



# UNIVERSITÀ DEGLI STUDI DI MILANO

**Facoltà di Scienze Matematiche, Fisiche e Naturali**

**Corso di Laurea in Sicurezza dei Sistemi e delle Reti Informatiche**

**LA CRITTOGRAFIA ELLITTICA :  
IMPLEMENTAZIONE E PERFORMANCE NELLE  
TRANSAZIONI WEB SICURE**

Relatrice

Prof.ssa Laura Citrini

Tesi di Laurea di

Massimo Venuto

Matricola n. 698679

Anno Accademico 2007/2008

.....a mio padre *Antonino*

INTRODUZIONE.....	5
1. Introduzione alle curve ellittiche.....	7
1.1 Teoria dei campi.....	7
1.1.1 Campi Finiti.....	8
1.1.2 I Campi Polinomiali .....	9
1.2 Le curve ellittiche.....	10
1.2.1 Leggi di gruppo .....	14
1.2.2 Ordine del gruppo.....	16
1.2.3 Curve ellittiche su $Z_p$ .....	17
1.2.4 Curve ellittiche su $G(F_{2^m})$ .....	18
1.2.5 Problema del logaritmo discreto .....	21
1.3 ECC: Crittografia a curve ellittiche.....	23
1.3.1 Crittosistemi a chiave pubblica : RSA .....	23
1.3.2 Aspetti computazionali dell' algoritmo RSA .....	25
1.3.3 ECDSA (Elliptic Curve Digital Signature Algorithm) .....	26
1.3.4 EC Diffie-Hellman .....	28
1.3.5 Sicurezza ed efficienza: confronto tra crittosistemi a chiave pubblica. ....	28
1.3.5 Standards dell'ECC .....	32
2. Implementazione e performance di ECC nelle transazioni Web sicure.....	34
2.1 Introduzione .....	34
2.1.1 Secure Sockets Layer .....	35
2.1.2 Il Protocollo SSL Record .....	37
2.1.3 Il protocollo Change Cipher Spec .....	38
2.1.4 Il Protocollo Alert.....	39
2.1.5 Il protocollo Handshake .....	39
2.1.6 La crittografia della chiave pubblica in SSL.....	50
2.2 Valutazione delle performance.....	52
2.2.1 Descrizione dell'ambiente.....	52
2.2.2 Specifiche di installazione dell'ambiente.....	53
2.2.3 Standard di misurazione .....	63
2.2.4 Misure dei livelli di performance degli algoritmi .....	63
Conclusioni.....	65
Appendici. ....	66

A. Caratteristica di un campo.....	66
B. Discriminante.....	66
C. Metodo Pollard $\rho$ .....	66
D. Metodo Pohlig-Hellman .....	68
<b>BIBLIOGRAFIA.....</b>	<b>70</b>

## INTRODUZIONE

La crittologia è la scienza che studia come decifrare i testi crittati e le scritture segrete.

Essa indaga le scritture nascoste da una duplice prospettiva:

- da un lato nell'intento di ideare metodi sempre più sicuri per occultare il reale significato di determinati segni (Crittografia);
- dall'altro al fine di decifrare testi occultati senza conoscere a priori il metodo usato per crittografarli (Crittoanalisi).

La Crittografia ha una lunga storia che affonda le sue radici nell'antichità: uno dei primi esempi del suo utilizzo risale al 500 a.C. , quando gli scribi ebrei utilizzarono un codice di sostituzione alfabetica inversa per scrivere parti del Libro di Geremia. Nei secoli successivi è stata frequentemente associata all'impiego militare, ad esempio per proteggere le comunicazioni militari; durante la campagna in Gallia per comunicare con la sua famiglia a Roma Giulio Cesare utilizzava un codice che traslitterava le lettere nella sequenza alfabetica di un numero prefissato di posizioni.

Attualmente per elaborare sistemi crittografici inattaccabili viene sfruttata soprattutto la teoria dei numeri e il continuo aumento delle performance dei computer consente di utilizzare una complessità di calcolo sempre maggiore.

Nel tempo l'elaborazione di metodi crittografici sempre più sofisticati ha stimolato ovviamente un analogo sviluppo anche nella crittoanalisi come studio delle tecniche per violare uno schema crittografico. In teoria qualsiasi schema crittografico potrebbe essere violato procedendo per tentativi ripetuti fino ad indovinarne la chiave di decifrazione, ma nei secoli la crescente complessità raggiunta dalle tecniche di cifratura - fino ad arrivare ai moderni algoritmi crittografici - ha reso del tutto antieconomico e improduttivo il ricorso a metodi così di "basso livello" e ha invece indicato come obiettivo del crittoanalista la ricerca dei punti deboli di algoritmi conosciuti per sfruttarli come via d'accesso al crittosistema, o la ricerca di informazioni su algoritmi crittografici non conosciuti onde poter violare anche i messaggi cifrati prodotti da questi ultimi.

Un crittosistema deve avere due caratteristiche:

- **Sicurezza** il codice deve essere compreso solo dagli interessati e non da eventuali estranei, anche se in possesso di parte di messaggi in chiaro e criptati
- **Efficienza** l'informazione deve essere facilmente ricavabile dagli utenti che sono in possesso delle apposite chiavi.

Consideriamo più nel dettaglio i differenti approcci che possono essere adottati per conseguire

questi due obiettivi.

Due interlocutori che vogliono scambiare delle informazioni in modo sicuro devono criptare i dati per mezzo di una chiave segreta concordemente condivisa e sconosciuta agli eventuali intercettori della comunicazione, detto metodo è conosciuto come **Crittografia simmetrica**. Il problema principale di questa tecnica è che il mittente e il ricevente devono accordarsi su una chiave comune e che devono usare un canale sicuro per scambiarsi quest'informazione, infatti qualsiasi tipo di scambio potrebbe essere intercettato e compromettere quindi la sicurezza della comunicazione.

Proprio questo limite della Crittografia simmetrica ha stimolato lo studio di una nuova metodologia chiamata **Crittografia asimmetrica** e basata su un sistema a **chiave pubblica**. In questo approccio vengono usate da ciascun interlocutore due chiavi, una pubblica e una privata: la prima è resa disponibile a tutti attraverso un repository comune oppure attraverso uno scambio diretto, la seconda invece è tenuta segreta. Alice (mittente) per crittografare il messaggio dovrà usare la chiave pubblica del destinatario (Bob), quest'ultimo per decifrarlo userà la sua chiave privata. Il sistema è creato in modo che sia impossibile recuperare la chiave privata dalla chiave pubblica, solo Bob quindi potrà decifrare il messaggio a lui inviato. Naturalmente perché il sistema funzioni è necessario che Alice usi la chiave pubblica di Bob. Per fare questo si usa una CA (Certification Authority) che garantisce, emette e gestisce le chiavi pubbliche degli utenti, certificando sia la proprietà dell'utente sia l'uso del certificato emesso; questa architettura è chiamata PKI (Public Key Infrastructure).

Il problema dello scambio delle chiavi private e pubbliche è alla base dei moderni sistemi crittografici.

Il primo sistema crittografico a chiave pubblica (basato sull' IFP ossia la fattorizzazione degli interi) è stato introdotto nel 1976 da Whitfield Diffie e Martin Hellmann

Dal 1977 sono stati formalizzati diversi crittosistemi a chiave pubblica, la cui sicurezza è affidata alla difficoltà di risoluzione di differenti problemi matematici.

- Crittosistema RSA (1977)<sup>1</sup> — la sicurezza è basata sulla difficoltà nella fattorizzazione di interi “grandi”
- Crittosistema di Merkle–Hellman Knapsack (1978) — la sicurezza del sistema è basata sulla NP–completezza del problema della somma di sottoinsiemi
  - ❖ Dato un insieme di interi, è possibile enucleare un sottoinsieme di somma

---

<sup>1</sup> Ronald Rivest, Adi Shamir e Leonard Adleman

nulla?

❖ Tutti i crittosistemi di questo tipo si sono rivelati insicuri, tranne il crittosistema di Chor– Rivest;

- Crittosistema di McEliece (1978) — è basato sulla teoria algebrica dei codici, ed è a tutt'oggi ritenuto sicuro; la sicurezza è dovuta alla difficoltà del problema di decodificare un codice lineare (che è NP– completo);
- Crittosistema di ElGamal (1984) — la sicurezza è basata sulla difficoltà del calcolo del logaritmo discreto in campi finiti;
- Crittosistemi basati su curve ellittiche — che traggono origine da sistemi tipo ElGamal, ma operano sulle curve ellittiche piuttosto che sui campi finiti; sono i sistemi più sicuri, anche per chiavi piccole.

I crittosistemi moderni dunque sono basati su problemi matematici intrinsecamente difficili come la fattorizzazione di grandi numeri primi e la difficoltà a trovare il modo matematico di rompere crittosistemi basati sulle funzioni ellittiche

Un algoritmo crittografico viene scelto in funzione della sua robustezza e del costo di computazione che comporta. La crescente richiesta di transazioni sicure per servizi come home banking, e-commerce, VPN – sempre più diffusi - comporta, per l'infrastruttura di Information Technology, un maggiore rapporto costo/prestazioni e costo/affidabilità.

La crittografia ellittica consente, a parità di robustezza, di utilizzare chiavi crittografiche più piccole di un fattore 10 e quindi permette un costo computazionale più basso.

Per questo oggi insieme agli algoritmi RSA è uno dei sistemi crittografici a chiave pubblica più utilizzati.

Nei prossimi capitoli descriveremo le strutture matematiche che sono alla base di questo particolare ambito della crittografia e le principali differenze tra questo approccio crittografico e quello fondato sugli algoritmi RSA, quindi illustreremo concretamente un'applicazione pratica della crittografia ellittica nel settore della comunicazione informatica sicura e più precisamente nell'ambito delle transazioni sicure via Web.

## **1. Introduzione alle curve ellittiche**

### **1.1 Teoria dei campi**

La teoria dei campi è una branca della matematica che studia le proprietà dei campi. Un campo è una struttura algebrica a due operazioni convenzionalmente indicate come addizione e moltiplicazione.

Le proprietà sono quelle di gruppo abeliano per la struttura additiva e di gruppo abeliano anche per quella moltiplicativa, qualora si tolga all'insieme l'elemento neutro rispetto alla somma.

Al nostro ambito di interesse pertiene soprattutto lo studio dei campi finiti.

### 1.1.1 Campi Finiti

Un campo finito  $F$  è un campo che contiene un numero finito di elementi. L'ordine di un campo finito è il numero di elementi che formano il campo. Se  $q$  è un primo grande, esiste un solo campo finito di ordine  $q$  (a meno di isomorfismi), denotato con  $F_q$ . In generale, infatti, tutti i campi finiti dello stesso ordine sono isomorfi.

Se  $q = p^m$  dove  $p$  è un primo ed  $m$  un intero positivo, allora  $p$  è chiamato *caratteristica* di  $F_q$ , mentre  $m$  è chiamato *grado di estensione* di  $F_q$ . Molti standard di tecniche per sistemi crittografici basati sul problema del logaritmo discreto su curve ellittiche, restringono l'ordine del campo o ad un primo dispari ( $q = p$ ) oppure a una potenza di 2 ( $q = 2^n$ ).

Poiché tutti i campi dello stesso ordine sono isomorfi, una possibile rappresentazione degli elementi di tale campo, nel caso in cui  $q$  è un numero primo è quella del campo dei resti modulo  $p$ , indicato usualmente con  $Z_p$  in cui gli elementi sono indicati con:  $\{0, 1, 2, \dots, p - 1\}$  e su di essi sono definite le seguenti operazioni:

#### **Addizione Modulare:**

Se  $a, b \in F_p$ , allora  $a + b = r$  dove  $r$  è il resto della divisione di  $a + b$  per  $p$  e di conseguenza risulta  $0 \leq r \leq p - 1$ .

Il neutro rispetto all'addizione è 0, l'inverso di un elemento  $a$  rispetto alla somma (opposto) è  $-a$  cioè  $p - a$  (infatti  $a + p - a = p \equiv 0 \pmod{p}$ ).

#### **Moltiplicazione Modulare:**

Se  $a, b \in F_p$ , allora  $a * b \equiv s \pmod{p}$ , cioè  $s$  è il resto della divisione di  $a * b$  per  $p$ .

Il neutro rispetto alla moltiplicazione è 1. L'inverso rispetto alla moltiplicazione, che esiste solo se  $a$  è diverso da 0 in  $F_p$ , denotato con  $a^{-1}$ , è l'unico intero  $c \in F_p$  per il quale risulta  $a * c \equiv 1 \pmod{p}$ .

Consideriamo come esempio il campo finito  $F_p$ , con  $p = 17$  ordine del campo. Gli elementi di tale campo sono:  $\{0, 1, 2, \dots, 16\}$

Su questo particolare campo  $F_{17}$  mostriamo alcuni esempi delle operazioni sopra definite :

*Addizione modulare*  $12 + 16 = 11$                        $((12 + 16) = 28 \text{ e } 28 \pmod{17} = 11)$

*Moltiplicazione modulare*  $8 * 10 = 12$                        $((8*10)=80 \text{ e } 80 \pmod{17} = 12)$

Inverso di 2:  $2^{-1} = 9$

$$((2*9)=18 \text{ e } 18 \text{ mod } 17 = 1)$$

Nel caso invece di  $q = 2^m$  il campo  $F_{2^m}$ , *chiamato campo finito binario*, con  $2^m$  elementi, può essere visto come un vettore di dimensione  $m$  sul campo  $Z_2$ , costituito dai due elementi 0 e 1. In  $F_{2^m}$  esistono  $m$  elementi  $\beta_0, \beta_1, \beta_2, \dots, \beta_{m-1}$  tali che ogni elemento  $\beta \in F_{2^m}$  può essere scritto univocamente nella forma:

$$\beta = a_0\beta_0 + a_1\beta_1 + a_2\beta_2 + \dots + a_{m-1}\beta_{m-1}$$

con  $a_i \in \{0, 1\}$ . L'addizione e la moltiplicazione dipendono dal tipo di rappresentazione utilizzato per gli elementi del campo. L'insieme  $(\beta_0, \beta_1, \beta_2, \dots, \beta_{m-1})$  è chiamato base di  $F_{2^m}$  su  $Z_2$ . In sostanza si introduce l'isomorfismo esistente tra ogni spazio vettoriale e le  $m$ -ple di elementi del campo soggiacente.

### 1.1.2 I Campi Polinomiali

Introduciamo come esempio il campo polinomiale su  $Z_2$

Sia  $f(x) = x^m + c_{m-1}x^{m-1} + \dots + c_1x + c_0$  (dove  $c_i \in \{0, 1\}$ ) un polinomio irriducibile di grado  $m$  su  $F_2$ , cioè tale che non può essere fattorizzato come un prodotto di due polinomi su  $F_2$ , ognuno di grado minore di  $m$ . Il polinomio  $f(x)$  è chiamato *polinomio ridotto o polinomio caratteristico*.

Il campo finito  $F_{2^m}$  è formato da tutti i polinomi su  $F_2$  di grado minore di  $m$ :

$$F_{2^m} = \{a_{m-1}x^{m-1} + \dots + a_1x + a_0 : a_i \in \{0,1\}\}$$

C'è isomorfismo tra l'anello dei polinomi di grado  $m + 1$  e le  $m$ -ple dei loro coefficienti. Gli elementi del campo possono quindi essere denotati da una stringa di bit  $(a_{m-1}, \dots, a_0)$  di lunghezza  $m$ , tale che:

$$F_{2^m} = \{(a_{m-1}, \dots, a_1, a_0) : a_i \in \{0,1\}\}$$

Di conseguenza gli elementi di  $F_{2^m}$  possono essere rappresentati dall'insieme di tutte le stringhe binarie di lunghezza  $m$ .

L'elemento neutro rispetto alla moltiplicazione è rappresentato dalla stringa di bit (00.....01) mentre l'elemento neutro rispetto all'addizione è rappresentato dalla stringa (00.....0).

Definiamo le operazioni su  $F_{2^m}$  quando è utilizzata la rappresentazione sotto forma di stringhe:

#### **Addizione:**

Siano  $a = (a_{m-1}, \dots, a_0)$  e  $b = (b_{m-1}, \dots, b_0) \in F_{2^m}$ , allora si definisce  $a + b = c$  dove  $c = (c_{m-1}, \dots, c_0)$  con  $c_i = (a_i + b_i) \text{ mod } 2$ .

**Moltiplicazione:**

Se  $a = (a_{m-1}, \dots, a_0)$  e  $b = (b_{m-1}, \dots, b_0) \in F_{2^m}$ , allora  $a * b = r$  con  $r = (r_{m-1}, \dots, r_0)$

Tale  $m$ -pla è ottenuta considerando il polinomio  $(r_{m-1} * x_{m-1} + \dots + r_1 * x + r_0)$ , resto della divisione del prodotto  $(a_{m-1}x^{m-1} + \dots + a_1x + a_0) * (b_{m-1}x^{m-1} + \dots + b_1x + b_0)$  per  $f(x)$  su  $F_2$ .

Come esempio di campo polinomiale esaminiamo il caso  $F_{2^4}$ .

Sia  $f(x) = x^4 + x + 1$  il polinomio ridotto. I 16 elementi di  $F_{2^4}$ , nei due modi di scrittura, cioè come polinomi o come stringhe di bit, sono:

0	(0000)	1	(0001)	x	(0010)	x + 1	(0011)
x <sup>2</sup>	(0100)	x <sup>2</sup> + 1	(0101)	x <sup>2</sup> + x	(0110)	x <sup>2</sup> + x + 1	(0111)
x <sup>3</sup>	(1000)	x <sup>3</sup> + 1	(1001)	x <sup>3</sup> + x	(1010)	x <sup>3</sup> + x + 1	(1011)
x <sup>3</sup> + x <sup>2</sup>	(1100)	x <sup>3</sup> + x <sup>2</sup> + 1	(1101)	x <sup>3</sup> + x <sup>2</sup> + x	(1110)	x <sup>3</sup> + x <sup>2</sup> + x + 1	(1111)

Vediamo un esempio delle operazioni:

**-Addizione**

$$[(x^3 + x^2 + 1) + (x^3 + 1)] \text{ mod } 2 = x^2 \text{ e quindi } (1101) + (1001) = (0100)$$

**-Moltiplicazione**

$$(x^3 + x^2 + 1) * (x^3 + 1) = (x^6 + x^5 + x^2 + 1) \text{ mod } (x^4 + x + 1) = (x^3 + x^2 + x + 1)$$

$$(1101) * (1001) = (1111)$$

**1.2 Le curve ellittiche**

Lo studio delle Curve Ellittiche, che spazia in vari campi della matematica, in particolare in Geometria, Algebra e nel campo della Teoria dei Numeri, risale al XIX secolo; detti studi sono serviti a risolvere molti problemi di varia natura come *il problema dei numeri congruenti*<sup>2</sup> e la fattorizzazione di numeri interi.

Le curve ellittiche sono molto importanti nella teoria dei numeri e ne costituiscono il maggior campo di ricerca attuale. Per esempio recentemente il matematico Andrew Wiles (con l'assistenza di Richard Taylor) utilizzando un'avanzata teoria delle Curve Ellittiche ha dimostrato l'ultimo Teorema di Fermat [6].

Le curve ellittiche non hanno niente a che vedere con le ellissi. Si chiamano in questo modo poiché sono descritte da equazioni cubiche, simili a quelle utilizzate per calcolare la lunghezza di un'ellisse. Di seguito vedremo una formulazione dettagliata di dette equazioni.

Preso un campo  $K$  generico si può definire un campo finito  $GF_q$ , con  $q$  potenza di un primo.

<sup>2</sup> Un intero positivo  $n$  è detto numero congruente se è uguale all'area di un triangolo rettangolo avente lati di lunghezza razionale ossia esistono numeri razionali positivi  $a, b$  e  $c$  tali che  $a^2 + b^2 = c^2$  e  $1/2 ab = n$ .

Ricordiamo che tutti i campi dello stesso ordine finito sono isomorfi.

Una curva ellittica  $E$  sul campo  $K$  è una curva “liscia” definita dall’**Equazione Generalizzata di Weierstrass**:

$$E: y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \quad [1.1]$$

Dove  $a_1, a_2, a_3, a_4, a_6 \in K$  e  $\Delta \neq 0$ , dove il  $\Delta$  è il *discriminante* (vedi Appendice B) di  $E$  ed è definito come:

$$\Delta = -d_2^2 d_8 - 8 d_4^3 - 27 d_6^2 + 9 d_2 d_4 d_6$$

in cui si è posto:

$$d_2 = a_1^2 + 4 a_2$$

$$d_4 = 2 a_4 + a_1 a_3$$

$$d_6 = a_1^2 + 4 a_6$$

$$d_8 = a_1^2 a_6 + 4 a_2 a_6 - a_1 a_3 a_4 + a_2 a_3^2 - a_4^2.$$

Il termine “liscia” significa che la curva non possiede punti di singolarità ossia punti dove le derivate parziali si annullano contemporaneamente (ed è chiamata di conseguenza *non-singolare*).

Si dimostra che  $\Delta \neq 0$  garantisce che la curva ellittica sia *non-singolare*.

Per i nostri scopi considereremo le sole curve ellittiche con una equazione della forma (1.1) compreso il loro “punto all’infinito” del quale verrà data spiegazione più avanti (dato il tipo di equazione cubica considerata e le condizioni imposte, la curva ammette un unico punto improprio reale), quindi scriveremo :

$$E(K) : \{(x, y) \in K \times K / y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6\} \cup \{\infty\}.$$

È possibile semplificare l’equazione di Weierstrass nel modo seguente. Consideriamo due curve ellittiche  $E_1$  ed  $E_2$  a coefficienti in  $K$  date dalle seguenti equazioni di Weierstrass :

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

$$E_2: y^2 + b_1xy + b_3y = x^3 + b_2x^2 + b_4x + b_6$$

Esse sono *isomorfe* su  $K$  se esistono  $u, r, s, t \in K$ , con  $u \neq 0$ , tale che il cambio di variabili

$$(x, y) \rightarrow (u^2 x + r, u^3 y + u^2 s x + t) \quad (1.2)$$

trasforma l’equazione  $E_1$  nell’equazione  $E_2$ . Detta trasformazione (1.2) è chiamata *trasformazione di variabili ammissibile*.

