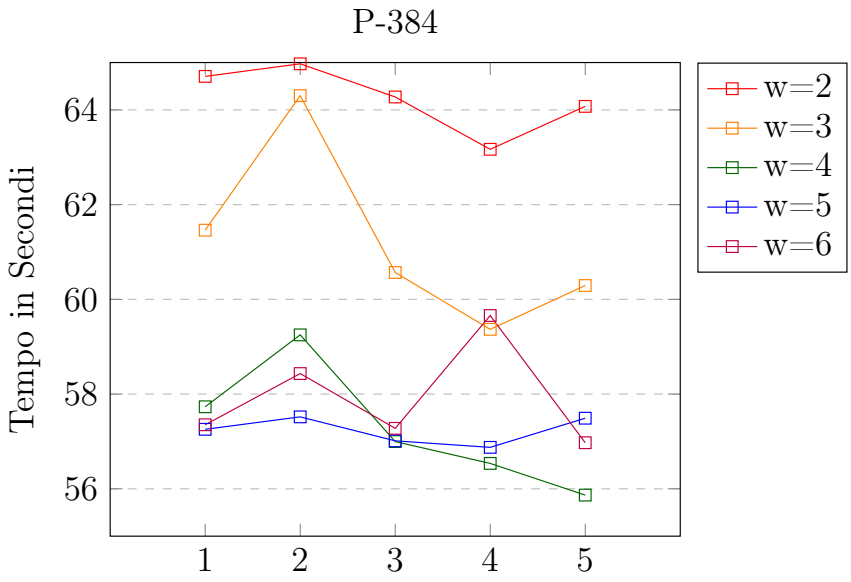
Nelle curve P-384 e P-521 i margini tra $w = 4$ e $w = 5$ in alcuni momenti diminuscono, fino ad avere situazioni in cui $w = 5$ performa meglio: calcolando la media i due hanno un margine di differenza molto piccolo, con un lieve vantaggio a favore di $w = 5$ in P-384 e uno lieve per $w = 4$ in P-521. Tuttavia in queste due curve, l'algortimo con parametro uguale a 4 ha performato meglio rispetto agli altri 6 volte su 10, ed è cosnsistentemente migliore rispetto agli altri sulle restanti curve, ed è quindi il valore da scegliere. Riguardo gli altri si nota dalle tabelle e dai grafici allegati come $w = 2$ e $w = 3$

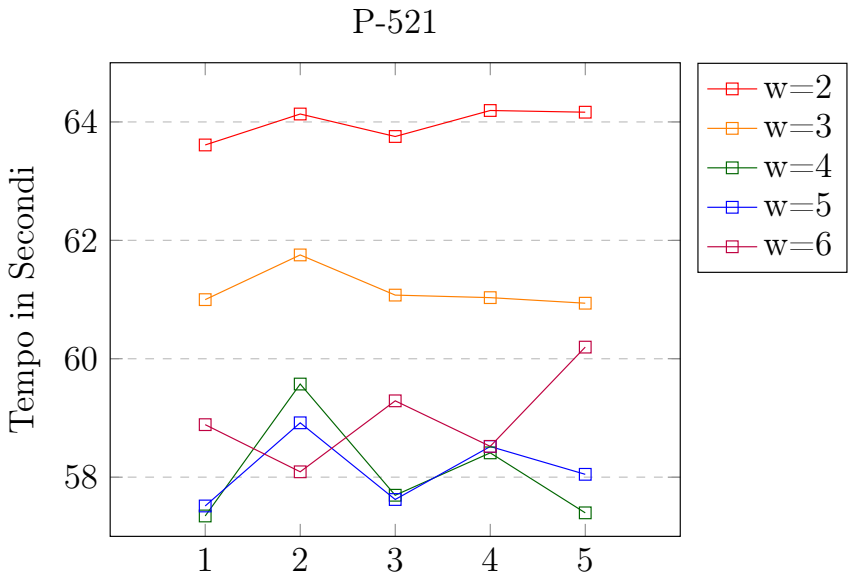
siano costantemente i peggiori mentre come le differenze tra $w = 4$, $w = 5$, $w = 6$ diminuiscono con l'aumentare delle dimensioni dei bit delle curve.






