Problemi di razionalità di tori algebrici e applicazioni alla crittografia

Alessio Palmero Aprosio

Indice

Introduzione

Prima dell'avvento di internet, la scienza che si occupava di inviare messaggi al riparo da occhi indiscreti era segregata a meri scopi militari, anzi molto spesso gli stessi scopritori di nuovi sistemi sicuri per comunicare venivano obbligati a mantenere il segreto, con l'inevitabile conseguenza che le scoperte non venivano loro riconosciute. Simile destino è toccato ad Alan Turing, ideatore dell'informatica moderna, che durante la Seconda Guerra Mondiale è riuscito a decifrare i messaggi delle forze tedesche, ma del quale non si è saputo nulla fino agli anni Settanta.

Al giorno d'oggi, però, gli interessi dietro alla protezione delle comunicazioni sono usciti in modo prepotente dall'ambito militare, grazie alla rapidissima diffusione del World Wide Web, meglio conosciuto come internet. Attraverso la rete possiamo comunicare, trovare informazioni, conoscere nuovi amici e anche gestire conti bancari o fare acquisti con la nostra carta di credito. In questi ultimi aspetti entra in gioco quella scienza che si occupa di nascondere le informazioni private agli occhi indiscreti: la crittografia.

La vera rivoluzione in questo ambito si è avuta nel 1976, quando Diffie e Hellman inventarono la crittografia a chiave pubblica. In pratica i due ricercatori hanno risolto l'annoso problema dello scambio di chiavi: prima di essi, il mittente e il destinatario legittimo del messaggio dovevano accordarsi preventivamente sul procedimento utilizzato per cifrare e decifrare. Con la chiave pubblica, invece, il destinatario comunica al mittente la sua chiave pubblica, che il mittente userà per celare il messaggio, dopo di che nemmeno lui potrà riottenerne il contenuto, perchè per decifrare il testo dovrà essere usata una chiave, differente dalla prima, custodita gelosamente dal destinatario.

Come è possibile ottenere questo risultato nella pratica? Basta trovare un problema matematico appartenente alla categoria NP-completo, ovvero un'operazione facile da compiere in un senso, che risulta però molto difficile nell'altro. Ad esempio dati due numeri primi (diciamo 37 e 97) è molto facile trovare il loro prodotto. Invece dato il loro prodotto (3589) non è per niente facile risalire ai numeri di partenza, nonostante il teorema fondamentale dell'aritmetica ci assicuri che sono unici. In questo caso, per chiarire l'idea, la chiave pubblica è 3589 e la chiave privata è uno qualsiasi dei suoi fattori primi (37 o 97), avendo uno dei quali si riesce a ricavare il secondo.

Uno dei problemi di questo tipo più diffusi e utilizzati per i cifrari moderni è il logaritmo discreto. Dati tre numeri p, q, r, si tratta di trovare un intero a tale che

$$p^a \equiv r \bmod q.$$

Col passare degli anni, tuttavia, ci si è resi conto che il logaritmo discreto sui campi di ricerca semplici come quello dei numeri naturali invecchiava troppo velocemente. Da una parte i calcolatori diventavano sempre più veloci nel tentare una ricerca della soluzione con metodi esaustivi. Dall'altra i ricercatori stessi affinavano e miglioravano algoritmi affinché non fosse necessario analizzare effettivamente tutti i casi possibili, ma che anzi permettessero di raggiungere la soluzione più rapidamente. Per questo motivo si sono esplorate le più remote aree dell'algebra per trovare campi e gruppi finiti *ad hoc*, che rendessero più difficile questa ricerca.

Un primo passo è stato fatto nel 1985 quando per la prima volta si è parlato di curve ellittiche a fini crittografici: su di esse il logaritmo discreto risulta molto più difficile.

Nel 1993, invece, per la prima volta si è presa in considerazione l'idea di "comprimere" i dati inviati mantenendo inalterato il grado di sicurezza. Nascono così i cifrari di Lucas, il più conosciuto XTR e, finalmente, i cifrari basati sui tori algebrici.

Tutti i cifrari appena citati, come abbiamo detto, basano la loro sicurezza sul problema del logaritmo discreto: esistono molti altri problemi che potrebbero adattarsi a costruire dei cifrari robusti. Concentrarsi molto sul loro studio e su una loro eventuale applicazione pratica ha spinto i ricercatori a esplorare anche proprietà che altrimenti sarebbero rimaste nell'ombra (ad esempio, per le curve ellittiche, la supersingolarità e altre proprietà di interesse crittografico).

L'argomento racchiude inoltre al suo interno molti più ambiti di ricerca di quanto si possa pensare.

(i) Innanzi tutto il punto di vista algebrico-geometrico. Vengono utilizzati strumenti come la restrizione di Weil degli scalari, la teoria di Galois, le mappe birazionali tra varietà algebriche e tutta l'infrastruttura necessaria per trattare con i tori algebrici.