Analizzeremo ora la semplificazione dell'equazione di Weierstrass a seconda che la caratteristica del campo<sup>3</sup>  $K$ , indicata con  $\text{Char}(K)$ , sia diversa o uguale ai due valori 2 e 3 .

Se  $\text{Char}(K) \neq 2, 3$  allora il seguente cambiamento di variabili

$$(x, y) \rightarrow \left( \frac{x - 3a_2^1 - 12a_2}{36}, \frac{y - 3a_1 x - a_1^3 + 4a_1 a_2 - 12a_3}{24} \right)$$

trasforma  $E$  nella curva di equazione

$$y^2 = x^3 + a x + b$$

dove  $a, b \in K$ .

Consideriamo il significato della non-singularità di questa curva, che in questo caso è ben preciso. Prendiamo l'equazione implicita  $F(x,y) = y^2 - f(x)$  dove  $f(x) = x^3 + a x + b$ ; la condizione di non singularità impone che non esista nessun punto di  $E$  in cui le derivate parziali

$$\frac{\partial F}{\partial x} = -f'(x), \text{ e } \frac{\partial F}{\partial y} = 2y$$

si annullino contemporaneamente. Calcolando la pendenza di una retta tangente  $\frac{dy}{dx}$  a  $E$

consideriamo l'equazione  $y^2 = f(x)$ ; derivando si ha  $2y \frac{dy}{dx} = f'(x)$  da cui

$$\frac{dy}{dx} = \frac{f'(x)}{2y} = - \frac{\frac{\partial F}{\partial x}}{\frac{\partial F}{\partial y}}$$

Possiamo considerare il caso in cui il numeratore sia diverso da zero e il denominatore uguale a zero come una retta parallela all'asse delle  $y$ , ma se entrambe le derivate parziali si annullano l'equazione perde di significato. A tale scopo consideriamo per ipotesi che in un punto  $P \equiv (x_0, y_0)$  le derivate parziali si annullino, ne consegue che  $f'(x_0) = 2y_0 = 0$ , da cui  $2y_0 = 0$ , ma allora essendo  $y^2 = f(x)$  si ha anche  $f(x_0) = 0$ . Si dimostra che un polinomio  $f(x)$  ammette radice multipla  $r$  se e solo se  $r$  è radice di  $f(x)$  e di  $f'(x)$ , quindi l'annullamento delle derivate parziali nel punto  $P \equiv (x_0, y_0)$  implica che  $x_0$  sia una radice multipla di  $f(x)$  e che quindi il discriminante di  $f(x)$  sia nullo.

Essendo  $f(x) = x^3 + a x + b$  il discriminante (vedi appendice B) è  $-(4a^3 + 27b^2)$  possiamo quindi affermare che:

**Una curva ellittica  $E$  sul campo  $K$ , avente caratteristica diversa da 2 e 3, è definita dall'equa-**

---

<sup>3</sup> Vedi appendice A.

## zione di Weierstrass

$$y^2 = x^3 + a x + b$$

dove  $a, b \in K$  e soddisfano la disuguaglianza  $-(4a^3 + 27b^2) \neq 0$

2. Se  $\text{Char}(K) = 2$ , ci sono due casi da considerare.

Se  $a_1 \neq 0$ , allora un'ammissibile cambio delle variabili

$$(x, y) \rightarrow \left( a_1^2 x, \frac{a_3}{a_1}, a_1^3 y + \frac{a_1^2 a_4 + a_3^2}{a_1^3} \right)$$

trasforma  $E$  nella curva di equazione

$$y^2 + xy = x^3 + a x^2 + b$$

dove  $a, b \in K$ . Tale curva è detta *non-supersingolare* e il discriminante  $\Delta = b$ .

Se  $a_1 = 0$  allora una ammissibile cambio di variabili è

$$(x, y) \rightarrow (x + a_2, y)$$

L'equazione della curva  $E$  diventa

$$y^2 + cy = x^3 + a x^2 + b$$

dove  $a, b, c \in K$ . Questa curva è detta *supersingolare* e ha  $\Delta = c^4$ .

3. Se la caratteristica di  $K$  è 3, allora ci sono due casi da considerare:

Se  $a_1^2 \neq -a_2$ , allora un'ammissibile cambio di variabili è:

$$(x, y) \rightarrow \left( x + \frac{d_4}{d_2}, y + a_1 x + a_1 \frac{d_4}{d_2} + a_3 \right)$$

Dove  $d_2 = a_1^2 + a_2$  and  $d_4 = a_4 - a_1 a_3$ , la curva  $E$  ha quindi equazione

$$y^2 = x^3 + a x^2 + b$$

dove  $a, b \in K$ .

Questa curva è chiamata *non-supersingolare* ed ha discriminante  $\Delta = -a^3 b$ .

Se invece  $a_1^2 = -a_2$ , allora può essere ammissibile un cambio delle variabili formato del tipo:

$$(x, y) \rightarrow (x, y + a_1 x + a_3)$$

che trasforma  $E$  nella curva di equazione

$$y^2 = x^3 + a x + b$$

dove  $a, b \in K$ . Questa curva è chiamata *supersingolare* ed ha discriminante  $\Delta = -a^3$ .

Quando abbiamo definito la curva ellittica è stato considerato cosiddetto anche il punto all'infinito  $\Theta$  della cubica stessa. Nel piano proiettivo complesso (che a noi in generale non interessa) ogni cubica ha tre punti all'infinito, di coordinate reali o complesse a seconda del caso. La cubica nella

forma di Weierstrass ha un solo punto improprio reale, che è il punto all'infinito dell'asse  $y$ , per cui una retta passante per un punto  $P$  e il punto  $\Theta$ , risulta la retta parallela all'asse delle  $y$  passante per il punto  $P$ .

Nel paragrafo successivo si vedrà l'impiego di tale punto per i nostri scopi.

### 1.2.1 Leggi di gruppo

Una caratteristica fondamentale dell'insieme dei punti della cubica ellittica di Weierstrass  $E(K)$  su un campo finito  $K$  è quella di definire una struttura algebrica su tale insieme. Nello specifico si può definire un'operazione di *addizione* “ + “ sui punti di  $E(K)$  in modo che  $(E(K), +)$  sia un gruppo abeliano.

Questo tipo di gruppo è alla base dell'utilizzo dei critto sistemi basati sulle curve ellittiche.

Sia dunque  $E$  una curva ellittica definita sopra un campo  $K$ . Possiamo costruire una regola di addizione tra i punti della curva (detta *regola della corda tangente*) partendo da due punti della curva per ottenerne un terzo. Vediamo come:

Siano  $P \equiv (x_1, y_1)$  e  $Q \equiv (x_2, y_2)$  due punti distinti della curva ellittica  $E$ , la somma  $R = P + Q$  è definita nel seguente modo: tracciata una retta che passa per  $P$  e per  $Q$ , tale retta intersecherà la curva in un terzo punto  $R'$ . Si individua, quindi, il punto simmetrico  $R$  di  $R'$  rispetto all'asse  $x$  (il punto  $R$  avrà quindi la stessa ascissa di  $R'$  e ordinata opposta).

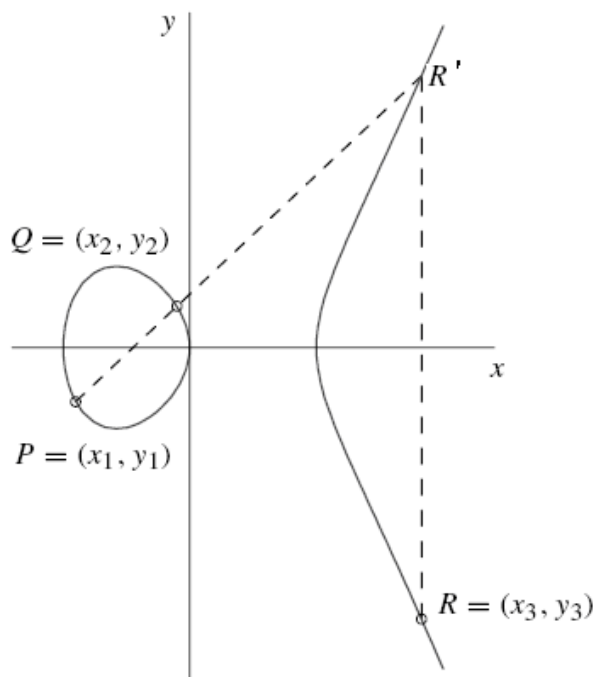


Figura 1

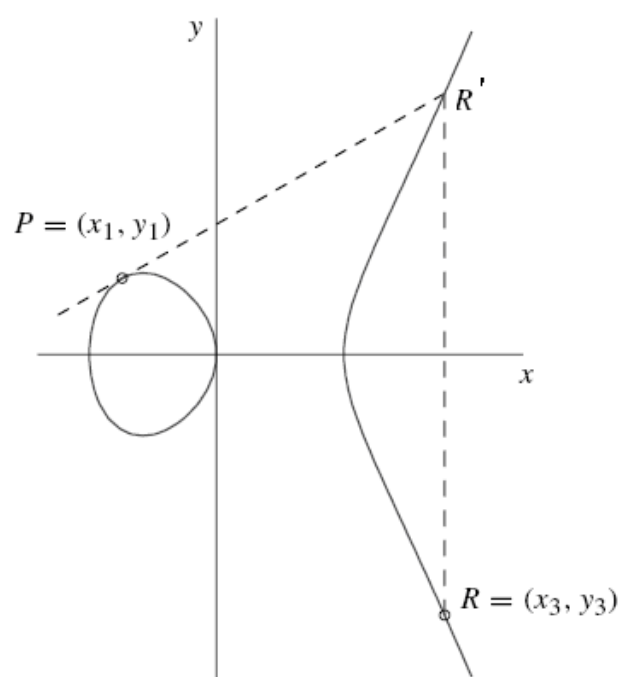


Figura 2

La *regola del raddoppio* (ossia quella che si ottiene sommando un punto  $P$  a se stesso) invece viene definita come segue: tracciata la retta tangente alla curva in  $P$ , questa interseca la curva in un punto  $R'$ , trovo similmente a prima il punto simmetrico rispetto all'asse delle  $x$  e ottengo  $R = P + P$ .

Nelle figure 1 e 2 si può vedere il procedimento descritto sopra nel caso  $K = \mathbb{R}$ :

Vediamo quindi di illustrare in questo caso gli assiomi di gruppo distinguendoli secondo la caratteristica del campo  $K$  usato per definire la cubica ellittica.

### A. Nel caso $\text{char}(K) \neq 2, 3$

1. **Addizione.** Siano  $P \equiv (x_1, y_1)$  e  $Q \equiv (x_2, y_2)$  due punti di  $E(K)$  con  $P \neq \pm Q$ .

$$\text{Si definisce } P + Q \equiv (x_3, y_3) = \left( \left( \frac{y_2 - y_1}{x_2 - x_1} \right)^2 - x_1 - x_2, \left( \frac{y_2 - y_1}{x_2 - x_1} \right) (x_1 - x_3) - y_1 \right)$$

2. **Identità.** Il punto di  $E(K)$  che "funziona" da punto neutro rispetto all'addizione è il punto all'infinito di  $E(K)$ , infatti  $P + \Theta = \Theta + P = P$  per ogni  $P \in E(K)$ . Infatti sommare  $P$  e  $\Theta$  significa tracciare la retta parallela all'asse  $y$  passante per  $P$  e per la simmetria della curva, il risultato è ancora il punto  $P$ .

3. **Opposto** (cioè inverso rispetto all'addizione). Se  $P \equiv (x, y) \in E(K)$ , allora risulta che il punto indicato da  $-P$  di coordinate  $(x, -y)$  è tale che  $(x, y) + (x, -y) = \Theta$ . È da notare che  $-P$  è davvero un punto in  $E(K)$ , per la simmetria della cubica rispetto all'asse  $x$ . Inoltre, come previsto,  $-\Theta = \Theta$

4. Anche se non rientra negli assiomi di gruppo, ricordiamo che il punto doppio di un punto  $P \equiv (x_1, y_1) \in E(K)$ , dove  $P \neq -P$ , è il punto indicato con  $2P \equiv (x_3, y_3)$ , dove

$$x_3 = \left( \frac{3x_1^2 + a}{2y_1} \right)^2 - 2x_1 \quad \text{e} \quad y_3 = \left( \frac{3x_1^2 + a}{2y_1} \right) (x_1 - x_3) - y_1$$

### B. Nel caso di una curva *non-supersingolare* $E(F_{2^m})$ di equazione $y^2 + xy = x^3 + ax^2 + b$

1. **Addizione.** Sia  $P \equiv (x_1, y_1) \in E(F_{2^m})$  e  $Q \equiv (x_2, y_2) \in E(F_{2^m})$  dove  $P \neq \pm Q$  si definisce

$$P + Q = (x_3, y_3), \text{ dove } x_3 = \lambda^2 + \lambda + x_1 + x_2 + a \quad \text{e} \quad y_3 = \lambda(x_1 + x_3) + x_3 + y_1 \quad \text{con } \lambda = \left( \frac{y_2 - y_1}{x_2 - x_1} \right).$$

2. **Identità.** Come nel caso precedente, il punto improprio è l'elemento neutro, infatti:

$$P + \Theta = \Theta + P = P \text{ per ogni } P \in E(F_{2^m})$$

3. **Opposto.** Se  $P \equiv (x, y) \in E(F_{2^m})$  risulta  $-P \equiv (x, x+y)$  infatti è  $(x, y) + (x, x+y) = \Theta$ . Come nel caso precedente  $-P$  è davvero un punto in  $E(F_{2^m})$  e  $-\Theta = \Theta$ .

4. Il punto doppio di  $P \equiv (x_1, y_1) \in E(F_{2^m})$ , dove  $P \neq -P$ , è il punto  $2P = (x_3, y_3)$ , con

$$x_3 = \lambda^2 + \lambda + a = x_1^2 + \frac{b}{x_1^2} \text{ e } y_3 = x_1^2 + \lambda x_3 + x_3 \text{ con } \lambda = x_1 + \frac{y_1}{x_1}$$

**C. Nel caso di una curva supersingolare**  $E(F_{2^m})$  di equazione  $y^2 + cy = x^3 + ax^2 + b$

1. **Addizione** Sia  $P \equiv (x_1, y_1) \in E(F_{2^m})$  e  $Q \equiv (x_2, y_2) \in E(F_{2^m})$  dove  $P \neq \pm Q$ . Si definisce

$$P + Q \equiv (x_3, y_3), \text{ dove } x_3 = \lambda^2 + x_1 + x_2 \text{ e } y_3 = \lambda(x_1 + x_3) + y_1 + c \text{ con } \lambda = \frac{(y_1 + y_2)}{(x_1 + x_2)}.$$

2. **Identità.** Anche in questo caso  $P + \Theta = \Theta + P = P$  per ogni  $P \in E(F_{2^m})$

3. **Opposto.** Se  $P \equiv (x, y) \in E(F_{2^m})$  allora  $-P \equiv (x, y + c)$ , infatti  $(x, y) + (x, y + c) = \Theta$  e valgono le considerazioni fatte negli altri casi.

4. **Punto doppio.** Sia  $P = (x_1, y_1) \in E(F_{2^m})$ , dove  $P \neq -P$ . Allora  $2P \equiv (x_3, y_3)$ , dove

$$x_3 = \left( \frac{x_1^2 + a}{c} \right)^2 \text{ e } y_3 = \left( \frac{x_1^2 + a}{c} \right)^2 (x_1 + x_3) + y_1 + c.$$

Nelle applicazioni crittografiche vengono utilizzate le curve ellittiche in campo  $E(2^m)$ , dove  $m$  risulta essere un numero grande (lo standard prevede  $m = 163.233.283.409.571$ ).

### 1.2.2 Ordine del gruppo

Ai fini di una buona realizzazione di un algoritmo crittografico una delle caratteristiche più importante delle curve ellittiche è determinare il numero di punti che appartengono ad una generica curva ellittica costruita su un campo finito  $G(F_q)$ . Tale grandezza si indica con  $\#E(GF(q))$  o più

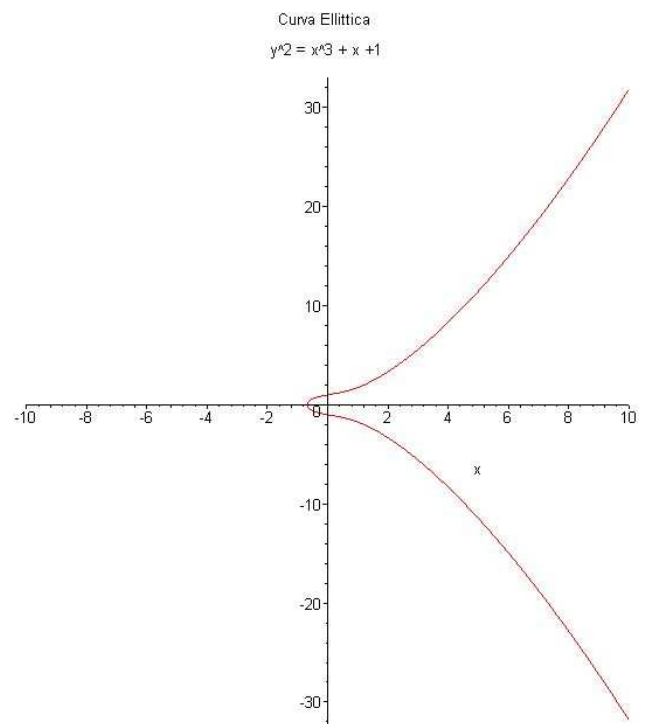


Figura 3  $y^2 = x^3 + x + 1$  nel campo Reale.

semplicemente  $\#E$ , ed è detta *ordine della curva*  $E/GF(q)$ .

Per esempio consideriamo la curva  $E/(GF(5))$  descritta dall'equazione  $y^2 = x^3 + x + 1$  rappresentata nella figura a fianco.

Per contare i punti di  $E$ , possiamo comporre una lista di possibili valori di  $x$ , poi di  $x^3 + x + 1 \pmod{5}$  e di  $y$ .

$x$	$x^3 + x + 1$	$y$	Punti
0	1	$\pm 1$	(0,1), (0,4)
1	3	-	
2	1	$\pm 1$	(2,1),(2,4)
3	1	$\pm 1$	(3,1),(3,4)
4	1	$\pm 2$	(4,2),(4,3)

Bisogna poi considerare anche il punto improprio, quindi,  $E(F_5)$  ha ordine 9.

Per valori molto grandi di  $q$  non è possibile utilizzare questo metodo, possiamo però usare il teorema di Hasse, di seguito esposto, che fornisce dei limiti precisi per  $\#E$ :

Sia  $E$  una curva ellittica definita sul campo  $GF(q)$ , l'ordine di  $E$  è tale che:

$$q + 1 - 2\sqrt{q} \leq \#E \leq q + 1 + 2\sqrt{q}$$

L'intervallo  $[q + 1 - 2\sqrt{q}, q + 1 + 2\sqrt{q}]$  è chiamato *intervallo di Hasse*.

Un'alternativa formulazione del Teorema di Hasse è la seguente: sia  $E$  una curva ellittica definita su  $Fq$ , allora l'ordine di  $E$  soddisfa la relazione

$$\#E(Fq) = q + 1 - t$$

dove  $|t| \leq 2\sqrt{q}$  è chiamata *traccia* di  $E$  su  $Fq$ .

### 1.2.3 Curve ellittiche su $Z_p$

La crittografia ellittica utilizza curve le cui variabili e i cui coefficienti sono ristretti agli elementi di un campo finito. Nelle applicazioni crittografiche vengono utilizzate due famiglie ellittiche: *curve prime* su  $Z_p$  e *curve binarie* su  $G(F_{2^m})$ . Come **curva prima** su  $Z_p$  si utilizza l'equazione cubica già esposta precedentemente nella quale le variabili e i coefficienti assumono valori nell'insieme di

interi da 0 a  $p - 1$  e i calcoli vengono effettuati modulo  $p$ . Le variabili e i coefficienti della **curva binaria** definita su  $G(F_{2^m})$  assumono valori in  $G(F_{2^m})$  e i calcoli sono effettuati in  $G(F_{2^m})$ . Fernandes <sup>4</sup> evidenzia che le curve prime sono le migliori per le applicazioni software in quanto non richiedono le numerose operazioni sui bit necessarie nel caso delle curve binarie; al contrario le curve binarie sono migliori per le applicazioni hardware dove bastano pochi elementi logici per creare un sistema crittografico veloce e potente.

Per le curve ellittiche si  $Z_p$ , come nel caso dei numeri reali, ci si limita alle equazioni della forma  $y^2 = x^3 + a x + b$  ma in questo caso con coefficienti e variabili limitati a  $Z_p$ :

$$y^2 \bmod p = (x^3 + a x + b) \bmod p \quad [1.3]$$

per esempio (cfr [4]), l'equazione di cui sopra è soddisfatta per  $a = 1, b = 1, x = 9, y = 7, p = 23$ .

$$7^2 \bmod 23 = (9^3 + 9 + 1) \bmod 23$$

$$49 \bmod 23 = 739 \bmod 23$$

$$3 = 3$$

Considerando l'insieme  $E_p(a, b)$  costituito da tutte le coppie di interi  $(x, y)$  che soddisfano l'equazione [1.3], insieme al punto all'infinito precedentemente introdotto, i coefficienti  $a$  e  $b$  sono tutti elementi di  $Z_p$ .

#### 1.2.4 Curve ellittiche su $G(F_{2^m})$

Come già esposto, un campo finito  $G(F_{2^m})$  è costituito da  $2^m$  elementi con le operazioni di somma e moltiplicazione che possono essere definite sui polinomi. Per le curve ellittiche su  $G(F_{2^m})$ , si usa l'equazione cubica, già analizzata, in cui le variabili e i coefficienti assumono i valori contenuti in  $G(F_{2^m})$  per qualche numero  $m$ , e in cui i calcoli vengono eseguiti utilizzando le regole dell'aritmetica in  $G(F_{2^m})$ .

L'equazione della cubica usata è:

$$y^2 + xy = x^3 + a x^2 + b \quad [1.4]$$

dove  $x, y, a, b$  appartengono a  $G(F_{2^m})$  e i calcoli vengono eseguiti in  $G(F_{2^m})$ .