P-256
Tempo in Secondi
40
38
36
34
1
2
3
4
5
w=2
w=3
w=4
w=5
w=6


P-384
Tempo in Secondi
64
62
60
58
56
1
2
3
4
5
w=2
w=3
w=4
w=5
w=6



4.2 Analisi tempo di esecuzione

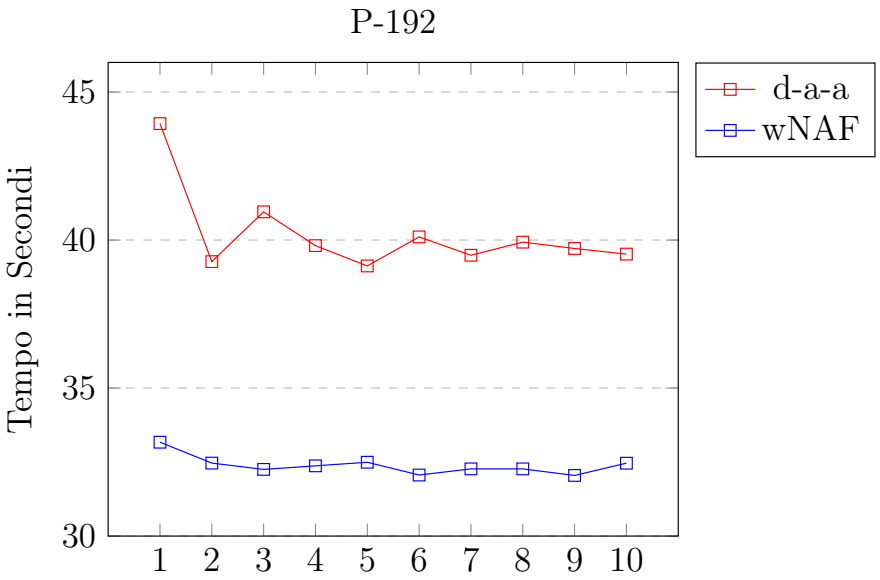
Avendo scelto il parametro per il w-NAF, si passa dunque all'analisi e al paragone con il double-and-add. Il procedimento di raccolta di dati è simile a quello utilizzato per la scelta del paramentro, solo che dovendo scegliere tra due algoritmi invece che tra 5 diversamente varianti, per ogni curva il procedimento è stato ripetuto 10 volte.

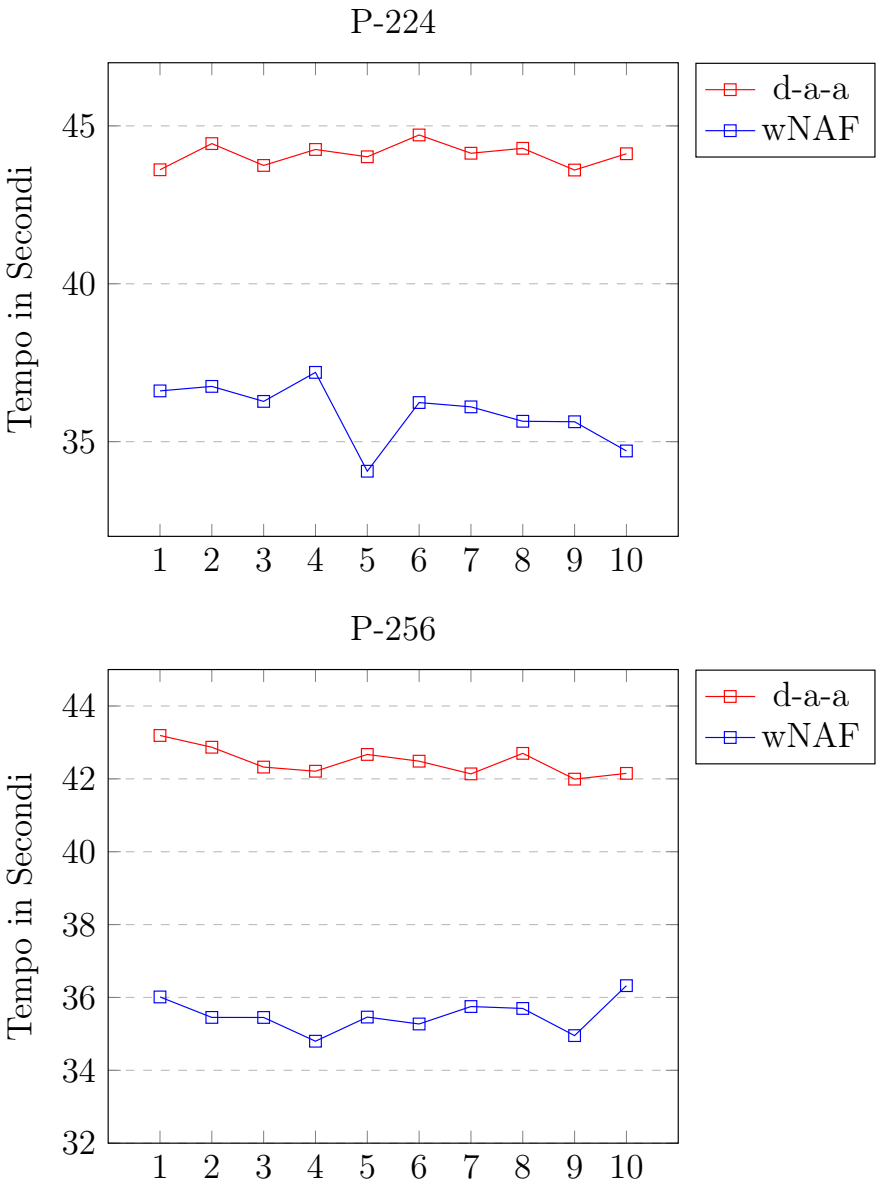
Dai dati raccolti è evidente che l'intuizione teorica è confermata perchè l'algortimo w-NAF performa meglio del double-and-add in qualsiasi circostanza; si puo' addirittura notare, paragonando i dati raccolti con quelli precedentemente analizzati che w-NAF con $w = 2$, che era la scelta peggiore per il parametro, risulta performare meglio del double-and-add su tutte le curve.