(ii) In secondo luogo il punto di vista crittografico. Vengono ideati nuovi algoritmi e, fattore essenziale, vengono trasformate in realtà e calcolate molte relazioni rimaste per anni (se non secoli) sotto la polvere della teoria: se esiste, perché darne una rappresentazione esplicita?

(iii) Infine non è di scarsa importanza l'aspetto computazionale. Gli studi recenti sulla crittografia hanno portato alla luce problemi di calcolo un tempo impensabili, come ad esempio trovare un numero primo di 20 cifre, scomporre in fattori primi numeri di 80 cifre o, semplicemente, elevare un elemento di un gruppo alla potenza n-esima, con n di svariate decine di cifre.

Il nostro lavoro, dunque, si pone al crocevia di questi tre studi, ed è proprio il fatto di unirli insieme che rende originali anche alcuni aspetti altrimenti già noti, per non parlare dei diversi punti di vista dal quale alcuni oggetti matematici possono essere visti e, di vitale importanza, dei nuovi problemi aperti portati alla comunità scientifica.

Concludiamo dicendo che, nonostante abbiamo cercato nel nostro lavoro di dare una panoramica il più possibile completa di un campo inevitabilmente in veloce espansione, esso non poteva essere trattato esaurientemente. Ecco la ragione per l'assenza di alcuni cifrari pur vicini come metodo a quelli trattati (curve cubiche, curve iperellittiche, varietà jacobiane arbitrarie, accoppiamenti di Weil), di alcuni metodi spinti di programmazione dedicata (rappresentazione di Montgomery, trattamento avanzato degli esponenti, metodi specifici per interi a 32 e 64 bit) nonché di considerazioni approfondite sulla complessità

degli algoritmi proposti. È stata effettuata invece una scelta adatta a rendere la trattazione quanto più possibile coesa e organica, per lo meno per quanto concerne gli argomenti effettivamente trattati.

Nel primo capitolo sono stati dati i fondamenti generali di algebra e geometria e le nozioni di base di crittografia. Si prosegue con il capitolo 2, dove si è scelto di fare una panoramica su alcuni cifrari scoperti negli ultimi anni. Il terzo e il quarto capitolo sono dedicati, rispettivamente, ai cifrari XTR e a quelli basati sui tori algebrici, fornendo le necessarie basi teoriche per la loro comprensione. L'ultimo capitolo, infine, contiene il lavoro di implementazione vero e proprio, con i sorgenti dei programmi scritti in Java, utilizzati per verificare tutti i procedimenti del capitolo 4 e per stimare le tempistiche di esecuzione nei casi reali di sicurezza a 1024 e 2048 bit.

1
Preliminari

1.1 Ragguagli di algebra

Per la nostra trattazione abbiamo bisogno di alcune nozioni di algebra elementare.

Innanzi tutto faremo largo uso dei **campi finiti**, ovvero quella categoria di campi che contengono un numero finito di elementi. Denoteremo $\mathbb{F}_q$ il campo finito con q elementi. Spesso utilizzeremo la notazione $\mathbb{F}_{p^n}$ per esaltare il fatto che i campi finiti hanno sempre un numero di elementi pari a p^n con p primo e n intero positivo. p è detta **caratteristica** del campo.

Vale il seguente teorema, utile anche per descrivere il gruppo dei punti di un toro algebrico (vedere Sezione la 4.1).

Teorema 1.1. *Il gruppo moltiplicativo di un campo finito è ciclico. Più in generale, se* G *è un sottogruppo finito del gruppo moltiplicativo di un campo qualsiasi, allora* G *è ciclico.*

Se E ed F sono campi tali che $F \subseteq E$, si dice che E è un'**estensione** di F. Useremo spesso la notazione E/F. Se consideriamo E come gruppo abeliano rispetto all'operazione di addizione, possiamo moltiplicare il "vettore" $x \in E$ per lo "scalare" $\lambda \in F$. Si verifica facilmente che gli assiomi di spazio vettoriale sono verificati, per cui spesso utilizzeremo, sulle estensioni di campo, proprietà che ci fornisce l'algebra lineare. La dimensione dello spazio vettoriale così ottenuto viene detto **grado dell'estensione** e si indica con $[E : F]$. Si parla di **estensione finita** o **infinita** a seconda che lo sia o meno il numero $[E : F]$.

Se f è un polinomio non costante a coefficienti in F e tale da non avere radici in F, possiamo sempre trovare una radice di f in un'estensione E. Dato invece un elemento $\alpha \in E$, si dice che α è **algebrico** su F se esiste un polinomio f a coefficienti in F per cui $f(\alpha) = 0$. Se questa proprietà vale per ogni $\alpha \in E$, allora si dice che l'estensione è algebrica.

Sia di nuovo f un polinomio non costante a coefficienti in F. Per quanto detto, per ogni radice del polinomio, possiamo trovare un'estensione che la contenga. La più piccola estensione E che contiene tutte le radici del polinomio è detta **campo di spezzamento** di f. In particolare, E è generata dagli elementi di F con l'aggiunta di tutte le radici di f. Il campo di spezzamento è unico. Un campo E in cui tutti i polinomi a coefficienti in E si spezzano in fattori lineari è detto **algebricamente chiuso**.