Consideriamo ora l'insieme di  $E_{2^m}(a, b)$  costituito da tutte le coppie di interi  $(x, y)$  che soddisfino l'equazione [1.4] insieme al punto all'infinito  $\Theta$ . Utilizziamo per esempio il campo  $G(F_{2^4})$  con il

---

<sup>4</sup> Fernandes, A. "Elliptic Curve Cryptography." *Dr. Dobb's Journal*, December 1999.

polinomio irriducibile  $f(x) = x^4 + x + 1$  già usato nel paragrafo 1.1.2 di cui riporto, per comodità la tabella usata.

0	(0000)	1	(0001)	x	(0010)	x + 1	(0011)
$x^2$	(0100)	$x^2 + 1$	(0101)	$x^2 + x$	(0110)	$x^2 + x + 1$	(0111)
$x^3$	(1000)	$x^3 + 1$	(1001)	$x^3 + x$	(1010)	$x^3 + x + 1$	(1011)
$x^3 + x^2$	(1100)	$x^3 + x^2 + 1$	(1101)	$x^3 + x^2 + x$	(1110)	$x^3 + x^2 + x + 1$	(1111)

Se cerchiamo un generatore di detto campo si può osservare che prendendo  $g = x = 0010$  si sviluppano, le potenze di  $g$  (ci si è serviti, per il calcolo, di Maple) a partire dalla potenza  $g^4$  (per i casi precedenti non c'è bisogno di aiuto),:

```

> rem(x^4,x^4+x+1,x) mod 2;          g^4
                                     1 + x
> rem(x+x^2,x^4+x+1,x) mod 2;      g^5
                                     x + x^2
> rem(x^3+x^2,x^4+x+1,x) mod 2;    g^6
                                     x^3 + x^2
> rem(x^4+x^3,x^4+x+1,x) mod 2;    g^7
                                     x^3 + 1 + x
> rem(x^4+x^2+x,x^4+x+1,x) mod 2;  g^8
                                     x^2 + 1
> rem(x^3+x,x^4+x+1,x) mod 2;      g^9
                                     x^3 + x
> rem(x^4+x^2,x^4+x+1,x) mod 2;    g^10
                                     x^2 + 1 + x
>
> rem(x^3+x+x^2,x^4+x+1,x) mod 2;  g^11
                                     x^3 + x + x^2
> rem(x^4+x^2+x^3,x^4+x+1,x) mod 2; g^12
                                     x^3 + x^2 + 1 + x
> rem(x^4+x^2+x^3+x,x^4+x+1,x) mod 2; g^13
                                     x^3 + x^2 + 1
> rem(x^4+x^3+x,x^4+x+1,x) mod 2;  g^14
                                     x^3 + 1
> rem(x^4+x,x^4+x+1,x) mod 2;      g^15
                                     1

```

La funzione  $\text{rem}(p1, p2, x)$  utilizzata calcola il resto di  $p1$  diviso per  $p2$ , modulo il numero assegnato, con  $x$  come incognita.

Si costruisce di conseguenza la seguente tabella:

$g^0 = 0001$	$g^4 = 0011$	$g^8 = 0101$	$g^{12} = 0001$
$g^1 = 0010$	$g^5 = 0110$	$g^9 = 1010$	$g^{13} = 0001$
$g^2 = 0100$	$g^6 = 1100$	$g^{10} = 0111$	$g^{14} = 0001$
$g^3 = 1000$	$g^7 = 1011$	$g^{11} = 1110$	$g^{15} = 0001$

Consideriamo ora la curva ellittica  $y^2 + xy = x^3 + g^4 x^2 + 1$ .

Quindi in questo caso si è posto  $a = g^4$  e  $b = g^0 = 1$ . Un punto che soddisfa l'equazione è

$$(g^3)^2 + (g^5)(g^3) = (g^5)^3 + (g^4)(g^5)^2 + 1$$

Da cui

$$g^6 + g^8 = g^{15} + g^{14} + 1$$

ossia

$$1100 + 0101 = 0001 + 1000 + 0001$$

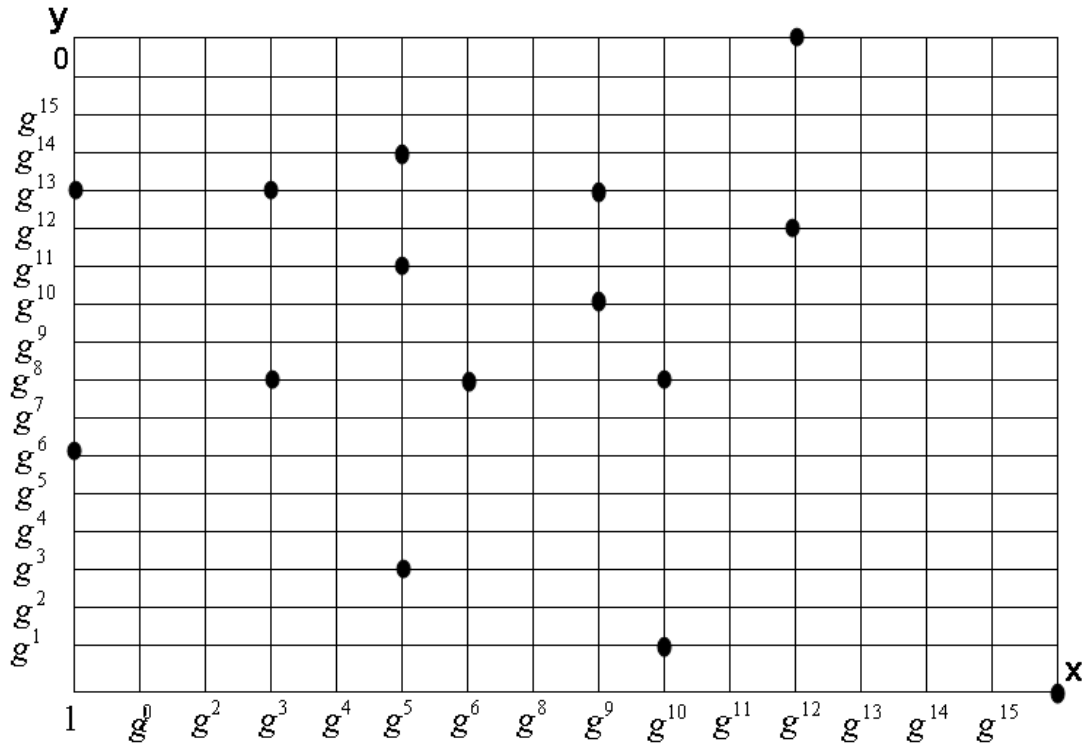
infine

$$1001 = 1001$$

Nella seguente tabella sono elencati i punti che fanno parte di  $E_{2^4}(g^4, 1)$ .

$(0,1)$	$(g^5, g^3)$	$(g^9, g^{13})$
$(1, g^6)$	$(g^3, g^8)$	$(g^{10}, g)$
$(1, g^{13})$	$(g^6, g^8)$	$(g^{10}, g^8)$
$(g^3, g^8)$	$(g^6, g^{14})$	$(g^{12}, 0)$
$(g^3, g^{13})$	$(g^9, g^{10})$	$(g^{12}, g^{12})$

Che riportati graficamente,



**Figura:** Insieme dei punti che soddisfano l'equazione della EC in campo  $E(2^4)$

### 1.2.5 Problema del logaritmo discreto

Il problema del logaritmo discreto può essere formulato nelle seguenti accezioni:

- 1. Il problema del logaritmo discreto in un insieme finito (DLP):** dato un insieme finito  $F_p$  e gli elementi  $g, h \in F_p$ , trovare un intero  $l$  tale che  $g^l = h$  in  $F_p$ , purché un tale intero esista.
- 2. Il problema del logaritmo discreto su curva ellittica (ECDLP):** data una curva ellittica  $E$  definita su insieme finito  $F_p$  e due punti  $P, Q \in E(F_p)$ , trovare un intero  $k$  tale che  $kP = Q$  in  $E$ , purché un tale intero esista.

Nel primo problema, è usata la notazione "moltiplicativa":  $g^l$  si riferisce al processo di moltiplicazione  $g$  per se stesso  $l$  volte. Nel secondo problema, è usata la notazione "additiva":  $kP$  si riferisce al processo di addizione  $P$  per se stesso  $l$  volte  $P + P + \dots + P = kP$ .

Il secondo problema è molto più difficile del primo perchè gli oggetti algebrici nel *DLP* sono forniti con due operazioni base ovvero addizione e moltiplicazione, mentre gli oggetti algebrici

nell'*ECDLP* sono forniti con una sola operazione cioè addizione di punti su curve ellittiche.

Sui campi finiti, esiste un algoritmo, detto "index calculus"<sup>5</sup>, che tramite opportune precomputazioni, permette di calcolare il logaritmo discreto con complessità sub esponenziale. Questo è possibile perché sui campi finiti esistono due operazioni:somma e prodotto. Al contrario, sulle curve ellittiche, che hanno solamente la struttura additiva, questo tipo di calcolo indicizzato non è possibile, conseguentemente le operazioni risultano più laboriose.

**La sicurezza dei crittosistemi basati su curve ellittiche, si fonda sulla difficoltà del problema del logaritmo discreto su curve ellittiche.** Ad oggi, non è conosciuto nessun algoritmo efficiente per la risoluzione di tale problema. La forte sicurezza, dovuta all'assenza di algoritmi subesponenziali per l'*ECDLP*, insieme con le implementazioni efficienti dell'aritmetica sulle curve ellittiche, fa sì che tali sistemi crittografici siano i più utilizzati oggi.

L'efficienza di *ECC* dipende dal calcolo:

$$Q = kP$$

dove  $P$  è un punto della curva ellittica e  $k$  è un intero.

Vediamo un esempio di calcolo di un  $kP$  :

data la curva ellittica di equazione  $y^2 = x^3 + 9x + 17$  su  $F_{23}$ , quale è il logaritmo discreto  $k$  di  $Q \equiv (4, 5)$  con base  $P \equiv (16, 5)$ ?

Una strada meccanica per calcolare  $k$  è calcolare passo per passo i multipli di  $P$  fino a quando  $Q$  non è stato trovata.:

Applicando la formula per il calcolo del punto doppio si ha:

$$\lambda = \left( \frac{3 * (16^2) + 9}{2 * 5} \right) = \left( \frac{777}{10} \right) = 80 \text{ mod mod } 23 = 11$$

quindi  $x_3 = \lambda^2 - 16 - 16 = 11^2 - 32 = 89 \text{ mod } 23 = 20$  il primo punto è **20**.

$y_3 = \lambda (16 - 20) - 5 = - 44 - 5 = -49 \text{ mod } 23 = 20$  quindi il punto doppio è **2P(20,20)**.

Per calcolare il terzo punto bisogna utilizzare la formula della somma dei punti con **P(16,5)** e **Q(20,20)** .

---

<sup>5</sup> <http://www.cecm.sfu.ca/CAG/abstracts/aaron27Jan06.pdf>

$\lambda = \left( \frac{20-5}{20-16} \right) = \left( \frac{15}{4} \right) = x \pmod{23}$  cioè significa risolvere  $4x - 15 = 23$  ossia  $x = \left( \frac{19}{2} \right)$  che significa ancora risolvere  $\left( \frac{19}{2} \right) = \lambda \pmod{23}$  cioè  $\lambda = 21$ .

Ne consegue che  $x_3 = 21^2 - 16 - 20 = 405 \pmod{23} = \mathbf{14}$  e via di seguito, riassumendo i primi multipli di  $P$  sono

$P \equiv (16, 5), 2P \equiv (20, 20), 3P \equiv (14, 4), 4P \equiv (19, 20), 5P \equiv (13, 10), 6P \equiv (7, 3), 7P \equiv (8, 7),$   
 $8P \equiv (12, 17), 9P \equiv (4, 5)$

Così  $9P = (4, 5) = Q$ , il logaritmo discreto di  $Q$  con base  $P$  è  $k = 9$ .

Logicamente l'approccio mostrato nell'esempio e fatto di calcoli successivi dei multipli del punto  $P$  è praticabile poiché  $F_{23}$  contiene un numero basso di punti della curva, non sarebbe invece un procedimento risolutivo efficace se stessimo operando nel campo  $E(2^{163})$  !

In sintesi l'efficienza degli algoritmi crittografici basati su curve ellittiche è dovuta, oltre ad una chiave opportunamente grande, anche ad altri due fattori:

- un minor numero di bit necessari per la chiave ( vedremo che  $E(2^{163})$  usa 163 bit, corrispondenti come equivalente grado di sicurezza ai 1024 bit del metodo RSA )
- conoscendo poche informazioni  $(P, Q)$ , i calcoli computazionali per criptare e decriptare non sono particolarmente onerosi

La sicurezza invece è dovuta all'enorme difficoltà nel risolvere inversamente il problema del logaritmo discreto su curva ellittica. Lavorando con  $m$  opportunamente grandi, senza conoscere informazioni su uno tra  $Q$  e  $P$ , risalire al valore di  $k$  risulta computazionalmente proibitivo (almeno con gli algoritmi di forzatura fino a qui conosciuti e con le attuali tecnologie).

### 1.3 ECC: Crittografia a curve ellittiche

I crittosistemi a chiave pubblica sono classificati in base al problema matematico su cui essi si basano. Prima di addentrarci nello studio dei crittosistemi basati su curve ellittiche, è riportata di seguito una breve panoramica dei sistemi crittografici a chiave pubblica basati rispettivamente sul problema della fattorizzazione degli interi e sul problema del logaritmo discreto.

#### 1.3.1 Crittosistemi a chiave pubblica : RSA

Lo schema RSA è una cifratura a blocchi in cui il testo in chiaro e il testo cifrato sono interi

compresi fra 0 e  $n - 1$  per un dato valore  $n$ . Normalmente  $n$  è pari a 1024 bit ovvero 309 cifre decimali. Ovvero,  $n$  è minore di  $2^{1024}$

Lo schema sviluppato da Rivest, Shamir e Adleman utilizza un'espressione con blocchi esponenziali. Il testo in chiaro viene crittografato a blocchi, dove ciascun blocco deve avere dimensioni minori o uguali di  $\log_2(n)$ ; ovvero deve avere dimensioni di  $k$  bit dove  $2^k < n \leq 2^{k+1}$ . La crittografia e la decrittografia di un determinato blocco di testo in chiaro  $M$  e del corrispondente blocco di testo cifrato  $C$  hanno la seguente forma:

$$C = M^e \text{ mod } n$$

$$M = C^d \text{ mod } n = (M^e)^d \text{ mod } n = M^{ed} \text{ mod } n$$

Il valore di  $n$  deve essere noto sia al mittente che al destinatario. Il mittente conosce il valore di  $e$  mentre solo il destinatario conosce il valore di  $d$ .

Pertanto questo è un algoritmo con una chiave pubblica  $PU = \{e, n\}$  e una chiave privata  $PR = \{d, n\}$ .

Per generare le chiavi si opera nel seguente modo.

Operazione	Condizioni
Si seleziona $p, q$	$p$ e $q$ entrambi primi con $p \neq q$
Calcolare $n = p \times q$	
Calcolare $\phi(n) = (p - 1)(q - 1)$	<i>Funzione di Eulero</i> di $n$
Selezionare intero $e$	
Calcolare $d \equiv e^{-1} \text{ mod } \phi(n)$	
Chiave pubblica	$PU = \{e, n\}$
Chiave privata	$PR = \{d, n\}$

Quindi per crittografare se abbiamo un testo in chiaro :  $M < n$

Testo cifrato	$C = M^e \text{ mod } n$
---------------	--------------------------

Per decrittografare

Testo cifrato	$C$
Testo in chiaro	$M = C^d \text{ mod } n$

### 1.3.2 Aspetti computazionali dell' algoritmo RSA

La crittografia e la decrittografia in RSA comportano l'elevamento di un intero a una potenza intera modulo  $n$ . Se venisse eseguita prima la potenza nel campo degli interi e poi la riduzione modulo  $n$ , i valori intermedi diventerebbero immensi. Fortunatamente, si può utilizzare una proprietà dell'aritmetica modulare:

$$(a \times b) \bmod n = [(a \bmod n) (b \bmod n)] \bmod n$$

Pertanto si possono ridurre i risultati intermedi modulo  $n$ , semplificando notevolmente il calcolo.

E' interessante notare l'efficienza dell'elevamento a potenza, poiché con RSA si ha che fare con esponenti potenzialmente altissimi. Per vedere come si può aumentare l'efficienza, si consideri per esempio il caso in cui si desidera calcolare  $x^{16}$ . Un approccio immediato prevede l'impiego di 15 moltiplicazioni.

Tuttavia si può ottenere lo stesso risultato finale utilizzando solo quattro moltiplicazioni se si prende ripetutamente il quadrato di ciascun risultato parziale formando successivamente  $x^2, x^4, x^8, x^{16}$ .

Un altro problema matematico definito in termini di aritmetica modulare è il problema del *logaritmo discreto*. Fissato un numero primo  $p$  e un intero  $g$  tra 0 e  $p - 1$  si ottiene  $y$  dalla relazione:

$$y = g^x \pmod{p}$$

Il problema del *logaritmo discreto* è di determinare l'intero  $x$  dati  $g, y$  e  $p$ .

Non è stato trovato nessun algoritmo efficiente in grado di risolvere questo problema.

*Taher ElGamal* fu il primo a proporre un sistema crittografico basato su questo problema. In particolare, propose due distinti sistemi: uno schema crittografico e uno schema di firme digitali (*DSA* si basa sul lavoro di *ElGamal*). Rompere uno di questi schemi significa risolvere il problema del *logaritmo discreto*.

Per avere una buona sicurezza il primo  $p$  deve essere lungo almeno 230 cifre decimali (760 bits).

Le prestazioni del sistema dipendono dalla velocità di esecuzione dell'esponenziazione modulare. Il calcolo dominante in ogni trasformazione è:

$$g^x \pmod{p}$$

con  $g$  compreso tra 0 e  $p - 1$ .

In conclusione la sicurezza di questi sistemi si basa sul problema del *logaritmo discreto* modulo  $p$ , mentre l'efficienza dipende dalla velocità di esecuzione dell'esponenziazione modulare.

Dopo aver introdotto le nozioni che stanno alla base delle curve ellittiche e dei sistemi crittografici a chiave pubblica, illustreremo ora due protocolli di sistemi crittografici che sfruttano le proprietà

delle curve ellittiche.

### 1.3.3 ECDSA (Elliptic Curve Digital Signature Algorithm)

ECDSA (Elliptic Curve Digital Signature Algorithm) è lo schema di firme digitali DSA basato su curve ellittiche. Fu proposto la prima volta nel 1992 da Scott Vanstone. Nel 1998 è diventato uno standard ISO (ISO 14888), nel 1999 è stato accettato come standard ANSI (ANSI X9.62) mentre nel 2000 è diventato uno standard IEEE (IEEE P1363 2). Di seguito viene discusso lo standard ANSI X9.62.

I parametri del dominio ECDSA consistono in una curva ellittica  $E$  definita su un campo finito  $F_q$  e di un punto base  $G \in E(F_q)$ . I parametri del dominio potrebbero essere condivisi da un gruppo di entità, oppure specificati su un singolo utente.

**Requisiti del campo.** Nel caso  $q = p$ , il campo finito è  $F_p$ , gli interi sono modulo  $p$ .

Nel caso  $q = 2^m$ , il campo finito è  $F_{2^m}$  i cui elementi sono rappresentati tramite un opportuno polinomio o una base normale.

**Requisiti curva ellittica.** Per evitare gli attacchi *rho di Pollard* e il *Pohlig-Hellman* presentati in appendice sul problema del logaritmo discreto sulle curve ellittiche, è necessario che il numero di punti razionali di  $F_q$  su  $E$  siano divisibili per un numero primo  $n$  sufficientemente grande. ANSI X9.62 ha posto che si consideri  $n > 2^{160}$ .

Alcuni ulteriori precauzioni dovrebbero essere utilizzate quando si seleziona la curva. Per evitare la degradazione degli algoritmi di *Menezes*, "*Okamoto e Vanstone*" e "*Frey e Ruck*"[5], la curva dovrebbe essere non supersingolare. In generale, si dovrebbe verificare che  $n$  non divida  $q^k - 1$  per tutto  $1 \leq k \leq C$ , dove  $C$  è molto grande così che sia computazionalmente impossibile trovare logaritmi discreti in  $F_q^C$ .

Un modo prudente di difendersi contro questi attacchi su speciali classi di curve che potrebbero essere scoperte in futuro è di selezionare la curva ellittica  $E$  a caso, a condizione che il numero di punti di  $E(F_q)$  sia divisibile per un primo grande.

Ricapitolando, i parametri del dominio sono composti da:

1. La grandezza del campo  $q$ , dove  $q = p$  o  $q = 2^m$
2. Un'indicazione **FR** (rappresentazione campo) della rappresentazione usata per gli elementi di  $F_q$
3. Due elementi campo  $a$  e  $b$  in  $F_q$  che definiscono l'equazione della curva ellittica  $E$  su  $F_q$   
(es.  $y^2 = x^3 + ax + b$  nel caso  $p > 3$  e  $y^2 = x^3 + ax^2 + b$  nel caso  $p = 2$ )

4. Due elementi del campo,  $x_G$  e  $y_G$  in  $F_q$  che definiscono un punto finito  $G = (x_G, y_G)$  di ordine primo in  $E(F_q)$
5. L'ordine  $n$  del punto  $G$ , con  $n > 2^{160}$  e  $n > 4\sqrt{q}$ ;
6. Cofattore  $h = \#N(F_q) / n$ .
7. Con questi parametri di dominio si procede quindi alla generazione delle chiavi pubbliche e private.
8.  $D = (q, FR, a, b, G, n, h)$

Il processo che in un sistema crittografico ECDSA porta alla generazione di una coppia di chiavi può essere riassunto in maniera schematica dai seguenti passi:

- a) Generazione dei parametri del dominio
- b) Validazione dei parametri del dominio
- c) Generazione di una curva ellittica casuale su  $F_q (F_{2^m})$ .
- d) Verifica del fatto che una curva ellittica sia stata generata casualmente su  $F_q (F_{2^m})$ .
- e) Generazione della coppia di chiavi
- f) Validazione della chiave pubblica
- g) Generazione e verifica della firma ECDSA

Dunque ad un dominio  $D = (q, FR, a, b, G, n, h)$  è possibile associare una coppia di chiavi  $(d, Q)$ .