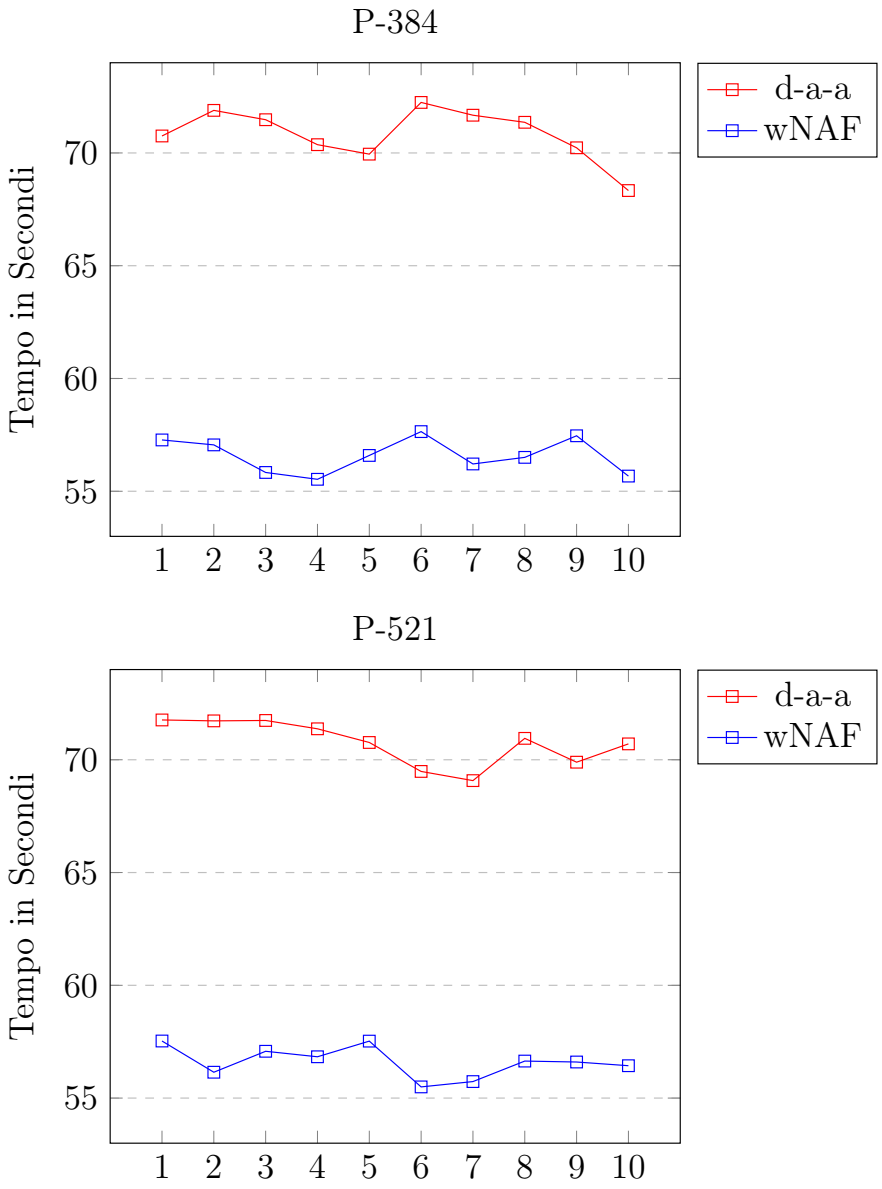
L'incremento di performance raggiunto con l'algoritmo è notevole, in quanto analizzando le medie dei dati raccolti otteniamo un aumento percentuale del 18%-20% su tutte le curve tranne che nella P-256 su cui risulta solo del 16%.