Un utente  $A$  per generare una coppia di chiavi  $(d, Q)$  esegue dunque i seguenti passi:

1. Seleziona un intero  $d$  casuale o pseudocasuale nell'intervallo  $[1, n - 1]$ .
2. Calcola  $Q = d * G$ .
3. la **chiave pubblica** di  $A$  è  $Q$ ; la **chiave privata** di  $A$  è  $d$ .

La validazione delle chiavi pubbliche assicura che una chiave pubblica abbia i requisiti di proprietà aritmetiche richiesti.

Le ragioni per eseguire la validazione della chiave pubblica includono in pratica:

- la prevenzione di maliziosi inserimenti di una chiave pubblica non valida che potrebbe consentire qualche attacco;
- la scoperta di codificazioni involontarie o trasmissioni di errori.

L'uso di una chiave pubblica non valida può annullare tutte le proprietà di sicurezza.

### 1.3.4 EC Diffie-Hellman

Descriviamo il protocollo di Diffie-Hellman basato su curve ellittiche.

Consideriamo due utenti,  $M$  e  $N$ , che devono trovare l'accordo su una chiave.  $M$  e  $N$  scelgono un campo finito  $F_p$ , una curva ellittica  $E$  e un punto  $B$  di ordine  $q$  su tale curva, che non ha bisogno di essere segreto.

La chiave di accordo sarà un punto  $P$  (*chiave pubblica*), scelto casualmente sulla curva.

$M$  sceglie la sua chiave segreta  $a \in [1, q-1]$  e calcola  $a * B \in E$ .

$N$  sceglie la sua chiave segreta  $b \in [1, q-1]$  e calcola  $b * B \in E$ .

A questo punto i due utenti si scambiano i rispettivi valori,  $M$  invia a  $N$  il valore calcolato  $a * B$  e  $N$  invia ad  $M$  il valore  $b * B$ .

Una volta ricevuti tali valori,  $M$  computa  $P = a * b * B$  e  $N$  computa  $P = b * a * B$ , l'accordo è raggiunto.

Vediamo un piccolo esempio per capire meglio.

Consideriamo un campo finito  $F_5$  ( $p = 5$ ) e una curva ellittica  $E: y^2 = x^3 + 2x + 1$ .

I punti di tale curva sono hanno coordinate:

$(0, 1)$   $(1, 3)$   $(3, 3)$   $(3, 2)$   $(1, 2)$   $(0, 4)$  insieme con il punto  $O$ .

Supponiamo che venga scelto il punto  $B \equiv (0, 1)$ .

Mario sceglie la sua chiave segreta  $a = 2$  e computa la sua chiave pubblica  $PA$  dove:

$$PA = a * B = 2 * B = B + B = (1, 3)$$

Nicola sceglie la sua chiave segreta  $b = 3$  e computa la sua chiave pubblica  $PB$  dove:

$$PB = b * B = 3 * B = B + B + B = (3, 3)$$

A questo punto Mario e Nicola si scambiano le rispettive chiavi pubbliche:

Mario riceve  $b * B = (3, 3)$  e Nicola riceve  $a * B = (1, 3)$ .

Mario computa  $P$  nel seguente modo:  $P = a * (b * B) = 2 * (3, 3) = (0, 4)$

Nicola computa  $P$  nel seguente modo:  $P = b * (a * B) = 3 * (1, 3) = (0, 4)$

Mario e Nicola hanno trovato l'accordo sulla chiave  $P = (0, 4)$ .

### 1.3.5 Sicurezza ed efficienza: confronto tra crittosistemi a chiave pubblica.

Senza dubbio i due maggiori parametri per il confronto tra crittosistemi a chiave pubblica sono la sicurezza e l'efficienza.

La sicurezza dei crittosistemi a chiave pubblica in generale è data dalla difficoltà di rompere tali sistemi. Esistono diversi tipi di attacchi a tali critto sistemi, la cosa importante da osservare è che **la**

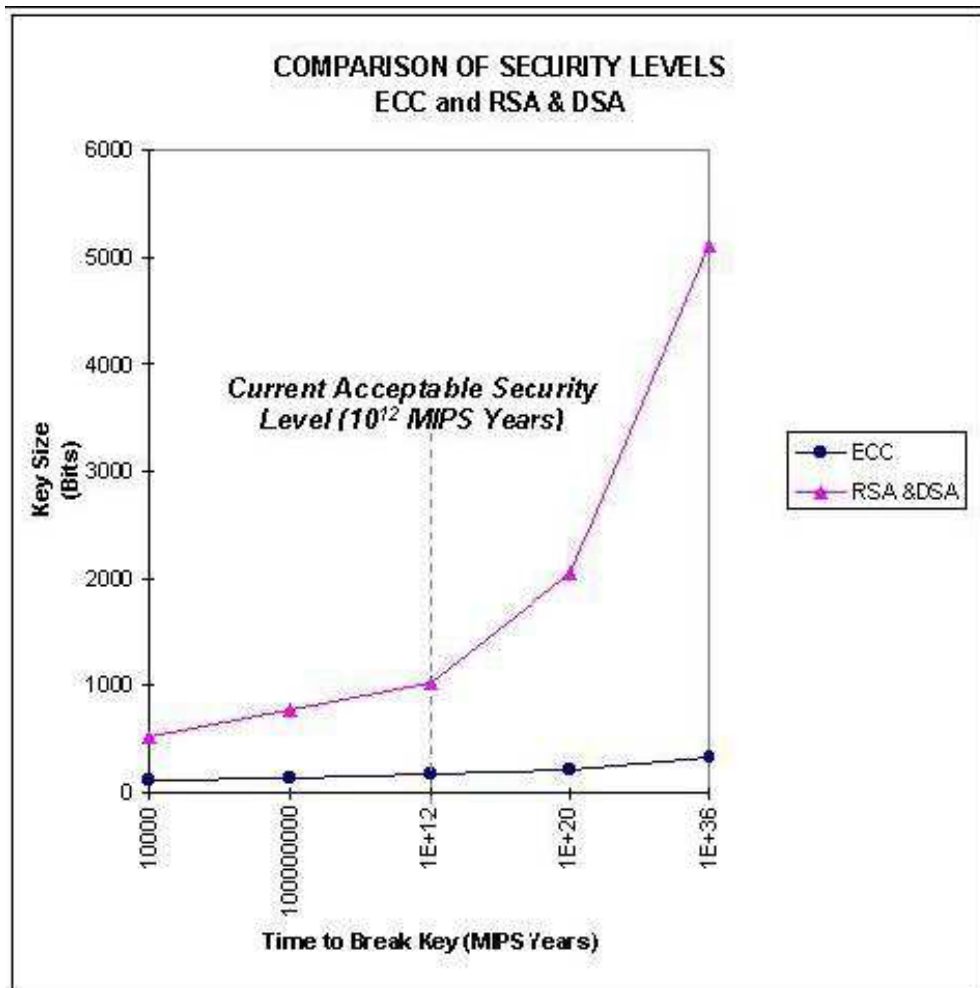
**rottura di tali crittosistemi, richiede la risoluzione del problema matematico su cui essi si basano.** Infatti molti sistemi sono stati rotti fornendo una dimostrazione matematica formale su tale problema. Sfortunatamente non ci sono problemi matematici per i quali può essere dimostrato che il miglior algoritmo prende tempo esponenziale. L'attenzione si sposta quindi sui migliori algoritmi conosciuti oggi per risolvere questi problemi. Per ogni problema in questione sono stati proposti algoritmi veloci.

Il problema del logaritmo discreto su curve ellittiche è relativamente semplice per piccole classi di curve ellittiche. La fattorizzazione degli interi e il problema del logaritmo discreto modulo  $p$ , hanno entrambi un tempo di risoluzione sub-esponenziale. I problemi che hanno un tempo di risoluzione sub esponenziale possono essere ancora considerati difficili, ovviamente non come i problemi difficili che ammettono per la loro risoluzione solo algoritmi pienamente esponenziali.

Il miglior algoritmo per la risoluzione del problema del logaritmo discreto su curve ellittiche richiede tempo pienamente esponenziale. Tale tempo è:  $O(\sqrt{p})$

Questo significa che il problema del logaritmo discreto su curve ellittiche è oggi considerato più difficile sia del problema della fattorizzazione degli interi che del problema del logaritmo discreto su aritmetica modulare.

Possiamo osservare ciò concretamente nella seguente figura in cui sono confrontati i tempi per rompere *RSA* o *DSA*, utilizzando diverse chiavi e i migliori algoritmi conosciuti. I valori sono confrontati in anni *MIPS*, un anno *MIPS* rappresenta il tempo di computazione di un anno su una macchina capace di eseguire un milione di operazioni al secondo. Normalmente si accetta come banco di misura ragionevolmente sicuro,  $10^{12}$  anni *MIPS*.



**Figura 4** : Confronto del grado di sicurezza tra gli algoritmi RSA e ECC al variare delle dimensioni della chiave . [1]

Dalla figura reperita dal sito [www.certicom.com](http://www.certicom.com) possiamo osservare che, per raggiungere una ragionevole sicurezza, *RSA* e *DSA* devono impiegare 1024 bit di lunghezza della chiave, mentre 160 bit di modulo potrebbero essere sufficienti per *ECC*. Non solo, si osserva anche che il gap relativo alla sicurezza tra i sistemi cresce al crescere della taglia delle chiavi.

Per quanto riguarda l'efficienza, ci sono tre distinti fattori da prendere in considerazione:

1. **Computer overheads:** quanti calcoli sono richiesti per eseguire trasformazioni da chiavi pubbliche a chiavi private.
2. **Key size:** quanti bit sono richiesti per immagazzinare la coppia di chiavi e altri parametri di sistema.
3. **Bandwidth:** quanti bit devono essere comunicati per trasferire un messaggio criptato o una firma.

Chiaramente il confronto deve essere fatto tra sistemi con simili livelli di sicurezza (160 bit *ECC* vengono confrontati con 1024 bit *DSA* o *RSA*).

Se per il punto 1 le differenze tra *RSA* ed sistemi *ECC* sono minime anche se *ECC* risulta comunque migliore, sono invece evidenti i vantaggi in termini di efficienza per quanto riguarda le dimensioni in bit di chiave e di messaggio criptato.

Con la seguente figura vengono messe a confronto le lunghezze delle chiavi in tre sistemi :

	<i>System parameters (bits)</i>	<i>Public key (bits)</i>	<i>Private key (bits)</i>
<i>RSA</i>	<i>n/a</i>	<i>1088</i>	<i>2048</i>
<i>DSA</i>	<i>2208</i>	<i>1024</i>	<i>160</i>
<i>ECC</i>	<i>481</i>	<i>161</i>	<i>160</i>

Appare chiaro come *ECC* risulti migliore rispetto a tutti e 3 i parametri di confronto.

In termini di "bandwidth" tutti e tre i sistemi hanno valori simili quando sono usati per crittografare o per firmare lunghi messaggi.

Le due figure successive invece mettono a confronto, per i tre sistemi, la lunghezza della firma e la lunghezza del messaggio crittografato, considerando un messaggio di 2000 bit per la firma ed un messaggio di 100 bit da cifrare.

	<i>Signature size (bits)</i>
<i>RSA</i>	<i>1024</i>
<i>DSA</i>	<i>320</i>
<i>ECC</i>	<i>320</i>

Taglia della firma per lunghi messaggi

	<i>Encrypted message (bits)</i>
<i>RSA</i>	<i>1024</i>
<i>ElGamal</i>	<i>2048</i>
<i>ECC</i>	<i>321</i>

Quindi *ECC* offre considerevole risparmio di bandwidth quando si considerano messaggi di taglia corta.

**In definitiva, *ECC* fornisce una maggiore efficienza in termini di computer overhead, key size e bandwidth, rispetto agli altri due sistemi.**

### 1.3.5 Standards dell'ECC

Lo studio delle curve ellittiche appena uscito dall'ambito accademico ha suscitato interesse per la varietà di applicazioni da terze parti cui poteva essere utilizzata. Questo interesse da parte di società, enti e organizzazioni mondiali ha inevitabilmente portato verso una standardizzazione del sistema crittografico. Si è iniziato nello standardizzare certe famiglie di curve, con relativi

parametri di dominio, da utilizzare per scopi crittografici. Il NIST ha pubblicato un elenco di *curveraccomandate* che costituisce un riferimento per tutte le implementazioni ECC chiamato “NIST, SP 800-57 “Recommendation for Key Management”, August 2005. Gli stard principali sono: **ANSI** L’American National Standards Institute è un organizzazione privata la cui missione è promuovere la diffusione di standards. Il comitato X9 è parte dell’ANSI e sviluppa standards per i servizi bancari e finanziari. In particolare il sottocomitato X9F, che si occupa delle problematiche di sicurezza informatica, ha sviluppato due standards relativi all’ECC:

- X9.62 (1999): standardizza l’ECDSA
- X9.63 (2001): standardizza vari protocolli di *key agreement* e *key transport*, tra i quali ECDH, ECMQV e ECIES.

**NIST** Il National Institute of Standards and Technology è un’agenzia federale del Dipartimento del Commercio degli Stati Uniti. Tra i suoi compiti vi è la pubblicazione dei FIPS (Federal Information Processing Standards) che regolamentano problematiche di sicurezza per ambienti governativi. Il NIST ha accreditato l’algoritmo ECDSA, insieme a DSA e RSA, come algoritmo di Firma Elettronica nel FIPS 186-2, pubblicato il 27 gennaio 2000. Un altro standard molto importante è il FIPS 140-2 il quale specifica i requisiti che il governo degli Stati Uniti richiede a tutti i prodotti hardware e software che trattano informazioni Crittografia basata su Curve Ellittiche 73 riservate (escluse quelle di carattere militare). In esso gli unici algoritmi asimmetrici riconosciuti sono RSA, DSA e ECDSA.

**IEEE** L’Institute of Electrical and Electronics Engineers è un organizzazione privata dedicata a promuovere pubblicazioni, convegni e standards. Il gruppo di lavoro IEEE P1363 si occupa della standardizzazione della crittografia a chiave pubblica e lo standard 1363-2000 (Standards Specification for Public-Key Cryptography) contiene, tra gli altri, gli algoritmi ECDSA, ECDH, ECMQV. La bozza 1363a invece è un complemento al 1363-2000 ed include anche ECIES. **IETF** L’Internet Engineering Task Force si occupa di sviluppare standards relativi ai protocolli utilizzati su Internet. A livello crittografico l’IETF è stata decisiva per la diffusione degli standards IPsec, TLS (Transport Layer Security), S/MIME. Vediamo come tali standards contemplano l’uso dell’ECC:

- IPsec: Il protocollo IKE (Internet Key Exchange), spiegato nell’RFC 2409, prevede l’impiego di ECDH per *key agreement*. Inoltre l’Internet Draft “*ECP Groups for IKE and IKEv2*” descrive come utilizzare certe curve raccomandate dal NIST in IKE. Su tale draft diversi produttori come Cisco e Nortel basano le proprie implementazioni di ECDH in IKE.

- SSL/TLS: Attualmente vi è solo un Internet-Draft, “*ECC Cipher Suites for TLS*”, che descrive come integrare gli algoritmi di scambio chiavi basati su ECC in TLS. In particolare specifica l’uso di ECDH nella fase di *handshake* e ECDSA come meccanismo di autenticazione. Considerato che è ancora in stato di *draft* (bozza) non esistono soluzioni commerciali che utilizzano l’ECC in TLS.

- S/MIME: L’utilizzo di S/MIME (RFC 3369) consente di proteggere la posta elettronica, basandosi sulla Cryptographic Message Syntax (CMS). L’RFC 3278 “*Use of ECC Algorithms in Cryptographic Message Syntax*” standardizza l’impiego degli algoritmi ECDH, ECDSA, ECMQV in CMS.

**ISO/IEC** L’International Standards Organization e l’International Electrotechnical Commission hanno congiuntamente sviluppato standards in ambito crittografico. Lo standard ISO/IEC 15946 descrive diversi algoritmi di Crittografia basata su 74 Curve Ellittiche firma elettronica e di *key establishment* basati su EC, come ECDSA, ECDH, ECMQV.

Infine per quanto riguarda l’aspetto commerciale particolare menzione va alla la Certicom. Questa si è impegnata, anche a livello di brevetti, nel commercializzare prodotti crittografici basati su EC.

## **2. Implementazione e performance di ECC nelle transazioni Web sicure**

### **2.1 Introduzione**

Negli ultimi anni il rapido sviluppo di applicazioni quali online banking, stock trading e corporate remote access ha determinato una crescita considerevole del volume di scambi di dati via Web, inoltre è progressivamente aumentata anche la diffusione di dispositivi alimentati a batteria, equipaggiati con wireless, dotati di memorie ridotte, CPU non molto veloci, etc. A queste linee di tendenza del mercato si associa il bisogno di efficienza, scalabilità e meccanismi di sicurezza che operino bene tanto in ambienti wired quanto in ambienti wireless.

Molti protocolli sicuri di sicurezza (SSL, IPsec) si basano sulla crittografia a chiave pubblica dopo una fase di handshake ricavano una chiave simmetrica che viene usata per aumentare la velocità degli scambi ferma restando la confidenzialità e integrità degli stessi

La sicurezza di un sistema si correla alla robustezza dei suoi componenti: per esempio la quantità di lavoro necessaria per “rompere” una chiave simmetrica può essere equivalente alla quantità di lavoro necessaria per decrittare una chiave pubblica usata per una chiave stabilita.

Di conseguenza – poiché i computer sono sempre più veloci – ci dovremmo aspettare una continua crescita delle chiavi al fine di garantire la sicurezza, invece la tabella

Simmetrica	ECC	RSA/DH/DSA
80	160	1024
128	224	3072
192	384	7680
256	512	15360

Tabella di confronto di computazione per chiavi di equivalente lunghezza

evidenzia non solo che ECC usa una chiave più piccola, a parità di sicurezza, ma anche che si nota una crescita non proporzionale a favore di ECC rispetto a RSA.

Tale dato risulta particolarmente interessante soprattutto per i wireless device poiché una chiave piccola è sinonimo di risparmio di potenza, banda e capacità computazionale.

### 2.1.1 Secure Sockets Layer

Dalla metà degli anni '90 la diffusione crescente di servizi di e-commerce ha riproposto con rinnovata urgenza il problema della sicurezza nelle transazioni via Web, i maggiori gruppi attivi nel settore hanno studiato differenti meccanismi di sicurezza e - sebbene inizialmente questi non siano stati adottati da IETF - nel tempo una delle proposte formulate è divenuta uno standard de facto: si tratta della tecnologia SSL (Secure Socket Layer), in origine sviluppata da Netscape.

Si trova allo stesso strato dell'API socket e prevede che - quando un client usa una SSL per contattare un server - il protocollo SSL consenta a ciascun lato di autenticare se stesso rispetto all'altro. I due lati poi adotteranno di comune accordo un algoritmo di crittografia per garantire la privacy delle loro comunicazioni e sulla base di questo stabiliranno appunto una connessione criptata.

IETF usa SSL come base per un protocollo chiamato Transport Layer Security (TLS, Sicurezza a livello di trasporto). SSL e TLS sono strettamente legati, usano entrambi la stessa porta nota e la maggior parte delle implementazioni di SSL supporta TLS.

L'architettura SSL è stata progettata per impiegare TCP con un servizio affidabile end-to-end.

SSL non è un unico protocollo, ma è costituito da due livelli di protocolli, come indicato nella figura seguente:

SSL Handshake Protocol	SSL Change Cipher Spec Protocol	SSL alert Protocol	HTTP
SSL Record protocol			
TCP			
IP			

Lo stack protocolli SSL.

Il protocollo SSL Record fornisce i servizi di sicurezza di base per i vari protocolli di livello superiore. In particolare, il protocollo HTTP, che fornisce il servizio di trasferimento per le interazioni Web client/server, può operare sopra SSL. Nell'ambito di SSL sono definiti tre protocolli di più alto livello: Handshake Protocol, Change Cipher Protocol e Alert Protocol. Questi protocolli SSL vengono utilizzati nella gestione degli scambi SSL.

I due concetti chiave alla base della tecnologia SSL sono la sessione e la connessione, che vengono definiti nel seguente modo:

- **connessione:** è una forma di trasporto che definisce un determinato tipo di servizio, per SSL le connessioni sono transitorie, ad ogni connessione è associata una sola sessione.
- **Sessione:** è un'associazione fra client e server. Le sessioni vengono create dal protocollo Handshake e definiscono un insieme di parametri di sicurezza crittografica che possono essere condivisi fra più connessioni. Le sessioni vengono utilizzate per evitare di svolgere la costosa negoziazione di nuovi parametri di sicurezza per ciascuna connessione.

A ciascuna sessione vengono associati più stati.

Una volta attivata una sessione, vi è uno stato operativo corrente per la lettura e la scrittura. Alla conclusione del protocollo di Handshake, gli stati provvisori diventano stati correnti.

Uno stato di sessione è definito dai seguenti parametri:

- **Sessioni identifier** : una sequenza di byte arbitraria scelta dal server per identificare lo stato di una sessione attiva o riattivabile.
- **Peer Certificate:** il certificato X.509 v3 del nodo.

- **Compression method:** l'algoritmo utilizzato per comprimere dei dati e l'algoritmo hash utilizzato per il calcolo del codice MAC.
- **Cipher Spec:** specifica l'algoritmo di crittografia dei dati (null, AES e così via) e l'algoritmo hash (come MD5 o SHA-1) utilizzato per il calcolo del codice MAC. Inoltre definisce gli attributi crittografici, per esempio hash\_size.
- **Master secret:** un codice segreto di 48 byte condiviso tra client e server
- **Is resumable:** un flag che indica se la sessione può essere utilizzata per iniziare nuove connessioni.

Lo stato di una connessione invece è definito dai seguenti parametri:

- **Server and client random:** sequenze di byte scelte dal server e dal client per ciascuna connessione.
- **Server write MAC secret:** la chiave segreta utilizzata nelle operazioni MAC per i dati inviati dal server
- **Client write MAC secret:** la chiave segreta utilizzata nelle MAC sui dati inviati dal client
- **Server write key:** la chiave di crittografia convenzionale per i dati crittografati dal client e decrittografati dal server.
- **Inizialization vector:** vettore di inizializzazione utilizzato durante l'handshake.
- **Sequence number:** numero di sequenza utilizzato per lo scambio di messaggi.

### 2.1.2 Il Protocollo SSL Record

Il protocollo SSL Record fornisce due servizi per le connessioni SSL.

- **Segretezza:** il protocollo Handshake definisce una chiave segreta condivisa utilizzata per la crittografia convenzionale del payload SSL.
- **Integrità del messaggio:** il protocollo Handshake definisce anche una chiave segreta condivisa che viene utilizzata per creare un codice MAC (Message Authentication Code).

SSL Record accetta il messaggio da trasmettere, frammenta i dati in blocchi di dimensioni appropriate, opzionalmente comprime i dati, applica un codice MAC, esegue la crittografia, aggiunge l'intestazione e trasmette l'unità risultante in un segmento TCP. I dati ricevuti vengono decrittografati e assemblati, quindi vengono consegnati agli utenti di livello più elevato.

Il primo passo di questo processo dunque è la frammentazione dei messaggi. Ciascun messaggio dal livello superiore viene frammentato in blocchi di  $2^{14}$  byte o meno, se necessario – a seguire – tali blocchi possono anche essere compressi.

Il secondo passo consiste nel calcolo del codice MAC sui dati compressi. A tale scopo viene utilizzata una chiave segreta: quindi il messaggio compresso più il codice MAC vengono crittografati utilizzando la crittografia simmetrica, la crittografia non può aumentare la lunghezza del contenuto di più di 1024 byte.

L'ultimo passo di elaborazione del protocollo SSL Record comporta invece l'aggiunta di un'intestazione iniziale costituita dai seguenti campi:

- **Content Type** (8 bit): il protocollo di livello superiore utilizzato per elaborare il frammento incluso
- **Major Version** (8 bit): la versione major di SSL in uso.
- **Minor Version** (8 bit): la versione minor in uso.
- **Compressed Length** (16 bit): la lunghezza in byte del frammento di testo in chiaro. Il valore massimo è  $2^{14} + 2048$ .

### 2.1.3 Il protocollo Change Cipher Spec

Un altro dei tre protocolli specifici di SSL legati a SSL Record è il protocollo Change Cipher Spec. Esso è costituito da un unico messaggio - rappresentato in figura xx - costituito da un unico byte contenente il valore 1. Lo scopo di questo messaggio è far sì che lo stato provvisorio venga copiato nello stato corrente, aggiornando e quindi attivando la cifratura che verrà utilizzata in questa connessione.

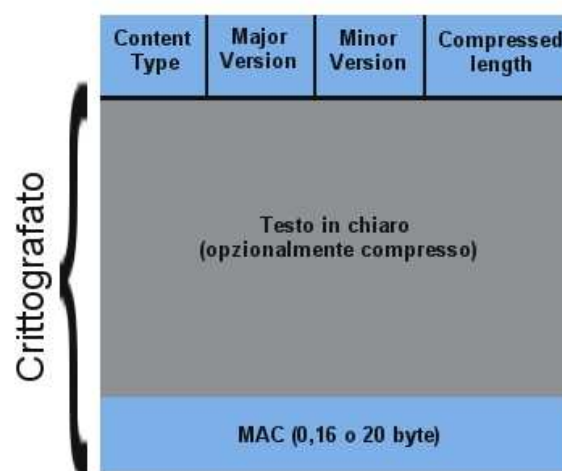


Figura 5

### 2.1.4 Il Protocollo Alert

Il protocollo Alert viene utilizzato invece per trasmettere allarmi SSL alle entità peer.

Come per altre applicazioni che utilizzano SSL, i messaggi alert vengono compressi e crittografati, come specificato nello stato corrente.

Ciascun messaggio di questo protocollo è costituito da due byte.

Il primo byte assume il valore warning o fatal per indicare la gravità del messaggio: se il livello è fatal, SSL chiude immediatamente la connessione, altre connessioni nella stessa sessione potranno continuare, ma non si potranno attivare nuove connessioni in questa sessione.

Il secondo byte contiene un codice che indica l'allarme specificato.

### 2.1.5 Il protocollo Handshake

La parte più complessa di SSL è il protocollo handshake. Questo protocollo consente al server e al client di autenticarsi l'un l'altro e di negoziare un algoritmo di crittografia e MAC, nonché le chiavi crittografiche da utilizzare per proteggere i dati inviati in un record SSL.

Il protocollo Handshake viene utilizzato prima della trasmissione di qualsiasi dato dell'applicazione.

È costituito da una serie di messaggi scambiati dal client e dal server, ogni messaggio contiene tre campi.

- **Type (1 byte):** indica un messaggio fra i dieci elencati nella tabella seguente.
- **Length (3 byte):** la lunghezza del messaggio in byte.
- **Content (≥ 0 byte):** i parametri associati a questo messaggio, anch'essi elencati nella tabella.

#### Tipi di messaggio e parametri del protocollo handshake

Tipo messaggio	Parametri
Hello_request	null
Client_hello	Version,random,sessioni d,chiper suite, compresion method
Server_hello	Version,random,sessioni d,chiper suite, compresion method
Certificate	Chain of X.509v3 certificates
Server_key_exchange	Parameters,signature
Certificate_request	Type,authorities

Server_done	null
Certificate_verify	Signature
Client_key_exchange	Parameters,signature
Finished	Hash value

La tabella illustra anche lo scambio di messaggi necessario per stabilire una connessione logica fra il client e server.

La procedura di handshake procede attraverso più fasi.

### **Fase 1. Inizializzazione delle funzionalità di sicurezza**

Questa fase viene utilizzata per avviare la connessione logica e per stabilire le funzionalità di sicurezza che possono essere impiegate. Lo scambio viene iniziato dal client che invia il messaggio client\_hello con i seguenti parametri.

- **Version** : versione di SSL più elevata fra quelle utilizzabili dal client
- **Random**: struttura casuale generata dal client costituita da un timestamp di 32 bit e 28 byte prodotti da un generatore sicuro di numeri casuali.
- **Session ID**: un identificatore di sessione di lunghezza variabile.
- **CipherSuite**: elenco che contiene gli algoritmi crittografici supportati dal client, in ordine decrescente di sicurezza.
- **Compression Method**: elenco dei metodi di compressione supportati dal client.

Dopo aver inviato il messaggio client\_hello, il client attende il messaggio **server\_hello** che contiene gli stessi parametri del proprio messaggio client\_hello. Per il messaggio server\_hello si applicano le seguenti convenzioni:

- il campo Version contiene la più bassa fra la versione suggerita dal client e la versione più elevata supportata dal server;
- il campo Random viene generato al server ed è indipendente dal campo Random del client.
- se il campo SessionID del client è diverso da zero, il server usa lo stesso valore, altrimenti il campo SessionID del server contiene il valore per una nuova sessione.

- il campo CipherSuite contiene il pacchetto di cifratura selezionato al server fra quelli proposti dal client.
- il campo Compression contiene il metodo di compressione selezionato dal server fra quelli proposti dal client.

Il primo elemento del parametro Cipher Suite è il metodo per lo scambio delle chiavi, ovvero il metodo con cui vengono scambiate le chiavi crittografiche per la crittografia convenzionale e il codice MAC. Per lo scambio delle chiavi sono supportati i seguenti metodi:

**RSA:** la chiave segreta viene crittografata con la chiave pubblica RSA del destinatario. Si deve rendere disponibile un certificato di chiave pubblica per la chiave del destinatario.

**Fixed Diffie-Hellman:** questo è uno scambio della chiave Diffie-Hellman in cui il certificato del server contiene i parametri pubblici Diffie-Hellman firmati all'autorità di certificazione. In pratica il certificato di chiave pubblica contiene i parametri della chiave pubblica Diffie-Hellman. Il client fornisce i propri parametri della chiave pubblica Diffie-Hellman in un certificato (se è richiesta l'autenticazione del client) oppure in un messaggio per lo scambio delle chiavi. Questo metodo produce come risultato una chiave segreta fissa fra i due nodi basata sul calcolo Diffie-Hellman utilizzando le chiavi pubbliche fisse.

**Ephemeral Diffie-Hellman:** questa tecnica viene utilizzata per creare chiavi segrete effimere (temporanee, monouso). In questo caso, vengono scambiate le chiavi pubbliche Diffie-Hellman, firmate utilizzando la chiave privata RSA o DSS del mittente. Il destinatario può verificare la firma utilizzando la corrispondente chiave pubblica. I certificati vengono utilizzati per autenticare le chiavi pubbliche. Questa sembra la più sicura delle tre opzioni Diffie-Hellman in quanto produce una chiave temporanea autenticata.

**Anonymous Diffie-Hellman:** viene utilizzato l'algoritmo base Diffie-Hellman, senza autenticazione, ovvero ciascuna parte invia all'altra parte i propri parametri pubblici Diffie-Hellman senza autenticazione. Questo approccio è vulnerabile ad attacchi man-in-the-middle in cui un estraneo esegue uno scambio anonimo Diffie-Hellman con entrambe le parti.

**Fortezza:** la tecnica definita per lo schema Fortezza.

Dopo la definizione del metodo per lo scambio delle chiavi, si trova CipherSpec, che include i seguenti campi.

- **CipherAlgorithm:** uno qualsiasi degli algoritmi menzionati in precedenza: RC4, RC2, DES, 3DES, DES40, IDEA, Fortezza.
- **MACAlgorithm:** MD5 o SHA-1.
- **CipherType:** Stream o Block.
- **IsExportable:** True o False
- **HashSize:** 0, 16 (per MD5) o 20 (per SHA-1) byte.
- **Key Material:** una sequenza di byte che contiene i dati utilizzati per la generazione delle chiavi di scrittura.
- **IV Size:** le dimensioni del vettore di inizializzazione per la crittografia CBC (Cipher Block Chaining).

## Fase 2. Autenticazione del server e scambio delle chiavi

Il server inizia questa fase inviando il proprio certificato, sempre che debba essere autenticato; il messaggio contiene un certificato X.509 oppure una catena di certificati. Il messaggio **certificate** è obbligatorio per ogni metodo concordato per lo scambio di chiavi tranne che per lo scambio Diffie-Hellman anonimo. Si noti che se viene utilizzato lo scambio fixed Diffie-Hellman, questo messaggio certificate funge da messaggio per lo scambio delle chiavi del server poiché contiene i parametri Diffie-Hellman pubblici del server.

A seguire, se necessario, può essere inviato un messaggio **server\_key\_exchange**. Questo non è necessario in due casi: quando il server ha inviato un certificato con i parametri fixed Diffie-Hellman o quando deve essere utilizzato lo scambio delle chiavi RSA. Il messaggio **server\_key\_exchange** è obbligatorio invece nei seguenti casi:

**Anonymous Diffie-Hellman:** il contenuto del messaggio è costituito dai due valori globali Diffie-Hellman (un numero primo e una sua radice primitiva) più la chiave Diffie-Hellman pubblica del server.

**Ephemeral Diffie-Hellman:** il contenuto del messaggio include i tre parametri Diffie-Hellman necessari per la forma Diffie-Hellman anonima più una firma di questi parametri.

Avviene uno scambio delle chiavi RSA caratterizzato dal fatto che il server usa solo la chiave RSA di firma: in questo caso il client non può semplicemente inviare una chiave segreta crittografata con la chiave pubblica del server. Al contrario il server deve creare una coppia temporanea di chiavi

RSA pubblica/privata e utilizzare il messaggio `server_key_exchange` per inviare la chiave pubblica. Il contenuto del messaggio include i due parametri della chiave pubblica temporanea RSA (esponente e modulo), più una firma di questi parametri.

### **Fortezza.**

Occorre fornire ulteriori dettagli sulle firme. Come di consueto, una firma viene creata prendendo il valore hash di un messaggio e crittografandolo con la chiave privata del mittente. In questo caso il codice hash è definito nel seguente modo:

```
hash(ClientHello.random ServerHello.random ServerParams)
```

Dunque il codice hash non copre solo i parametri Diffie-Hellman o RSA ma anche i due nonce dei messaggi hello iniziali. Questo evita gli attacchi a replay. Nel caso di una firma DSS, il calcolo hash viene eseguito utilizzando l'algoritmo SHA-1. Nel caso di una firma RSA, vengono calcolati entrambi i valori hash MD5 e SHA-1 e il concatenamento di questi due valori hash (36 byte) viene crittografato utilizzando la chiave privata del server.

Quindi un server non anonimo (cioè un server che non utilizza la forma Diffie-Hellman anonima) può richiedere un certificato al client. Il messaggio `certificate_request` include due parametri: `certificate_type` e `certificate_authorities`. Il `certificate_type` indica l'algoritmo a chiave pubblica e il suo impiego.

- RSA, solo firma.
- DSS, solo firma.
- RSA per fixed Diffie-Hellman; in questo caso la firma viene utilizzata solo per l'autenticazione inviando un certificato firmato con RSA.
- DSS per fixed Diffie-Hellman; usato anch'esso solo per l'autenticazione.
- RSA per Ephemeral Diffie-Hellman.
- DSS per Ephemeral Diffie-Hellman.
- Fortezza.

Il secondo parametro del messaggio `certificate_request` è un elenco delle autorità di certificazione accettate.

L'ultimo messaggio della Fase 2 è obbligatorio ed è il messaggio **server\_done** che viene inviato dal server per indicare la fine dei messaggi di hello del server. Dopo aver inviato questo messaggio, il server attende una risposta del client. Questo messaggio non ha alcun parametro.

### Fase 3. Autenticazione del client e scambio delle chiavi

Dopo aver ricevuto il messaggio `server_done`, il client deve verificare che il server abbia fornito un certificato valido (se richiesto) e deve controllare che i parametri di `server_hello` siano accettabili. Se tutto è soddisfacente, il client invia al server uno o più messaggi.

Se il server ha richiesto un certificato, il client inizia questa fase inviando un messaggio **certificate**. Se non è disponibile alcun certificato adatto, il client invia l'allarme `no_certificate`.

Poi viene il messaggio **client\_key\_exchange** che deve essere inviato in questa fase. Il contenuto del messaggio dipende dal tipo di scambio delle chiavi.

**RSA:** il cliente genera un valore segreto pre-master di 48 byte e ne esegue la crittografia con la chiave pubblica tratta dal certificato del server o con la chiave RSA temporanea tratta da un messaggio `server_key_exchange`. Il suo uso per calcolare il codice segreto master verrà descritto più avanti.

**Ephemeral o Anonymous Diffie-Hellman:** vengono inviati i parametri pubblici Diffie-Hellman del client.

**Fixed Diffie-Hellman:** i parametri Diffie-Hellman pubblici del client sono stati precedentemente inviati in un messaggio `certificate` e dunque il contenuto di questo messaggio è nullo.

**Fortezza:** vengono inviati i parametri Fortezza del client.

Infine in questa fase, il client può inviare un messaggio `certificate_verify` per fornire una verifica esplicita di un certificato del client. Questo messaggio viene inviato solo dopo un certificato del client che ha funzionalità di firma (ovvero tutti i certificati tranne quelli contenenti i parametri `fixed Diffie-Hellman`).

### Fase 4. Fine.

Questa fase completa l'impostazione di una connessione sicura. Il client invia un messaggio `change_cipher_spec` e copia il `CipherSpec` temporaneo nel `CipherSpec` corrente. Si noti che questo messaggio non è parte del protocollo Handshake, ma viene inviato utilizzando il protocollo Change Cipher Spec. Il client invia immediatamente il messaggio `finished` con i nuovi algoritmi, chiavi e segreti. Il messaggio `finished` verifica che i processi di scambio delle chiavi e di autenticazione abbiano avuto successo. Il contenuto del messaggio `finished` è il concatenamento dei due valori hash:

`MD5(master_secret || pad2 || MD5(handshake_messages || Sender || master_secret || pad1))`

`SHA(master_secret || pad2 || SHA(handshake_messages || Sender || master_secret || pad1))`

dove Sender è un codice che identifica che il mittente è il client e handshake\_messages è costituito a tutti i dati di tutti i messaggi di handshake fino a questo messaggio escluso.

In risposta a questi due messaggi, il server invia il proprio messaggio *change\_cipher\_spec*, trasferisce il CipherSpec provvisorio nel CipherSpec corrente e invia il proprio messaggio finished. A questo punto la procedura di handshake è completa e il client e il server possono iniziare a scambiarsi i dati a livello applicazione.

### **Calcoli crittografici**

Per l'applicazione della tecnologia SSL sono cruciali altri 2 concetti: la creazione di un valore segreto master condiviso tramite lo scambio di chiavi e la generazione dei relativi parametri crittografici.

#### **La creazione del valore segreto master**

Il valore segreto master condiviso è un valore monouso di 48 byte (384 bit) generato per una sessione tramite scambio di chiavi sicuro.

La creazione si svolge in due fasi. Innanzitutto viene scambiato il valore pre\_master\_secret, poi entrambe le parti calcolano il valore master\_secret.

Per lo scambio del valore pre\_master\_secret vi sono due possibili modalità:

- **RSA:** il client genera un valore pre\_master\_secret di 48 byte, crittografato con la chiave RSA pubblica del server e la invia al server. Il server esegue la decrittografia del testo cifrato utilizzando la propria chiave privata per recuperare il valore pre\_master\_secret.
- **Diffie-Hellman:** sia il client che il server generano una chiave pubblica Diffie-Hellman. Dopo aver scambiato queste chiavi, ognuno dei due svolge il calcolo Diffie-Hellman per creare il valore pre\_master\_secret.

#### **Generazione dei parametri crittografici**

CipherSpecs richiede un segreto MAC di scrittura per il client, un segreto MAC di scrittura per il server, una chiave di scrittura per il client, una chiave di scrittura per il server, un vettore di inizializzazione di scrittura per il client e un vettore di inizializzazione di scrittura per il server che vengono generati dal valore segreto master in questo ordine. Questi parametri vengono generati dal valore segreto master tramite calcoli hash che producono parametri di lunghezza appropriata.

#### 2.1.6 TLS, Transport Layer Security

TLS (Transport Layer Security) è un protocollo di livello 5 (sessione) che opera quindi al di sopra del protocollo di livello trasporto. È uno standard IETF e deriva dal protocollo SSL (Secure Socket Layer). TLS 1.0 (la versione attuale) è molto simile a SSL 3.023 tuttavia non è compatibile con questo, le implementazioni comunque supportano generalmente sia TLS 1.0 sia SSL 3.0, ed è garantita l'interoperabilità.

Lo scopo di TLS è permettere una comunicazione sicura tra due applicazioni, fornendo al flusso di dati tra di esse servizi di autenticazione, integrità e riservatezza.

Si compone di due livelli: il TLS Record Protocol e il TLS Handshake Protocol.

Il primo è il livello più basso, che opera direttamente al di sopra di un protocollo di trasporto affidabile come TCP ed è usato per incapsulare (offrendo servizi di sicurezza) protocolli di livello superiore tra cui l'Handshake Protocol stesso; quest'ultimo si occupa della fase di negoziazione in cui si autentica l'interlocutore e si stabiliscono le chiavi segrete condivise.

TLS è indipendente dal protocollo di livello applicazione che ne utilizza i servizi, tuttavia lo standard non specifica come l'applicazione lo debba utilizzare, ovvero come decidere di iniziare un handshake TLS e come interpretare i certificati scambiati per l'autenticazione delle parti.

### **TLS Record Protocol**

Il TLS Record Protocol riceve i dati dal livello superiore, li suddivide in blocchi di dimensioni opportune, eventualmente li comprime, calcola un MAC, cifra il tutto e trasmette il risultato di questa elaborazione. I dati in ricezione vengono, nell'ordine, decifrati, verificati (verifica del MAC), decompressi, riassemblati e infine consegnati al livello superiore. I tipi di dati che possono essere consegnati al Record Protocol (e che quindi identificano il protocollo di livello superiore) sono quattro, ovvero: ``handshake protocol'', ``alert protocol'', ``change cipher spec protocol'' e ``application data protocol''; i primi tre fanno parte del TLS Handshake Protocol e saranno descritti successivamente, mentre l'ultimo indica che si tratta di dati del livello applicazione, ovvero del protocollo che sta sopra a TLS.

Il Record Protocol opera sempre all'interno di uno stato, che definisce gli algoritmi di compressione, cifratura e autenticazione e i parametri relativi (come le chiavi). Alla connessione sono sempre associati quattro di questi stati: gli stati correnti in lettura e in scrittura e i corrispondenti stati pendenti. I dati vengono elaborati secondo gli stati correnti, mentre il TLS Handshake Protocol può impostare i parametri per gli stati pendenti e rendere corrente uno stato

pendente. In quest'ultimo caso lo stato pendente viene re-inizializzato a uno stato vuoto (che non può diventare corrente); lo stato iniziale specifica che non si utilizza nessun algoritmo di compressione, cifratura e autenticazione.

I parametri di sicurezza definiti per uno stato sono i seguenti:

- **connection end**: specifica se l'entità in questione è il client o il server (in TLS c'è sempre questa distinzione);
- **bulk encryption algorithm**: indica l'algoritmo di cifratura e i relativi parametri, come la lunghezza della chiave o il fatto che sia "esportabile" oppure no;
- **MAC algorithm**: indica l'algoritmo di autenticazione e i relativi parametri;
- **compression algorithm**: indica l'algoritmo di compressione e i relativi parametri;
- **master secret**: sequenza segreta di 48 byte condivisa dai due interlocutori;
- **client random**: 32 byte casuali forniti dal client;
- **server random**: 32 byte casuali forniti dal server.

A partire da questi parametri vengono generati sei valori, ovvero le chiavi per la cifratura e per l'autenticazione e il vettore di inizializzazione (quest'ultimo solo per cifrari a blocco in modalità CBC), sia per il client che per il server; ognuno dei due interlocutori utilizza i propri parametri (parametri in scrittura) in fase di trasmissione e quelli dell'altra parte (parametri in lettura) in fase di ricezione.

Una volta che i parametri di sicurezza sono stati impostati e le chiavi sono state generate, gli stati della connessione possono essere istanziati rendendoli correnti.

Gli stati devono essere aggiornati ad ogni record elaborato, e ciascuno comprende le seguenti informazioni: stato della compressione (stato corrente dell'algoritmo di compressione), stato della cifratura (comprende la chiave e altre informazioni necessarie a definire lo stato dell'algoritmo, per esempio l'ultimo blocco nel caso di un cifrario a blocco in modalità CBC), MAC secret e numero di sequenza.