Tabella 4.2: Confronto della media dei tempi di esecuzione (in secondi)

curva	nr. chiavi	doub. add	wNAF	%
P-192	1000	40,1859	32,4041	19,36%
P-224	800	44,0923	36,1206	18,07%
P-256	600	42,4724	35,5172	16,38%
P-384	400	70,8260	56,5771	20,11%
P-521	200	70,7477	56,6010	19,99%




P-224
Tempo in Secondi
45
40
35
1 2 3 4 5 6 7 8 9 10
d-a-a
wNAF
P-256
Tempo in Secondi
44
42
40
38
36
34
32
1 2 3 4 5 6 7 8 9 10
d-a-a
wNAF



4.3 Potenziali Migliorie

Dopo aver discusso delle prestazioni degli algortimi, un punto in cui le prestazioni possono migliorare di molto è sulle operazioni aritmetiche, e soprattutto sul modulo, considerando i numeri molto grandi con cui lavora questo algorti-

mo. Il NIST, per ognuna delle sue curve fornisce dei valori con cui è possibile effettuare le operazioni di modulo efficientemente, utilizzando una decomposizione in blocchi da 64 bit. [4],[5]

Questo però è possibile se l'implementazione è di una sola curva, e non nel caso in cui si voglia mantenere la possibilità di poter utilizzare qualsiasi parametro come in questa implementazione.

Bibliografia

[1] A. Bernasconi, P. Ferragina, and F. Luccio, *Elementi di Crittografia.* Pisa University Press, 2015.

[2] Wikipedia, "Gruppo abeliano."

[3] ——, "Operazione binaria."

[4] E. Barker, L. Chen, A. Roginsky, A. Vassilev, and R. Davis, "Nist special publication 800-56a revision 3 recommendation for pair-wise key establishment schemes using discrete logarithm cryptography," *NIST: Gaithersburg, MD, USA*, 2018.

[5] L. Chen, D. Moody, A. Regenscheid, and K. Randall, "Recommendations for discrete logarithm-based cryptography: Elliptic curve domain parameters," National Institute of Standards and Technology, Tech. Rep., 2019.

[6] Wikipedia, "Elliptic curve point multiplication."

[7] D. Khleborodov, "Fast elliptic curve point multiplication based on window non-adjacent form method," *Applied Mathematics and Computation*, vol. 334, pp. 41–59, 2018.

[8] Python, "os — miscellaneous operating system interfaces." [Online]. Available: https://docs.python.org/3/library/os.html