Come detto in precedenza, il TLS Record Protocol riceve dei dati in modo opaco dal livello superiore, li suddivide in blocchi, applica un algoritmo di compressione, poi inserisce un MAC (calcolato tramite un hash sulla chiave segreta, il numero di sequenza, il frammento compresso e altri parametri) e infine cifra il tutto, MAC compreso. Si veda in proposito la figura 6

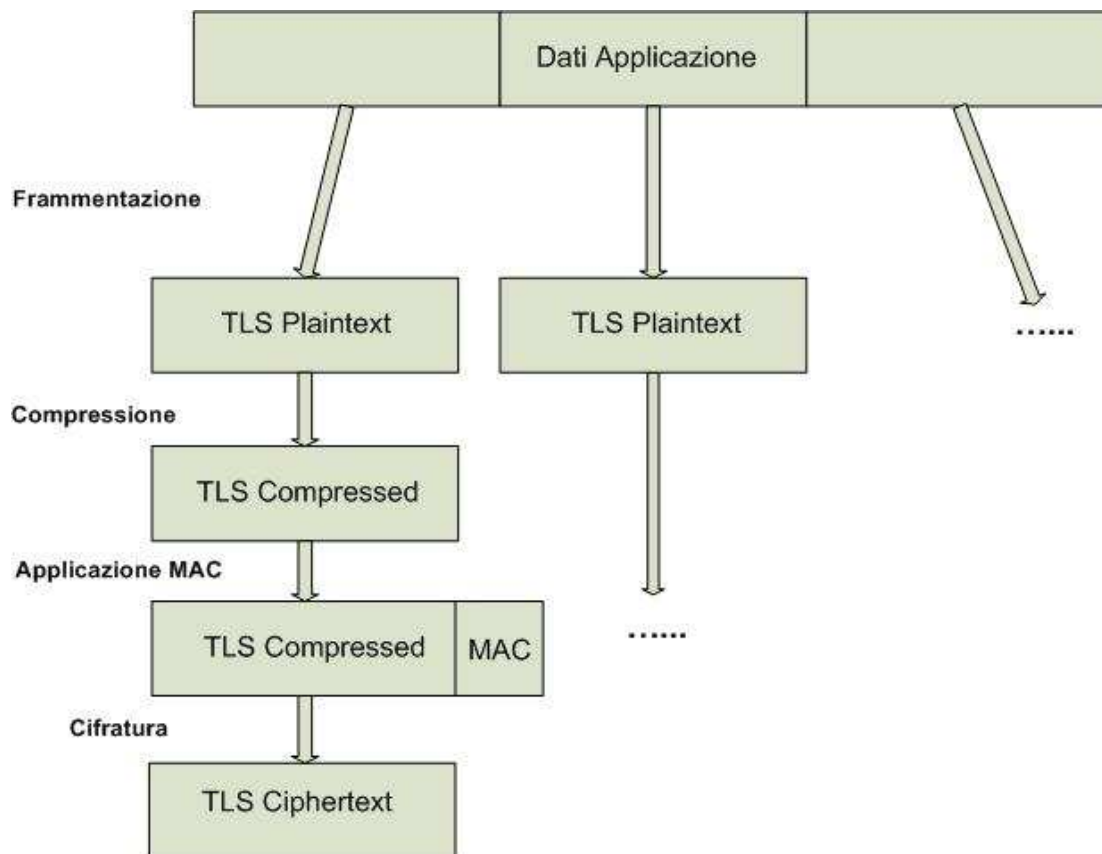


Figura 6 TSL Record Protocol

## TLS Handshake Protocol

Il TLS Handshake Protocol è costituito da tre sotto-protocolli (*Change Cipher Spec*, *Alert* e *Handshake*), che permettono ai due interlocutori di autenticarsi, negoziare i parametri di sicurezza, istanziare i parametri negoziati e notificare condizioni di errore. Il TLS Handshake Protocol è responsabile della negoziazione di una sessione, che è costituita dai seguenti parametri: *Session Identifier* (identificatore della sessione scelto dal server), "Peer Certificate" (certificato X.509 dell'interlocutore -- può mancare), Compression method (algoritmo di compressione), Cipher Spec (algoritmi di cifratura e autenticazione e relativi parametri crittografici), master secret, *is resumable* (flag che indica se la sessione può essere utilizzata per iniziare nuove connessioni). Da questi sono ricavati i parametri di sicurezza visti in precedenza, che vengono utilizzati dal Record Protocol. La stessa sessione, come suggerito dalla presenza del flag *is resumable*, può essere utilizzata per creare più connessioni.

### 1. Change cipher spec

Il Change Cipher Cspec protocol consiste in un solo messaggio, elaborato secondo lo stato corrente, che indica di passare al nuovo stato negoziato. Subito dopo aver mandato questo messaggio il mittente copia lo stato pendente in scrittura in quello corrente, mentre la ricezione del messaggio fa sì che l'attuale stato pendente in lettura divenga quello corrente.

## **2. Alert**

TLS supporta una serie di messaggi di controllo ed errore, che possono essere inviati per notificare all'interlocutore una particolare situazione o evento. Sono presenti ad esempio messaggi per notificare la chiusura della connessione, un errore nella verifica del MAC o nella decifrazione, vari tipi di errore relativi ai certificati. I messaggi di alert, come tutti i dati trasmessi, vengono elaborati secondo lo stato corrente, e a seconda della loro gravità sono classificati come *fatal* (questi messaggi provocano sempre l'interruzione immediata della connessione) oppure come *warning*.

## **3. Handshake**

L'Handshake protocol è utilizzato per negoziare i parametri di sicurezza di una sessione.

La specifica fa notare che le applicazioni non dovrebbero contare sul fatto che TLS stabilisca la connessione più sicura possibile tra i due interlocutori, perché sono possibili alcuni attacchi che mirano a far accordare le due parti su un basso livello di sicurezza anche se sarebbe possibile utilizzarne uno maggiore. Per questo l'applicazione dovrebbe essere consapevole del livello di sicurezza richiesto e in base a questo decidere come comportarsi rispetto a ciò che è stato negoziato. Con il primo messaggio, ClientHello, il client indica quali algoritmi supporta in ordine di preferenza e inserisce un valore casuale; il server risponde con un ServerHello indicando quali algoritmi (compressione, cifratura, autenticazione) ha scelto, fornendo inoltre il codice identificativo della sessione e un numero casuale. Se il metodo di autenticazione scelto non è anonimo, il server invia poi il proprio certificato (messaggio Certificate); se invece è anonimo oppure se il certificato è valido solo per la firma, invia un messaggio ServerKeyExchange che contiene una chiave pubblica RSA, oppure un valore per effettuare uno scambio Diffie-Hellman. Se il server non è anonimo può poi richiedere un certificato al client con il messaggio CertificateRequest (nel quale può indicare i tipi di certificato e le certification authority che accetta), infine manderà il messaggio ServerHelloDone, che indica al client di procedere a sua volta. A questo punto il client invia il proprio certificato (se richiesto) e poi il messaggio ClientKeyExchange, che contiene il premaster secret (generato casualmente dal client) cifrato con la chiave RSA fornita dal server, oppure un valore per completare lo scambio Diffie-Hellman la cui chiave sarà il premaster secret. Se il client ha mandato un certificato che permette la firma invia poi il messaggio CertificateVerify come verifica esplicita del proprio certificato. A questo punto le parti

si sono autenticate e sono in possesso dei dati necessari per calcolare il master secret (generato a partire dal premaster secret e dai valori casuali precedentemente scambiati), quindi il client manderà un messaggio ChangeCipherSpec e copierà lo stato pendente appena negoziato in quello corrente. I parametri del nuovo stato sono utilizzati per la prima volta con il messaggio Finished, che contiene dei dati che permettono di verificare la correttezza dell'Handshake; il server risponde a sua volta con un messaggio ChangeCipherSpec e un messaggio Finished. Con quest'ultimo scambio l'handshake può dirsi terminato e le due parti possono iniziare a scambiarsi i dati (application data).

Se successivamente i due interlocutori dovessero decidere di riprendere una vecchia sessione oppure di duplicare la sessione esistente potrebbero utilizzare un handshake abbreviato: il client inizierebbe con un messaggio ClientHello, in cui verrebbe incluso l'identificatore della sessione in corso, il server risponderebbe con un ServerHello e procederebbe subito con i messaggi ChangeCipherSpec e Finished, infine anche il client invierebbe i messaggi ChangeCipherSpec e Finished. Questa procedura può essere avviata anche dal server con un messaggio di HelloRequest, i passaggi successivi sarebbero poi quelli appena descritti.

Le differenze tra SSL 3.0 e TLS 1.0 sono poche e riguardano aspetti secondari, tuttavia sono sufficienti a far sì che i due protocolli non siano tra loro compatibili. Generalmente però vengono implementati entrambi, per cui l'interoperabilità funziona come segue: se un server che non supporta TLS riceve una richiesta di connessione con numero di versione 3.1 (che, per ragioni storiche, identifica TLS 1.0) risponderà che la massima versione supportata è la 3.0, per cui l'Handshake potrà continuare con SSL 3.0. Analogamente, un server che supporta TLS risponderà a una richiesta SSL 3.0 con un handshake SSL.

Rispetto a SSL 3.0, TLS definisce molti più messaggi di errore, inoltre presenta alcune piccole modifiche volte ad aumentarne la sicurezza dal punto di vista crittografico: ci sono per esempio delle differenze nel calcolo del MAC (sia nell'algoritmo che nei campi coinvolti) e nell'espansione del master secret per la generazione delle chiavi (TLS usa una funzione pseudocasuale).

### **2.1.6 La crittografia della chiave pubblica in SSL**

Riassumendo, la tabella xx riporta le varie operazioni di crittografia a chiave pubblica effettuate dal client e dal server in differenti modi a seconda del tipo di handshake utilizzato.

Si distinguono due casi:

- Senza autenticazione del client
  1. RSA Handshake
 

Il client effettua due operazioni RSA a chiave pubblica – una per verificare il certificato del server e un altro per crittografare la premaster secret con la chiave pubblica del server. Il server effettua solo una RSA operazione di chiave privata per decrittografare il messaggio di *Client Exchange* e recuperare la premaster secret.
  2. ECDH-ECDSA Handshake
 

Il client effettua una verifica ECDSA per accertare il certificato ECDSA del server, una operazione usa la chiave privata ECDH e la chiave pubblica del server computa la premaster condivisa. Tutti i server necessitano di effettuare un'operazione ECDH per arrivare al segreto condiviso.

Tabella 1: operazioni crittografiche in una SSL handshake

	<b>RSA</b>	<b>ECDH-ECDSA</b>
Client	$RSA_{verify} + RSA_{encrypt}$	$ECDSA_{verify} + ECDH_{op}$
Server	$RSA_{decrypt}$	$ECDH_{op}$

(a) Solo autenticazione del server

	<b>RSA</b>	<b>ECDH-ECDSA</b>
Client	$RSA_{verify} + RSA_{encrypt} + RSA_{sign}$	(i) $ECDSA_{verify} + ECDH_{op}$ (ii) $ECDSA_{verify} + ECDSA_{sign} + ECDH_{op}$
Server	$2 * RSA_{verify} + RSA_{decrypt}$	(i) $ECDSA_{verify} + ECDH_{op}$ (ii) $2 * ECDSA_{verify} + ECDH_{op}$

(b) entrambi (client e server) sono autenticati

- Con l'autenticazione del client:
  1. RSA Handshake
 

Il client effettua due operazioni di chiave pubblica RSA (come senza l'autenticazione del client) ma in più effettua un'operazione di chiave privata RSA per generare il messaggio *Certificate verify*. Il server effettua due operazioni a chiave pubblica RSA (una verifica il *client's certificate* e un'altra per verificare la signature del client nel messaggio *Certificate verify*) e una operazione di chiave privata per decrittografare la premaster secret.
  2. ECDH-ECDSA Handshake
    - (i) Quando il client usa un certificato ECDH, entrambi i siti fanno una operazione di verifica ECDSA sull'altro certificato, seguita da un'operazione ECDH per computare la premaster secret.

(ii) Quando il client usa un certificato ECDSA, l'operazione richiede che uno dei due site sia asimmetrico. Il client effettua una verifica ECDSA del certificato del server, un'operazione ECDH per computare la premaster secret e una signature ECDSA per generare il messaggio *Certificate Verify*. Il server esegue una operazione ECDH per computare la premaster secret e due verifiche ECDSA, una per verificare il certificato del client e un'altra per verificare il messaggio *Certificate Verify*.

## 2.2 Valutazione delle performance

Il presente studio si proponeva di descrivere le strutture matematiche alla base della crittografia ellittica e di illustrarne un'applicazione pratica.

Come caso di test si è scelto allora di implementare un server Web con certificati e in particolare di svilupparne una duplice realizzazione: in un caso si sfruttavano algoritmi di crittografia ellittica, nell'altro certificati RSA, l'obiettivo era misurare e comparare le differenti prestazioni offerte dai 2 sistemi di certificazione, studiare l'impatto di efficienza dell'impiego di ECC nei confronti di RSA implementato nel protocollo SSL.

Basata sulla PKI, una transazione HTTPS coinvolge molte altre operazioni quali la crittografia simmetrica, hashing, paring dei messaggi e accesso ai file system.

Il costo dell'encryption e dell'hashing dipende dalla quantità dei dati trasferiti.

Per avere una realistica stima di SSL performance è importante l'utilizzo di un buon carico di lavoro.

### 2.2.1 Descrizione dell'ambiente

Al fine di poter misurare realmente le performance è stato implementato un ambiente in totale ambito open source, sfruttando gli strumenti messi a disposizione dalla suite Openssl. L'ambiente comprende le seguenti componenti:

- Software S.O.
  1. Ubuntu 8.04 Hardy come server di software di virtualizzazione
  2. VMWARE Server 2.0 per quanto riguarda la virtualizzazione di sistemi operativi;
  3. Ubuntu 8.04 Hardy come server Web senza interfaccia grafica per ragioni di efficienza;
  4. Ubuntu 8.04 Hardy come client di simulazione di connessione.

- Software applicativo
  1. Openssl versione 0.9.8.j,
  2. Apache vers. 2.2.2 con ECC pach “enable-ecc-in-modssl-20060725171010.patch”
  3. gcc vers 4.2 come compilatore
  
- Hardware
  1. Pentium4 - 2,4 Ghz “ Gb RAM
  2. Laptop Acer TravelMate 4230
  3. I sistemi sono collegati a mezzo di switch a 100 Mbit

Nella Figura 7 si può osservare lo schema architetturale dell’ambiente:

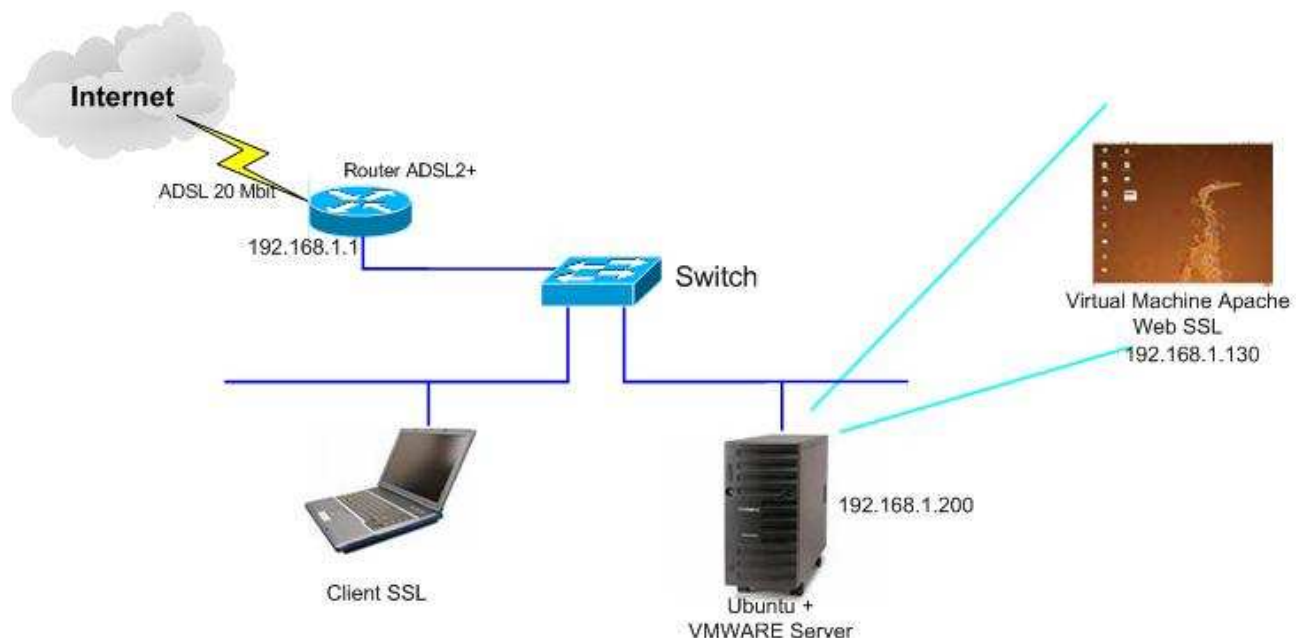


Figura 7 Ambiente di test

### 2.2.2 Specifiche di installazione dell’ambiente

Dopo numerosi tentativi, rivelatisi infruttuosi, di utilizzare l’ultima versione di Apache ossia la 2.2.11, è stata utilizzata la versione 2.2.2 con ECC patch chiamata “enable-ecc-in-modssl-20060725171010.patch”, indispensabile per supportare la crittografia ellittica nel modulo “mod\_ssl”, seguendo le precise procedure di compilazione e installazione della documentazione di Apache.

Per snellire l’utilizzo dei certificati è stata utilizzata la procedura self-signed nel seguente modo:

generazione della chiave, server.key, con curva ellittica prime256v1 (The NSA's Suite-B usa "prime256v1" ((aka secp256r1) per il traffico "SECRET" e secp384r1 per il traffico "TOP SECRET" )

```
openssl ecparam -name prime256v1 -genkey -out server.key
```

Comando di richiesta del certificato prima creato,

```
openssl req -new -key server.key -out server.csr
```

Generazione del certificato auto-firmato,

```
openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt
```

Con il comando possiamo vedere se il certificato è stato correttamente generato:

```
./openssl x509 -text -in server.crt
```

Certificate:

Data:

Version: 1 (0x0)

Serial Number:

b2:46:df:fd:8e:24:a1:e1

**Signature Algorithm: ecdsa-with-SHA1**

Issuer: C=IT, ST=Roma, L=Roma, O=max, OU=max, CN=massimov.it

Validity

Not Before: Mar 3 17:34:51 2009 GMT

Not After : Mar 3 17:34:51 2010 GMT

Subject: C=IT, ST=Roma, L=Roma, O=max, OU=max, CN=massimov.it

Subject Public Key Info:

Public Key Algorithm: id-ecPublicKey

EC Public Key:

pub:

04:b1:ed:c7:5f:2d:3a:dd:5e:fd:38:fd:e9:a6:56:  
3d:a3:72:52:9b:a7:b7:36:6c:21:d6:cf:f4:50:71:  
78:53:82:cd:c6:2e:53:de:be:96:36:06:50:05:60:  
eb:eb:65:a8:84:8e:60:e6:31:87:3d:b0:88:16:f1:  
2e:22:d3:82:3a

ASN1 OID: **prime256v1**

**Signature Algorithm: ecdsa-with-SHA1**

30:44:02:20:62:f3:00:54:45:3b:3f:ba:36:51:30:7f:31:ca:  
e2:34:1b:2c:84:eb:57:ad:1b:82:32:d2:f9:62:c6:07:f1:fa:  
02:20:2d:22:fd:6d:d5:b5:35:d0:df:76:29:3f:f8:dc:60:c0:  
21:ce:33:6a:ff:5a:71:57:2b:6a:e8:b8:e9:29:14:c5

-----BEGIN CERTIFICATE-----

```
MIIBpzCCAU8CCQCyRt/9jiSh4TAJBgcqhkjOPQQBMF0xCzAJBgNVBAYTAklUMQ0w
CwYDVQQIEwRSb21hMQ0wCwYDVQQHEwRSb21hMQwwCgYDVQQKEwNtYXgxDDAKBgNV
BAsTA21heDEUMBGA1UEAxMLbWFzc2ltb3YuaXQwHhcNMDkwMzAzMTczNDUxWhcN
MTAwMzAzMTczNDUxWjBdMQswCQYDVQQGEwJJVDENMA5GA1UECBMEU9tYTENMA5G
A1UEBxMEU9tYTEMMAoGA1UEChMDbWFzc2ltb3YuaXQwCgYDVQQLEwNtYXgxFDASBgNVBAMT
C21he3NpbW92Lml0MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEse3HXy063V79
OP3ppLY9o3JSm6e3Nmwh1s/0UHF4U4LNxi5T3r6WNgZQBWDr62WohI5g5jGHPbCI
FvEuItOCOjAJBgcqhkjOPQQBA0cAMEQCIGLzAFRFOz+6NIEwFzHK4jQbLITrV60b
gjLS+WLGB/H6AiAtIv1t1bU10N92KT/43GDAlc4zav9acVcraui46SkUxQ==
```

-----END CERTIFICATE-----

Successivamente sono stati aggiunti i seguenti parametri nel file **httpd-ssl.conf**

```
SSLCertificateFile "/usr/local/apache2/certs/server.crt"
SSLCertificateKeyFile "/usr/local/apache2/certs/server.key"
```

infine sono stati imposti gli algoritmi crittografici da utilizzare sempre nel file di configurazione con le istruzioni,

Dopo lo start del servizio è stata effettuata la connessione dal client con il seguente comando, specificando precisamente la cifratura utilizzata:

```
openssl s_client -connect 192.168.1.30:443 -cipher ECDHE-ECDSA-AES256-SHA
```

Viene riportato l'output dell'avvenuto SSL-Handshake e della crittografia usata, si vedano evidenziate in grassetto le righe più significative:

```
depth=0 /C=IT/ST=Roma/L=Roma/O=max/OU=max/CN=massimov.it
verify error:num=18:self signed certificate
verify return:1
depth=0 /C=IT/ST=Roma/L=Roma/O=max/OU=max/CN=massimov.it
verify return:1
---
Certificate chain
 0 s:/C=IT/ST=Roma/L=Roma/O=max/OU=max/CN=massimov.it
  i:/C=IT/ST=Roma/L=Roma/O=max/OU=max/CN=massimov.it
---
Server certificate
-----BEGIN CERTIFICATE-----
MIIBpzCCAUS8CCQCyRt/9jiSh4TAJBgcqhkjOPQQBMF0xCzAJBgNVBAYTAklUMQ0w
CwYDVQQIEwRSb21hMQ0wCwYDVQQHEwRSb21hMQwwCgYDVQQKEwNtYXgxDAAKBgNV
BAAsTA21heDEUMBIGAlUEAxMLbWFzc2ltb3YuaXQwHhcNMDkwMzAzMTczNDUxWhcN
MTAwMzAzMTczNDUxWjBdMQswCQYDVQQGEwJJVDENMA5GA1UECBMEU9tYTENMA5G
A1UEBxMEU9tYTEMMAoGA1UEChMdbWF4MzAzMTczNDUxWjBdMQswCQYDVQQLEwNt
YXgxDAAKBgNVBAMT
C21hc3NpbW92Lml0MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEse3HXy063V79
OP3pplY9o3JSm6e3Nmwh1s/0UHF4U4LNxi5T3r6WNgZQBWDr62WohI5g5jGHPbCI
FvEuItOCOjAJBgcqhkjOPQQBA0cAMEQCIGLzAFRFOz+6NIEwfzHK4jQbLITrV60b
gjLS+WLGB/H6AiAtIv1t1bU10N92KT/43GDAIc4zav9acVcraui46SkUxQ==
-----END CERTIFICATE-----
subject=/C=IT/ST=Roma/L=Roma/O=max/OU=max/CN=massimov.it
issuer=/C=IT/ST=Roma/L=Roma/O=max/OU=max/CN=massimov.it
---
No client certificate CA names sent
---
SSL handshake has read 740 bytes and written 180 bytes
---
```

**New, TLSv1/SSLv3, Cipher is ECDHE-ECDSA-AES256-SHA**

Server public key is 256 bit

Compression: NONE

Expansion: NONE

SSL-Session:

**Protocol : TLSv1**

**Cipher : ECDHE-ECDSA-AES256-SHA**

Session-ID: 8A9D2989B96F5CA0C989420003F695A8EEAF6A803C135C4A122246E70A8CA3D7

Session-ID-ctx:

Master-Key:

C1447FD4668465DEF83A1F48FDD498484694191CAB4BD3557F5FE853200F39DC73B228DB4C353E1FB  
BD3DDC13116157F

Key-Arg : None

Start Time: 1236329775

Timeout : 300 (sec)

Verify return code: 18 (self signed certificate)

E il log del server:

```
[Fri Mar 06 09:19:49 2009] [info] [client 88.41.252.232] Connection to child 1 established (server massimov.it:443)
[Fri Mar 06 09:19:49 2009] [info] Seeding PRNG with 136 bytes of entropy
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_kernel.c(1779): OpenSSL: Handshake: start
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_kernel.c(1787): OpenSSL: Loop: before/accept initialization
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1775): OpenSSL: read 11/11 bytes from BIO#82c2f80 [mem: 82df200] (BIO dump follows)
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1722): +-----+
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0000: 16 03 01 00 a3 01 00 00-9f 03 01          |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1753): +-----+
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1775): OpenSSL: read 157/157 bytes from BIO#82c2f80 [mem: 82df20b] (BIO dump follows)
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1722): +-----+
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0000: 49 b0 e4 34 bf 83 64 89-1f 07 5e 8c 8e 03 a8 de I..d..^.... |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0010: f4 80 92 a9 09 1d 46 a2-6e 27 65 30 2c 3d e3 0a .....F.n'e0,.. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0020: 00 00 44 c0 0a c0 14 00-88 00 87 00 39 00 38 c0 ..D.....9.8. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0030: 0f c0 05 00 84 00 35 c0-07 c0 09 c0 11 c0 13 00 .....5..... |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0040: 45 00 44 00 33 00 32 c0-0c c0 0e c0 02 c0 04 00 E.D.3.2..... |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0050: 41 00 04 00 05 00 2f c0-08 c0 12 00 16 00 13 c0 A....//..... |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0060: 0d c0 03 fe ff 00 0a 01-00 00 32 00 00 00 18 00 .....2..... |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0070: 16 00 00 13 6d 61 73 73-69 6d 6f 76 2e 64 79 6e ....massimov.dyn |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0080: 64 6e 73 2e 6f 72 67 00-0a 00 08 00 06 00 17 00 dns.org..... |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0090: 18 00 19 00 0b 00 02 01-00 00 23          .....# |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1751): | 0157 - <SPACES/NULS>
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1753): +-----+
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_kernel.c(1787): OpenSSL: Loop: SSLv3 read client hello A
```

```

[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_kernel.c(1787): OpenSSL: Loop: SSLv3 write server hello A
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_kernel.c(1787): OpenSSL: Loop: SSLv3 write certificate A
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_kernel.c(1169): [client 88.41.252.232] handing out temporary 256 bit ECC key
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_kernel.c(1787): OpenSSL: Loop: SSLv3 write key exchange A
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_kernel.c(1787): OpenSSL: Loop: SSLv3 write server done A
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_kernel.c(1787): OpenSSL: Loop: SSLv3 flush data
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1775): OpenSSL: read 5/5 bytes from BIO#82c2f80 [mem: 82df200] (BIO dump follows)
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1722): +-----+
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0000: 16 03 01 00 46                ....F      |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1753): +-----+
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1775): OpenSSL: read 70/70 bytes from BIO#82c2f80 [mem: 82df205] (BIO dump follows)
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1722): +-----+
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0000: 10 00 00 42 41 04 73 36-58 49 b4 b2 87 47 ea 26 ...BA.s6XL..G.& |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0010: a8 11 d8 49 4d ff 7c 62-a1 c5 92 60 da 0e 2f c6 ...IM.|b...^../. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0020: e2 1c 2e 3f ce 66 71 e7-87 29 41 33 98 7d 84 01 ...?fq..)A3.}|
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0030: 6c 9b 8f c2 88 80 06 00-4c cf 9a e0 bf 03 35 49 1.....L.....5I |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0040: 18 82 83 e2 9a 4a                .....J      |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1753): +-----+
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_kernel.c(1787): OpenSSL: Loop: SSLv3 read client key exchange A
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1775): OpenSSL: read 5/5 bytes from BIO#82c2f80 [mem: 82df200] (BIO dump follows)
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1722): +-----+
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0000: 14 03 01 00 01                .....      |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1753): +-----+
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1775): OpenSSL: read 1/1 bytes from BIO#82c2f80 [mem: 82df205] (BIO dump follows)
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1722): +-----+
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0000: 01                .          |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1753): +-----+
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1775): OpenSSL: read 5/5 bytes from BIO#82c2f80 [mem: 82df200] (BIO dump follows)
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1722): +-----+
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0000: 16 03 01 00 30                ....0      |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1753): +-----+
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1775): OpenSSL: read 48/48 bytes from BIO#82c2f80 [mem: 82df205] (BIO dump follows)
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1722): +-----+
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0000: bb 9e f8 87 78 e7 2c dd-81 35 18 e5 d0 ca 66 c6 ...x...5...f. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0010: bd 64 81 0c 38 cc 17 1f-dd 07 d6 83 11 1d 52 f9 ..d.8.....R. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0020: 77 bf 31 53 21 df a5 a0-4e 54 e1 52 9f a1 79 0c w.lS!...NT.R..y. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1753): +-----+
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_kernel.c(1787): OpenSSL: Loop: SSLv3 read finished A
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_kernel.c(1787): OpenSSL: Loop: SSLv3 write session ticket A
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_kernel.c(1787): OpenSSL: Loop: SSLv3 write change cipher spec A
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_kernel.c(1787): OpenSSL: Loop: SSLv3 write finished A
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_kernel.c(1787): OpenSSL: Loop: SSLv3 flush data
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_kernel.c(1783): OpenSSL: Handshake: done

```

```

[Fri Mar 06 09:19:49 2009] [info] Connection: Client IP: 88.41.252.232, Protocol: TLSv1, Cipher: ECDHE-ECDHE-AES256-SHA
(256/256 bits)
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1775): OpenSSL: read 5/5 bytes from BIO#82c2f80 [mem: 82df200] (BIO dump
follows)
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1722): +-----+
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0000: 17 03 01 01 b0          ..... |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1753): +-----+
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1775): OpenSSL: read 432/432 bytes from BIO#82c2f80 [mem: 82df205] (BIO dump
follows)
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1722): +-----+
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0000: ac 0c fb b0 a1 c5 38 9e-cf d8 1e 76 bf ae a3 f7 .....8....v... |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0010: 3a ce 91 89 dd 3d 55 57-fa 83 2b 64 2c 65 b8 d8 .....=UW..+d.e.. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0020: 56 5e 1b f5 f1 9a 6c c4-d8 ae 5b 1f 31 b3 e3 0b V^...!...[.1... |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0030: 8d 5c 09 44 ed 0e 21 26-b9 44 da 0a 46 ca 6a 9b .\D.!&.D.F.j. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0040: 29 d2 b4 ae bb cd 8f 37-a3 63 11 8a 30 5b 3e 5d ).....7.c.0[> |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0050: 06 73 10 ff 9e c5 56 73-89 69 ea 56 54 e0 01 2e ...s...Vs.i.VT... |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0060: ae 0e 93 6a 40 0d d3 05-f9 ce 0c 4f 24 fe 17 5a ...j@.....O$.Z |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0070: 05 86 0d c6 19 23 92 bb-1f 77 7b 82 95 8d 1b af ....#...w{..... |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0080: d6 56 3e 76 84 d3 af 7c-4f 38 48 9f 51 d4 d3 6d .V>v...|O8H.Q.m |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0090: 4e 8e 1f c2 74 11 c9 6a-21 87 ff 0f 6f ca 2a 14 N...t.j!...o.*. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 00a0: c5 d1 b4 2b 35 e8 f5 35-1a 57 5c 27 a9 a3 cd 16 ...+5..5.W\.... |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 00b0: 73 bf d0 99 59 64 a5 ec-7b f5 18 58 ba 29 d2 d7 s...Yd..{..X).. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 00c0: 6e a0 39 d7 be a6 ac f8-79 c3 65 05 39 e3 62 f5 n9.....y.e.9.b. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 00d0: 7c 96 78 c7 2e 3e 57 fb-13 d2 41 a8 b6 69 9b dd |x.>W...A..i. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 00e0: 59 11 02 b0 4b 5f ab 73-6e bd 0c 1e 1d 50 0a 8d Y...K..sn...P.. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 00f0: a2 1d b7 e5 72 0c 0c ce-e8 e4 93 98 d4 b6 cf 49 ....r.....I |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0100: 26 99 40 fc 7b 95 de c4-0a b6 a2 2b fc d6 40 ce &.@.{.....+@. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0110: 4e 15 da 44 15 0f be 46-db 29 ef ff 6e a6 6f 96 N..D...F)..n.o. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0120: c3 06 f3 fc 26 15 a0 71-59 67 e2 58 1f eb 5e 0a ....&.qYg.X.^. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0130: 13 dc 8a 81 3b 8e 07 22-a2 5c d3 ba 31 7c c1 7c .....".\..1. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0140: 4e d7 5e 2e 7b a5 3c 71-b0 30 e0 5e 2d 38 73 c2 N..^.{<q.0.^8s. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0150: 2c 2c 9e bd 12 23 04 5e-08 a5 a4 a2 60 39 b2 2e ....#.^....`9. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0160: ea 0c 5e aa c7 14 4b 6e-93 c2 5f e6 2b 98 2b f4 .....Kn...+..+.. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0170: a5 15 65 4f 48 e5 94 0e-cc bd 9e 00 6d 4a 10 e9 ..eOH.....mJ.. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0180: f7 dc 04 2b ea 58 9a 66-f5 61 70 8f 4d f6 2a 23 ...+X.f.ap.M.*# |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0190: 2f 2f 6d 65 e4 de e8 a1-63 c9 c2 2d b2 e8 92 ed //me...c..... |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 01a0: 75 94 ad 15 ef 2d 96 e5-65 ee 81 cf 1b 46 00 4e u....-e.....F.N |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1753): +-----+
[Fri Mar 06 09:19:49 2009] [info] Initial (No.1) HTTPS request received for child 1 (server massimov.it:443)
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1775): OpenSSL: read 5/5 bytes from BIO#82c2f80 [mem: 82df200] (BIO dump
follows)
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1722): +-----+
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0000: 17 03 01 01 a0          ..... |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1753): +-----+
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1775): OpenSSL: read 416/416 bytes from BIO#82c2f80 [mem: 82df205] (BIO dump
follows)
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1722): +-----+
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0000: 1e 83 0a 3d c3 00 06 43-1c b7 dd 0d 66 7f a6 3d ...=...C...f..= |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): | 0010: 47 41 c2 28 d2 f7 ed 34-6b d5 b0 f6 21 04 0e f8 GA(...4k...!... |

```

```

[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): |0020: 7c b7 b0 1a 03 ec 04 b1-06 64 9a 6b f4 65 05 1b |.....d.k.e. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): |0030: e3 02 3f 95 b8 e4 75 f4-c6 8d e8 56 b4 e0 68 60 ..?..u...V..h` |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): |0040: 7b ab 67 34 17 79 33 47-a4 36 bd fb 87 14 17 2e {.g4.y3G.6..... |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): |0050: 9a 59 de 0b 33 66 7f 4e-7c c5 6e 9f f8 16 22 3a .Y..3f.N|.n..": |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): |0060: 90 70 bb 35 45 59 f1 98-fa 10 7a 60 b6 8c 2b 43 .p.5EY....z'..+C |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): |0070: 0f 8d 07 25 41 4e e2 fc-d9 db 08 2c 22 07 8b d3 ...%AN....."..." |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): |0080: 7f 69 51 c9 5c 30 c9 16-2d 32 ac 70 fe 43 d3 cb .iQ.\(0..-2.p.C.. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): |0090: 6e 5e 44 4c bf 1e 39 04-7f 44 e5 68 e5 93 e8 61 n^DL..9..D.h...a |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): |00a0: e4 05 eb 70 d1 8c 81 65-59 f6 cf 95 19 a2 04 df ...p...eY..... |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): |00b0: d2 84 39 29 e5 e4 03 fe-56 61 f5 26 97 83 7e d3 ..9)....Va.&..~. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): |00c0: 6e 62 bd 24 5c 5d 18 19-bc 5a 74 4d c0 72 6b ed nb.$\|...ZtM.rk. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): |00d0: 1a ea 1e 07 d6 27 05 51-e9 7b d4 52 1f 82 d7 90 .....Q.{.R.... |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): |00e0: 75 d8 41 e2 35 ee 76 0f-7e d1 67 6d c1 df 5b fc u.A.5.v.-.gm..[. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): |00f0: 86 9e fd f3 39 e1 99 36-12 c8 23 1f c0 7e 98 53 ....9..6.#..~.S |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): |0100: da 6b 4b 67 48 f5 71 8b-ad 1f b7 ea c3 99 c5 50 .kKgH.q.....P |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): |0110: be c8 7e b6 50 a8 7f a8-d7 80 14 fc 84 8a 6a df ..~.P.....j. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): |0120: 9d a8 17 96 c4 cf 22 d5-ec 18 49 03 b3 9d e4 68 .....L...h |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): |0130: a9 9c 44 57 1e 06 75 3c-a0 e3 cf 33 14 07 75 a3 ..DW..u<...3..u. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): |0140: c0 bc e2 6b 0b 47 29 6d-9c 57 7e 16 18 2d 1d 40 ...k.G)m.W~.-.@ |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): |0150: 26 ea 95 68 77 14 84 6b-15 08 17 f4 df 4b bb bc &..hw..k.....K.. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): |0160: 04 34 cf d2 45 ec 45 f1-54 a5 e9 91 30 89 d2 ee .4..E.E.T...0... |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): |0170: 63 b9 81 54 81 61 62 a1-0b e0 d8 7b d8 f4 8d 2e c..T.ab....{.... |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): |0180: cd f8 a8 17 4e 9f c8 c9-f9 02 a5 9a bb 45 2a b6 ....N.....E*. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1747): |0190: 50 c9 71 ea 9d d0 83 de-a7 31 20 fa 3a 62 03 a9 P.q.....1 .:b. |
[Fri Mar 06 09:19:49 2009] [debug] ssl_engine_io.c(1753): +-----+
[Fri Mar 06 09:19:49 2009] [info] Subsequent (No.2) HTTPS request received for child 1 (server massimov.it:443)
[Fri Mar 06 09:19:49 2009] [error] [client 88.41.252.232] File does not exist: /usr/local/apache2.2.2/htdocs/favicon.ico
[Fri Mar 06 09:19:54 2009] [debug] ssl_engine_io.c(1786): OpenSSL: I/O error, 5 bytes expected to read on BIO#82c2f80 [mem: 82df200]
[Fri Mar 06 09:19:54 2009] [info] [client 88.41.252.232] (70007)The timeout specified has expired: SSL input filter read failed.
[Fri Mar 06 09:19:54 2009] [debug] ssl_engine_kernel.c(1797): OpenSSL: Write: SSL negotiation finished successfully
[Fri Mar 06 09:19:54 2009] [info] [client 88.41.252.232] Connection closed to child 1 with standard shutdown (server massimov.it:443)
[Fri Mar 06 09:24:28 2009] [info] [client 192.168.1.200] Connection to child 0 established (server massimov.it:443)
[Fri Mar 06 09:24:28 2009] [info] Seeding PRNG with 136 bytes of entropy
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_kernel.c(1779): OpenSSL: Handshake: start
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_kernel.c(1787): OpenSSL: Loop: before/accept initialization
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1775): OpenSSL: read 11/11 bytes from BIO#82c2f80 [mem: 82dd1f8] (BIO dump follows)
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1722): +-----+
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1747): |0000: 80 2c 01 03 01 00 03 ..... |
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1751): |0011 - <SPACES/NULS>
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1753): +-----+
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1775): OpenSSL: read 35/35 bytes from BIO#82c2f80 [mem: 82dd203] (BIO dump follows)
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1722): +-----+
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1747): |0000: 00 c0 0a 9b 88 56 b5 84-0a eb 93 2e 9a a3 58 35 ....V.....X5 |
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1747): |0010: 18 96 49 3a 0b 6f cd 1f-f8 84 25 68 90 47 19 42 ..i.o....%h.G.B |
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1747): |0020: 4a db d6 J. |
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1753): +-----+
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_kernel.c(1787): OpenSSL: Loop: SSLv3 read client hello A