Appendice A

Codice

Il codice scritto puo' essere trovato su Github:
https://github.com/pirrofra/DH-ECC

curve.py

```
#Classe che definisce un Punto, e conserva le coordinate x e y
class Point:
    x=0
    y=0

    def __init__(self,x,y):
        self.x=x
        self.y=y

#Classe che definisce una curva ellittica e le operazioni elementari possibili in essa.
class EllipticCurve:
```

```
12      __a=0
13      __b=0
14      __p=0
15
16      def __init__(self,a,b,n):
17          self.__a=a
18          self.__b=b
19          self.__p=n
20
21      #Inverso in modulo p tramite l'algoritmo di Euclide Esteso
22      def __mulinv(self,x):
23          t=0
24          r=self.__p
25          new_t=1
26          new_r=x
27
28          while new_r != 0:
29              q= r // new_r
30              t,new_t=new_t, t-(q*new_t)
31              r,new_r=new_r, r-(q*new_r)
32
33          if t<0:
34              return t+self.__p
35          else:
36              return t
37
38      #calcolo del doppio di un punto P
39      def double (self,p):
40          #il doppio del punto all'infinito, è il punto all'↩
                infinito
41          if p.x==0 and p.y==0:
42              return Point(0,0)
```

```
43          lmb_n= ((3*(p.x**2))+self.__a) #numeratore del lamba, ↩
                della formula del raddoppio
44          lmb_d=(2*p.y) % self.__p #denominatore del lamba, della↩
                 formula del raddoppio
45          if lmb_d==0: #se il denominatore è 0, la coordinata y ↩
                di P era 0, quindi la tangente incontra la curva al↩
                 punto all'infinito
46              return Point(0,0)
47          #per effettuare il calcolo di lamba, in modulo, ↩
                dobbiamo calcolare l'inverso e poi moltiplicare
48          #non possiamo dividere
49          lmb_d_inv=self.__mulinv(lmb_d)
50          lmb=(lmb_n*lmb_d_inv)
51          xr=((lmb**2) - (2*p.x)) % self.__p
52          yr=((lmb*(p.x-xr)) -p.y) % self.__p
53          return Point(xr,yr)
54
55      def sum(self, p,q):
56          #controllo se P e Q sono punti all'infinito
57          if(p.x==0 and p.y==0):
58              return q
59          if(q.x==0 and q.y==0):
60              return p
61          #Controllo se la somma che sto facendo in àrealt è un ↩
                raddoppio
62          if(p.x==q.x and p.y==q.y):
63              return self.double(p)
64          lmb_n =  (q.y - p.y) #numeratore del lamba, della ↩
                formula della somma
65          lmb_d = (q.x - p.x) % self.__p #denominatore del lamba,↩
                 della formula della somma
66          if(lmb_d==0): #se il denominatore è 0, P e Q sono ↩
                opposti quindi la loro somma è il punto all'infinito
```

```
67              return Point(0,0)
68          #per effettuare il calcolo di lamba, in modulo, ↩
               dobbiamo calcolare l'inverso e poi moltiplicare
69          #non possiamo dividere
70          lmd_d_inv=self.__mulinv(lmb_d)
71          lmb=(lmb_n*lmd_d_inv)
72          xr=((lmb**2) - (p.x) - (q.x) ) % self.__p
73          yr=(lmb*(p.x-xr) - p.y) % self.__p
74          return Point(xr,yr)
75
76      #Algortimo double and add per moltiplicare
77      def __double_and_add(self,d,P):
78          R=Point(0,0)
79          while d!=0:
80              if d % 2 == 1:
81                  R=self.sum(R,P)
82              P=self.double(P)
83              d=d//2
84          return R
85
86      #Verifica la presenza di un punto sulla curva sostituendo x↩
            e y nell'equazione della curva.
87      def isOnCurve(self,P):
88          arg1=P.y**2 % self.__p
89          arg2=((P.x**3) + (self.__a*P.x)+self.__b) % self.__p
90          if arg1==arg2:
91              return True
92          else:
93              return False
94
95      def mul(self,d,P):
96          return self.__double_and_add(d,P)
```

wnaf.py

```
1  from curve import *
2
3  #Classe che estende EllipticCurve, ed implementa wNAF
4  class WNAF(EllipticCurve):
5
6      #questo costruttore richiede in ùpi la w, rispetto alla sua↩
           superclasse
7      def __init__(self,a,b,n,w):
8          super(WNAF,self).__init__(a,b,n)
9          self.__w=w
10         self.__expw=pow(2,w)
11
12     #Funzione che implementa il modulo in segno
13     def __mod(self,d):
14         mod=d % self.__expw
15         if  mod >= self.__expw//2:
16             return mod -self.__expw
17         else:
18             return mod
19
20     #Funziona che calcola il w-NAF di d
21     def __NAF(self,d):
22         i=0
23         naf=[] #array che conserva le cifre k della ↩
               rappresentazione, di dimensione i
24         while d>0:
25             if(d % 2)==1:
26                 d_i=self.__mod(d)
27                 naf.append(d_i)
28                 d=d-d_i
```

```
29              else:
30                  naf.append(0)
31              d=d//2
32              i=i+1
33          return (i-1,naf)
34
35      #Pre calcoli dei punti {P,3P,5P.... } e {-P,-3P,-5P....}
36      def __precalc(self,P):
37          precalculated={} #dizionario che conserva i punti
38          i=1
39          double_p=self.double(P)
40          Q=P
41          while(i<=self.__expw-1):
42              precalculated[i]=Q
43              precalculated[-i]=Point(Q.x,0-Q.y) #-P ha la stessa↩
                    coordinata x di P, ma y di segno opposto.
44              Q=self.sum(Q,double_p)
45              i=i+2
46          return precalculated
47
48      #algoritmo per la moltiplicazione
49      def __wNAF(self,d,P):
50          (i,naf)=self.__NAF(d) #calcola il w-NAF
51          precalculated=self.__precalc(P) #precalcolo dei punti
52
53          #Ottiene Q a partire dalla sua rappresentazione w-NAF
54          Q=Point(0,0)
55          while(i>=0):
56              Q=self.double(Q)
57              di=naf[i]
58              if(di != 0):
59                  Q=self.sum(Q,precalculated[di])
60              i=i-1
```

```
61          return Q
62
63      #sovrascrive mul èaffinch utilizzi wNAF e non double-and-↩
            add
64      def mul(self,d,P):
65          return self.__wNAF(d,P)
```

dh.py

```
1  from curve import *
2  from wnaf import *
3  import os
4  import struct
5  import sys
6  import xml.etree.ElementTree as ET
7
8  class ECC_CDH:
9      __ecc=None
10     __n=0
11     __G=None
12     #Costruttore
13     def __init__(self,path,wNAF=0,w=4):
14         tree=ET.parse(path)
15         curve=tree.getroot()
16         #Lettera dei valori dal file .xml in path
17         p=int(curve.find("p").text,10)
18         a=int(curve.find("a").text,10)
19         b=int(curve.find("b").text,10)
20         n=int(curve.find("n").text,10)
21         G=curve.find("G")
```

```
22          x=int(G.find("x").text,10)
23          y=int(G.find("y").text,10)
24          #scelta se utilizzare wNAF o meno, in base al valore ↩
                del parametro
25          if wNAF==0:
26              self.__ecc=EllipticCurve(a,b,p)
27          else:
28              self.__ecc=WNAF(a,b,p,w)
29          self.__n=n
30          self.__G=Point(x,y)
31
32      #Funzione che genera la chiave pubblica Q, e il segreto d ↩
            come specificato dal NIST
33      def key_pair_generator(self):
34          L=self.__n.bit_length()+64
35          returned_bits=os.urandom(L) #funzione ↩
                crittograficamente sicura
36          c=int.from_bytes(returned_bits,byteorder=sys.byteorder,↩
                signed=False)
37          d= (c % (self.__n -1))+1 #calcolo di d, a partire dal ↩
                valore randomico ottenuto
38          Q= self.__ecc.mul(d,self.__G) #Q=d*G
39          #Errori lanciati se la chiave generata è 0, o è ↩
                generato un punto non appartente alla curva
40          #Controllo richiesto dal NIST
41          if Q.x==0 and Q.y==0:
42              raise ValueError("Public Key is zero")
43          elif(not self.__ecc.isOnCurve(Q)):
44              raise ValueError("Invalid Point")
45          return d,Q
46
47      #Funzione che ottiene la chiave segreta condiva a partire ↩
            dal proprio segreto d e dalla chiave pubblica Q di un ↩
```

```
        altro utente
48      def shared_key_generator(self,d,Q):
49          P=self.__ecc.mul(d,Q) #P=d*Q
50          if P.x==0 and P.y==0:
51              raise ValueError("P is zero")
52          z=P.x #si restituisce solo la coordinata x, come richiesto dal NIST
53          return z
```