```

```

[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_kernel.c(1787): OpenSSL: Loop: SSLv3 write server hello A
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_kernel.c(1787): OpenSSL: Loop: SSLv3 write certificate A
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_kernel.c(1169): [client 192.168.1.200] handing out temporary 256 bit ECC key
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_kernel.c(1787): OpenSSL: Loop: SSLv3 write key exchange A
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_kernel.c(1787): OpenSSL: Loop: SSLv3 write server done A
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_kernel.c(1787): OpenSSL: Loop: SSLv3 flush data
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1775): OpenSSL: read 5/5 bytes from BIO#82c2f80 [mem: 82dd1f8] (BIO dump follows)
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1722): +-----+
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1747): | 0000: 16 03 01 00 46                ....F      |
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1753): +-----+
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1775): OpenSSL: read 70/70 bytes from BIO#82c2f80 [mem: 82dd1fd] (BIO dump follows)
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1722): +-----+
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1747): | 0000: 10 00 00 42 41 04 71 d1-62 be f4 6a e8 fa 36 58 ...BA.q.b..j..6X |
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1747): | 0010: a7 a0 af de 44 12 50 d8-f3 fc 85 5b f1 5d 95 ab ....D.P....|. |
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1747): | 0020: a6 5e 83 b0 b2 98 a4 7c-21 e9 3c 00 d8 6c 4e 1f .^....|!<.IN. |
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1747): | 0030: d6 b6 a9 7e f6 52 be 8d-4c b2 9d 41 ec d7 c6 ad ...~.R.L.A.... |
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1747): | 0040: 9b 1b e9 69 dc 91                ...i..      |
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1753): +-----+
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_kernel.c(1787): OpenSSL: Loop: SSLv3 read client key exchange A
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1775): OpenSSL: read 5/5 bytes from BIO#82c2f80 [mem: 82dd1f8] (BIO dump follows)
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1722): +-----+
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1747): | 0000: 14 03 01 00 01                ....      |
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1753): +-----+
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1775): OpenSSL: read 1/1 bytes from BIO#82c2f80 [mem: 82dd1fd] (BIO dump follows)
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1722): +-----+
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1747): | 0000: 01                .          |
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1753): +-----+
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1775): OpenSSL: read 5/5 bytes from BIO#82c2f80 [mem: 82dd1f8] (BIO dump follows)
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1722): +-----+
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1747): | 0000: 16 03 01 00 30                ....0      |
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1753): +-----+
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1775): OpenSSL: read 48/48 bytes from BIO#82c2f80 [mem: 82dd1fd] (BIO dump follows)
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1722): +-----+
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1747): | 0000: 22 57 d4 30 80 be 2b 00-2e 7b d6 43 53 4d dd d0 "W.0.+..{.CSM.` |
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1747): | 0010: 95 9f 74 c6 f6 2e c6 d7-c0 cf eb 3c 94 5a 0c da ..t.....<.Z.. |
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1747): | 0020: 55 4b 32 a0 35 a0 9d 0a-60 df b2 8b 38 95 7f 3d UK2.5...^...8..= |
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_io.c(1753): +-----+
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_kernel.c(1787): OpenSSL: Loop: SSLv3 read finished A
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_kernel.c(1787): OpenSSL: Loop: SSLv3 write change cipher spec A
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_kernel.c(1787): OpenSSL: Loop: SSLv3 write finished A
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_kernel.c(1787): OpenSSL: Loop: SSLv3 flush data
[Fri Mar 06 09:24:28 2009] [debug] ssl_scache_shmcb.c(670): inside shmcb_store_session
[Fri Mar 06 09:24:28 2009] [debug] ssl_scache_shmcb.c(676): session_id[0]=138, masked index=10

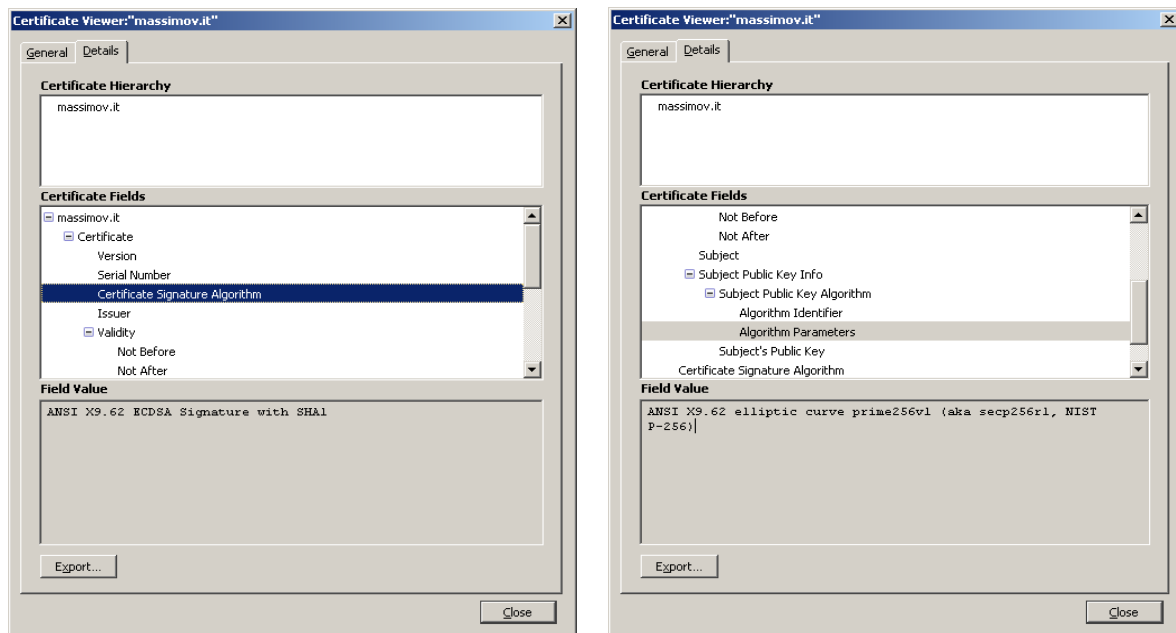
```

```

[Fri Mar 06 09:24:28 2009] [debug] ssl_scache_shmcb.c(1059): entering shmcb_insert_encoded_session, *queue->pos_count = 3
[Fri Mar 06 09:24:28 2009] [debug] ssl_scache_shmcb.c(983): entering shmcb_expire_division
[Fri Mar 06 09:24:28 2009] [debug] ssl_scache_shmcb.c(1007): will be expiring 3 sessions
[Fri Mar 06 09:24:28 2009] [debug] ssl_scache_shmcb.c(1032): we now have 0 sessions
[Fri Mar 06 09:24:28 2009] [debug] ssl_scache_shmcb.c(1115): we have 14386 bytes and 133 indexes free - enough
[Fri Mar 06 09:24:28 2009] [debug] ssl_scache_shmcb.c(1144): storing in index 0, at offset 0
[Fri Mar 06 09:24:28 2009] [debug] ssl_scache_shmcb.c(1159): session_id[0]=138, idx->s_id2=157
[Fri Mar 06 09:24:28 2009] [debug] ssl_scache_shmcb.c(1170): leaving now with 148 bytes in the cache and 1 indexes
[Fri Mar 06 09:24:28 2009] [debug] ssl_scache_shmcb.c(1174): leaving shmcb_insert_encoded_session
[Fri Mar 06 09:24:28 2009] [debug] ssl_scache_shmcb.c(704): leaving shmcb_store successfully
[Fri Mar 06 09:24:28 2009] [debug] ssl_scache_shmcb.c(418): shmcb_store successful
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_kernel.c(1625): Inter-Process Session Cache: request=SET status=OK
id=8A9D2989B96F5CA0C989420003F695A8EEAF6A803C135C4A122246E70A8CA3D7 timeout=300s (session caching)
[Fri Mar 06 09:24:28 2009] [debug] ssl_engine_kernel.c(1783): OpenSSL: Handshake: done
[Fri Mar 06 09:24:28 2009] [info] Connection: Client IP: 192.168.1.200, Protocol: TLSv1, Cipher: ECDHE-ECDSA-
AES256-SHA (256/256 bits)
[Fri Mar 06 09:24:33 2009] [debug] ssl_engine_io.c(1786): OpenSSL: I/O error, 5 bytes expected to read on BIO#82c2f80
[mem: 82dd1f8]
[Fri Mar 06 09:24:33 2009] [info] [client 192.168.1.200] (70014)End of file found: SSL input filter read failed.
[Fri Mar 06 09:24:33 2009] [debug] ssl_engine_kernel.c(1797): OpenSSL: Write: SSL negotiation finished successfully
[Fri Mar 06 09:24:33 2009] [info] [client 192.168.1.200] Connection closed to child 0 with standard shutdown (server
massimov.it:443)

```

Ulteriore verifica è stata eseguita con il browser Firefox versione 3.0.6 (IE Explorer 7.xx non supporta la cifratura ellittica), si riporta qui di seguito il certificato visualizzato in modo grafico dove vengono evidenziati l'algoritmo e i parametri di cifratura.



### 2.2.3 Standard di misurazione

I test sono stati effettuati con l'aiuto dell'utility *s\_time* inclusa nella suite openssl.

Tale applicativo effettua una connessione, specificando la crittografia, usa SSL Handshaking per determinare il numero di connessioni per secondo, usando sia le nuove connessioni sia quelle già attive, opzione quest'ultima configurabile richiamando una pagina web di test di 0 Kb, 10Kb, 30Kb.

### 2.2.4 Misure dei livelli di performance degli algoritmi

Sempre utilizzando la suite Openssl con il comando *openssl ecparam -list\_curves* si ottiene la lista delle tipologia di curve da associare, sono stati scelti gli algoritmi consigliati dal NIST :

RSA 1024	
ECC 163	NIST/SECG curve over a 163 bit binary field
RSA3072	
ECC283	NIST/SECG curve over a 283 bit binary field
RSA7680	
ECC409	NIST/SECG curve over a 409 bit binary field
RSA15360	
ECC571	NIST/SECG curve over a 571 bit binary field

Con l'utilizzo di una delle opzioni della suite openssl e precisamente con *openssl s\_time -connect* è stato interrogato il web server dopo aver generato e installato i certificati relativi, con i seguenti parametri :

- Senza alcun trasferimento di pagina, quindi solo handshake  
`openssl s_time -connect 192.168.1.130:443 -cipher RSA -new -ssl3`
- Con trasferimento di una pagina html da 10 k  
`openssl s_time -connect 192.168.1.130:443 -cipher RSA -new -ssl3 -www /test10k.html`
- Con trasferimento di una pagina html da 30 k  
`openssl s_time -connect 192.168.1.130:443 -cipher RSA -new -ssl3 -www /test30k.html`

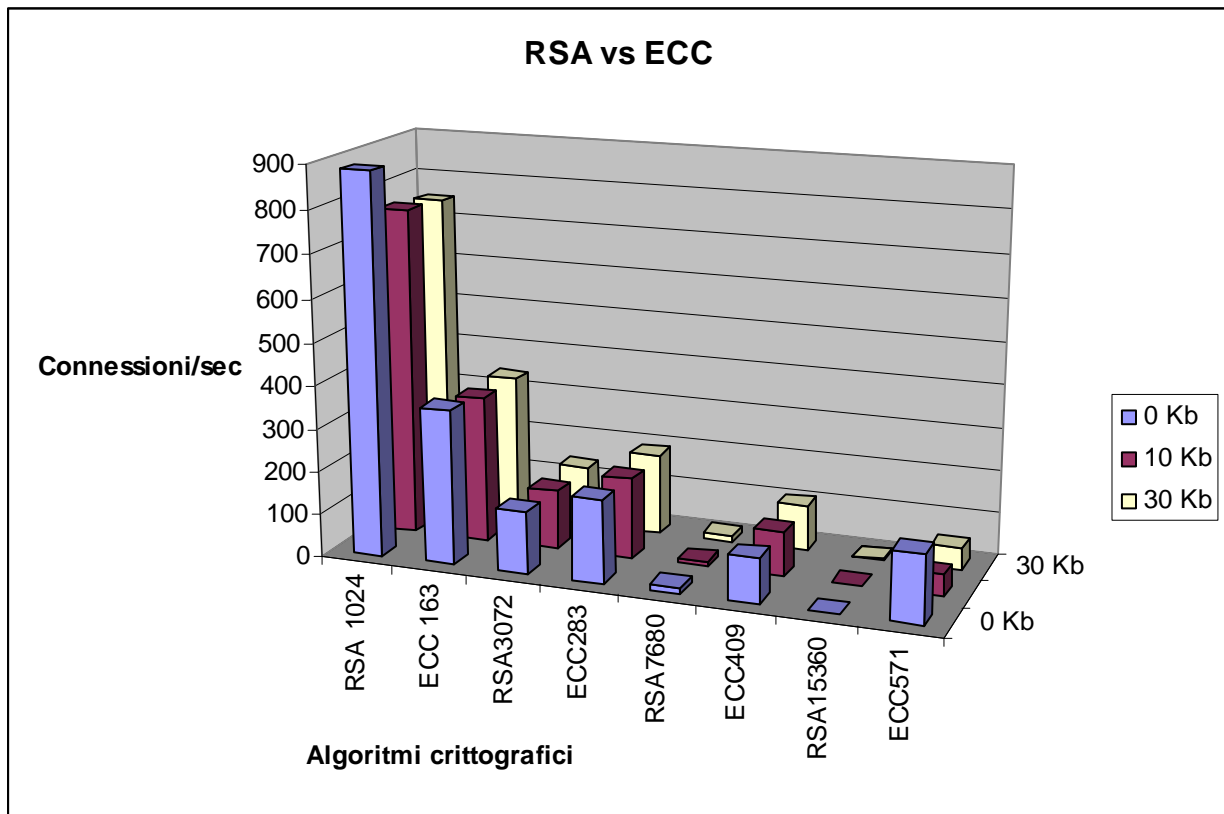
I risultati di queste interrogazioni sono riportati nella seguente tabella (i risultati sono stati moltiplicati di un fattore 10 in modo di avere una efficace visualizzazione grafica) :

## Risultati dei test

### Dimensioni delle pagine

	0 Kb	10 Kb	30 Kb
RSA 1024	885	766	753
ECC 163	361	340	340
RSA3072	147	139	139
ECC283	195	187	190
RSA7680	14	14	14
ECC409	105	105	107
RSA15360	2	2	2
ECC571	159	50	51

E qui di seguito i medesimi dati sono proposti anche in forma grafica:



Si nota immediatamente che al crescere della chiave l'algoritmo RSA pecca di efficienza rispetto a ECC, questo si traduce nel fatto che a parità di sicurezza il server web è in grado di processare una maggiore richiesta di carico e/o rispondere più velocemente.

## Conclusioni

Lo studio della crittografia che utilizza curve ellittiche - in principio relegato in ambito accademico - ha successivamente suscitato l'interesse di enti internazionali quali ANSI, ISO, così come di società operanti nel campo informatico; a titolo di esempio si considerino:

- la società Elliptic Semiconductor <http://www.ellipticsemi.com/>, specializzata nella costruzione e implementazione di soluzioni basate sulla crittografia ellittica.
- Il contratto stipulato nel 2003 tra la società Certicom e la NSA, National Security Agency, appunto per la vendita di sistemi basati sulle curve ellittiche.

Inoltre l'ECC, come effetto della sua crescente popolarità, negli anni è stata sottoposta a numerosi tentativi di attacco, che hanno finito per confermarne forza e credibilità in materia di sicurezza.

Indubbiamente proprio la crescente richiesta di sicurezza, velocità, mobilità da parte del mercato dei prodotti informatici determinerà in breve tempo un'ulteriore diffusione di tale tecnologia.

Le novità più salienti in materia saranno esposte nella ECC conference 2009 che si svolgerà in Calgary, Canada, nell'ultima settimana del prossimo aprile.

In tale contesto ad esempio saranno discussi problemi quali la pairing-based cryptography e il problema computazionale dell'ECDLP.

La novità più attesa è lo sviluppo delle "Edwards Elliptic Curves", un altro tipo di curve ellittiche non nella forma di Weierstrass, che promettono una maggiore efficienza computazionale rispetto al metodo usuale, per maggiori informazioni si veda <http://cr.ypt.to/newelliptic/newelliptic.html>.

Appendici.

### A. Caratteristica di un campo.

La *caratteristica* di un campo  $F$  è definita come il più piccolo numero naturale  $n$  tale che l'elemento

$$\underbrace{1+1+1+\dots+1}_n$$

sia uguale a 0. Se questo non esiste, cioè  $1+1+1+\dots+1$  è sempre diverso da zero, la caratteristica è zero per definizione.

La caratteristica di un campo  $F$  viene indicata con il simbolo  $\text{char}(F)$ .

### B. Discriminante

Il discriminante di un polinomio è il prodotto delle differenze dei quadrati delle differenze delle radici. Consideriamo un polinomio di grado  $n$

$$a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

il suo discriminante è definito come:

$$\Delta = a_n^{2n-2} \Delta \prod_{i < j}^n (x_i - x_j)^2$$

dove le varie  $x_k$  sono le radici del polinomio.

Quindi imporre che sia  $\Delta \neq 0$  significa imporre che il polinomio non abbia radici multiple.

In particolare analizziamo il polinomio:

$$x^3 + a x + b$$

indicando con  $x_1, x_2, x_3$ , le radici di polinomio di cui sopra si può dimostrare che il discriminante

$$\Delta = ((x_1 - x_2)(x_1 - x_3)(x_2 - x_3))^2 = -(4a^3 + 27b^2)$$

### C. Metodo Pollard $\rho$

Questo algoritmo venne introdotto nel 1978. In suo funzionamento si basa sul cosiddetto “*Paradosso del Compleanno*” che può essere così riassunto: il numero minimo di persone che devo considerare affinché due di esse compiano gli anni lo stesso giorno, con probabilità del 50% è 24 (valore approssimabile a  $\sqrt{\pi 365/2}$ ). Più in generale, ogniqualvolta si selezionano a caso degli

elementi da un insieme di dimensione  $n$ , è sufficiente selezionare  $O(\sqrt{n})$  per avere una probabilità del 50% di selezionare lo stesso elemento (di avere cioè *collisione*).

Analisi dell'algoritmo. Si abbia sempre da calcolare il logaritmo  $k = \log_p Q$ , con  $n$  ordine di  $P$ .

L'algoritmo considera una funzione iterativa  $f : \langle P \rangle \rightarrow \langle P \rangle$  dal comportamento pseudo-casuale.

Inizia con un punto casuale  $P_0$  e calcola le iterazioni Logaritmo Discreto su Curve Ellittiche.

$P_{i+1} = f(P_i)$ . Poiché  $\langle P \rangle$  è un insieme finito, ad un certo punto ci sarà collisione, ossia si avranno due indici  $t < j$  tali che  $P_t = P_j$ . Per come è costruita la sequenza, inoltre, si ha per ogni  $P_{t+l} = P_{j+l}$  per ogni  $l > 0$ . Ciò significa che la sequenza  $P_i$ , dopo la collisione, diventa periodica di periodo  $s = j - t$ , ossia

$$P_i = P_{i-s} \quad \text{per tutti gli } i \geq t + s$$

La rappresentazione grafica di tale processo ricorda la lettera greca  $\rho$  (da cui il nome di tale algoritmo). Il valore  $t$  è chiamato lunghezza della coda, mentre  $s$  è detto lunghezza del ciclo. Se la funzione  $f$  è sufficientemente casuale, si avrà collisione in un tempo  $O(\sqrt{n})$ .

Un implementazione banale memorizzerebbe tutti i punti  $P_i$  fino ad ottenere collisione, ma questo richiederebbe un'occupazione  $O(\sqrt{n})$ . L'algoritmo di Pollard invece utilizza il Metodo di cycle-finding di Floyd: si evita lo spreco di memoria, al prezzo di qualche calcolo in più. L'idea di Floyd consiste nel calcolare le coppie  $(P_i, P_{2i})$  con  $i = 1, 2, \dots$  fino a quando si ottiene il matching  $P_i = P_{2i}$ . Dopo aver calcolato una nuova coppia la precedente viene eliminata: in tal modo l'utilizzo di memoria è minimo. Il numero  $i$  di coppie da calcolare prima di avere *matching* è tale che  $t \leq i \leq t+s$ . In tale intervallo infatti esiste sicuramente un  $i$  che soddisfa l'equazione sopra esposta ed è multiplo di  $s$ , pertanto  $i$  e  $2i$  differiscono per un multiplo di  $s$  e quindi  $P_i = P_{2i}$ . Di conseguenza il numero di passi per trovare un *matching* è al più un multiplo di  $\sqrt{n}$ .

Vediamo ora come scegliere la funzione iterativa  $f$ . Essa, oltre a generare una sequenza pseudo-casuale, deve fornire informazioni utili per trovare  $k$  in caso di matching. Un modo per ottenere questo consiste nel partizionare  $\langle P \rangle$  in  $L$  sottoinsiemi disgiunti dalla dimensione confrontabile  $\{S_1, S_2, \dots, S_L\}$  (valori tipici di  $L$  sono 16 e 32). Si scelgono  $2L$  interi casuali  $a_i + b_i \pmod n$  e si definisce

$$M_i = a_i P + b_i Q$$

Infine si definisce la funzione  $f$ :

$$f(G) = G + M_i \text{ se } G_i \text{ appartiene a } S_i$$

Intuitivamente si può pensare ad  $f$  come ad una passeggiata casuale (*random walk*) in  $\langle P \rangle$  con passi rappresentati dai valori  $M_i$ . Per iniziare la random walk si scelgono due interi casuali  $a_0$  e  $b_0$  e si definisce il punto iniziale  $P_0 = a_0 P + b_0 Q$ .

Per come è costruita la sequenza, quindi, il generico punto  $P_i$  sarà sempre esprimibile nella forma  $P_i = u_i P + v_i Q$ . La presenza di un matching  $P_i = P_j$  significa  $u_i P + v_i Q = u_j P + v_j Q$  quindi  $(u_i - u_j)P = (v_j - v_i)Q = (v_j - v_i)kP$ . Pertanto si ha  $(u_i - u_j) \equiv (v_j - v_i)k \pmod{n}$  e il valore  $k$  è presto calcolato:

$$k = (u_i - u_j) (v_j - v_i)^{-1} \pmod{n}$$

L'algoritmo Pollard  $\rho$  ha un running time atteso di circa  $\equiv \sqrt{\pi \cdot n/2}$  steps (dove uno step corrisponde ad una somma EC) e ad oggi è considerato il metodo più veloce per risolvere l'ECDLP. Nel 1999 Van Oorschot e Wiener hanno proposto una versione distribuita su  $r$  processori, ottenendo così un *running time* pari a  $(\sqrt{\pi \cdot n/2})/r$  [8]. L'esecuzione in parallelo, quindi, ha aumentato la velocità di un fattore  $r$ , ma la complessità è rimasta di tipo fully-exponential, infatti la complessità è legata al numero di bits  $O(\log n)$  che esprimono l'input  $n$ , pertanto  $\sqrt{n} = n^{1/2} = 2^{(\log n)/2}$  è esponenziale in  $\log n$ .

#### D. Metodo Pohlig-Hellman

L'algoritmo Pohlig-Hellman riduce il calcolo di  $k = \log_p Q$  nel calcolo del logaritmo discreto nei sottogruppi di  $\langle P \rangle$  di ordine pari ai fattori primi di  $n$  dove  $n$  è l'ordine di  $P$ . Sia quindi  $n = p_1^{e_1} p_2^{e_2} \dots p_r^{e_r}$  la fattorizzazione prima di  $n$ . La strategia seguita dall'algoritmo consiste nel calcolare  $k_i = k \pmod{p_i^{e_i}}$  per ogni  $i$  appartenente  $[1, r]$  e risolvere il sistema di congruenze

$$\left\{ \begin{array}{l} k \equiv k_1 \pmod{p_1^{e_1}} \\ k \equiv k_2 \pmod{p_2^{e_2}} \\ \dots \\ k \equiv k_r \pmod{p_r^{e_r}} \end{array} \right.$$

Il teorema del Resto Cinese garantisce l'unicità della soluzione  $k \pmod{n}$ .

## **Ringraziamenti**

Grazie a Silvia, moglie paziente, che mi ha sempre incoraggiato.

Grazie alla Prof.ssa Laura Citrini per il continuo supporto alla realizzazione di questa tesi.

Grazie a Federico, Pierluca e Marco per aver condiviso con me questa meravigliosa avventura.

Grazie a Sabrina per il suo instancabile e insostituibile aiuto per ben tre anni.

Infine grazie ad Alessia per il supporto nei momenti matematicamente difficili.

## BIBLIOGRAFIA

- [1] Certicom Corp. white paper, "Remarks on the security of the elliptic curve cryptosystem", September 1997. Available from <http://www.certicom.com>
  
- [2] “ Crittosistemi basati su Curve Ellittiche”, Prof. Alfredo De Santis ,“ Appunti da: “Sistemi di elaborazione dell'informazione (Sicurezza su Reti)”. Riferimento da <http://www.dia.unisa.it/~ads/corso-security/www/CORSO-0001/ECC/index.htm>
  
- [3] Mitch Tulloch “Enciclopedia della sicurezza” Mondadori Informatica 2004
  
- [4] William Stallng “Crittografia e sicurezza della reti” McGraw Hill 2007
  
- [5] A. Menezes, T. Okamoto e S. Vanstone, "*Reducing elliptic curve logarithms to logarithms in a finite field*", IEEE Transactions on information theory, 39 (1993) 1639 -1646
  
- [6] L. C. Washington, “Elliptic Curves, Number Theory and Cryptography”, CRC Press, 2003.
  
- [7] Elliptic Curves - Number Theory and Cryptography, 2nd Edition-(Lawrence C. Washington) Chapman & Hall-CRC 2008
  
- [8] A. Menezes, D. Hankerson, S. Vanstone, “Guide to Elliptic Curve Cryptography”, Springer, 2004