wnaf_bench.py

```
1  import timeit
2
3  #Testa DH generando simulando lo scambio di chiavi tra due utenti, e controllando che entrambi generino la stessa chiave z
4  def key_test(dh):
5      try:
6          d1,Q1=dh.key_pair_generator()
7          d2,Q2=dh.key_pair_generator()
8          z1=dh.shared_key_generator(d1,Q2)
9          z2=dh.shared_key_generator(d2,Q1)
10     except:
11         print("*** An error has occurred, and a shared key could not be generated ***")
12         raise ValueError("Error during the generation")
13
14     if z1==z2:
15         #print("-Key["+hex(z1)+"] successfully generated")
16         pass
```

```
17      else:
18          print("*** Two different secret key were generated ***"↩
                )
19          raise ValueError("Two different secret key were ↩
                generated")
20
21  #Genera un certo numero di chiavi su di una curva
22  #con w da 2 a 6
23  #e misura il tempo di esecuzione con timeit
24  def bench(curve,times):
25      print("Type of curve: "+curve)
26
27      for w in range(2,7):
28          SETUP='''
29  from __main__ import key_test
30  from dh import ECC_CDH
31  dh=ECC_CDH("'''+curve+'''",wNAF=1,w='''+str(w)+''')'''
32          code='''key_test(dh)'''
33          wNAF=timeit.timeit(setup=SETUP, stmt=code,number=times)
34          print(str(times)+" keys generated using wNAF with w="+↩
                str(w)+" in "+str(wNAF)+" seconds")
35
36
37  def multiple_bench(curve,keyTimes,nBench):
38      for i in range(0,nBench):
39          bench(curve,keyTimes)
40      print("————————————————————————————————————————————————————↩
            ")
41
42  multiple_bench("curves/p192.xml",1000,5)
43  multiple_bench("curves/p224.xml",800,5)
44  multiple_bench("curves/p256.xml",600,5)
45  multiple_bench("curves/p384.xml",400,5)
```

```
multiple_bench("curves/p521.xml",200,5)
```

main_bench.py

```
import timeit

#Testa DH generando simulando lo scambio di chiavi tra due utenti, e controllando che entrambi generino la stessa chiave z
def key_test(dh):
    try:
        d1,Q1=dh.key_pair_generator()
        d2,Q2=dh.key_pair_generator()
        z1=dh.shared_key_generator(d1,Q2)
        z2=dh.shared_key_generator(d2,Q1)
    except:
        print("*** An error has occurred, and a shared key could not be generated ***")
        raise ValueError("Error during the generation")

    if z1==z2:
        #print("-Key["+hex(z1)+"] successfully generated")
        pass
    else:
        print("*** Two different secret key were generated ***")
        raise ValueError("Two different secret key were generated")

#Genera un certo numero di chiavi su di una curva
```

```
22 #prima con double-and-add, poi con wNAF
23 #e misura il tempo di esecuzione con timeit
24 def bench(curve,times):
25     print("Type of curve: "+curve)
26     SETUP='''
27 from __main__ import key_test
28 from dh import ECC_CDH
29 dh=ECC_CDH("'''+curve+'''")'''
30     code='''key_test(dh)'''
31     d_a_a=timeit.timeit(setup=SETUP,stmt=code,number=times)
32     print("*** All keys correctley generated ***")
33     print(str(times)+" keys generated using Double-and-Add in "↩
           +str(d_a_a)+" seconds")
34
35     SETUP='''
36 from __main__ import key_test
37 from dh import ECC_CDH
38 dh=ECC_CDH("'''+curve+'''",wNAF=1)'''
39     code='''key_test(dh)'''
40     wNAF=timeit.timeit(setup=SETUP, stmt=code,number=times)
41     print("*** All keys correctley generated ***")
42     print(str(times)+" keys generated using wNAF in "+str(wNAF)↩
           +" seconds")
43
44     if(wNAF<d_a_a):
45         print("*** WNAF IS FASTER***")
46     else:
47         print("*** DOUBLE AND ADD IS FASTER***")
48     print("————————————————————————————————————————————↩
           ")
49
50 def multiple_bench(curve,keyTimes,nBench):
51     for i in range(0,nBench):
```

```
52          bench(curve,keyTimes)
53      print("----------------------------------------------------------------------
            ")
54
55  multiple_bench("curves/p192.xml",1000,10)
56  multiple_bench("curves/p224.xml",800,10)
57  multiple_bench("curves/p256.xml",600,10)
58  multiple_bench("curves/p384.xml",400,10)
59  multiple_bench("curves/p521.xml",200,10)
```