Se F è un campo qualsiasi (non necessariamente algebricamente chiuso), un'estensione algebrica di F algebricamente chiusa è detta **chiusura algebrica** di F. La chiusura algebrica di un campo F esiste sempre ed è unica a meno di isomorfismi.

Date due estensioni E ed E' del campo F, l'isomorfismo $i : E \to E'$ è detto **F-isomorfismo** se fissa il sottocampo F di E, ovvero se $i(a) = a$ per ogni elemento $a \in F$. Similmente possiamo definire F-omomorfismi, F-monomorfismi, ecc.

Un polinomio irriducibile f a coefficienti nel campo F è **separabile** se non ha radici ripetute nel campo di spezzamento. È utile la seguente proposizione.

Proposizione 1.2. *Sia dato un campo E. Se vale una delle seguenti:*

(i) *E ha caratteristica 0;*

(ii) *E è un campo finito;*

allora ogni polinomio a coefficienti in E è separabile.

Se E/F è un'estensione e $\alpha \in E$, allora α è detto **separabile** su F se α è algebrico su F e se il polinomio minimo di α su F è separabile. Se ogni elemento di E è separabile su F, allora l'estensione è detta **separabile**. Per la proposizione precedente, ogni estensione di un campo a caratteristica 0 o di un campo finito è separabile.

Un'estensione E/F è detta **normale** se ogni polinomio irriducibile a coefficienti in F che ha una radice in E si spezza in E. Vale inoltre il seguente teorema.

Teorema 1.3. *Un'estensione finita E/F è normale se e solo se esiste un polinomio f a coefficienti in f di cui E è il campo di spezzamento.*

Diamo infine le ultime due definizioni importanti per la nostra teoria.

Definizione 1.4. *Data un'estensione E/F, il gruppo dei suoi F-automorfismi è detto gruppo di Galois di E/F e si indica con $\mathrm{Gal}(E/F)$.*

Definizione 1.5. *Un'estensione E/F normale e separabile è detta **estensione di Galois**.*

1.1.1 La mappa di Frobenius

Definizione 1.6. *Sia $\mathbb{F}_q$ un campo finito. La funzione $\phi_q : \overline{\mathbb{F}}_q \to \overline{\mathbb{F}}_q$ definita da*

$$\phi_q(x) = x^q \text{ per ogni } x \in \overline{\mathbb{F}}_q$$

*è detta **mappa di Frobenius** alla potenza q.*

Proposizione 1.7. *Sia p un numero primo e $q = p^n$. Allora valgono le seguenti:*

(i) *$\overline{\mathbb{F}}_q = \overline{\mathbb{F}}_p$;*

(ii) ϕ_q *è un automorfismo di* $\overline{\mathbb{F}}_q$*, valgono cioè per ogni* $x, y \in \overline{\mathbb{F}}_q$

$$\phi_q(x+y) = \phi_q(x) + \phi_q(y) \quad \text{e} \quad \phi_q(xy) = \phi_q(x)\phi_q(y);$$

(iii) se $a \in \overline{\mathbb{F}}_q$*, allora*

$$a \in \mathbb{F}_{q^m} \iff \phi_q^m(a) = a;$$

(iv) ϕ_q *è un automorfismo di* $\mathbb{F}_q$ *e il gruppo di Galois dell'estensione* $\mathbb{F}_q/\mathbb{F}_p$ *è generato da* ϕ_p*.*

1.1.2 Polinomi ciclotomici

Si definisce **estensione ciclotomica** di un campo F il campo di spezzamento del polinomio $f(x) = x^n - 1$ su F. Se ad esempio $F = \mathbb{Q}$, è noto dal teorema fondamentale dell'algebra che un polinomio di grado n ha n zeri nel campo dei numeri complessi: il polinomio f avrà quindi n radici, dette **radici n-esime dell'unità**. L'insieme C di queste radici forma un sottogruppo di $F^\times$, il gruppo moltiplicativo del campo F. C è ciclico per il Teorema 1.1, quindi conterrà almeno un elemento di ordine n (che di conseguenza genererà l'intero gruppo).

Definizione 1.8. *Una radice* n*-esima dell'unità è detta* ***primitiva*** *se genera l'intero gruppo* C*.*

Se il campo F ha caratteristica p, il discorso funziona solamente se p e n sono primi tra loro. In caso contrario, infatti, sia $m = n/p$ (ricordiamo che p è primo, pertanto se n e p non sono primi tra loro significa che $p|n$). Allora per ogni radice n-esima ω dell'unità vale

$$0 = \omega^n - 1 = (\omega^m - 1)^p$$

da cui risulta che l'ordine di ω è minore di n.

Definizione 1.9. *L'*n***-esimo polinomio ciclotomico*** *si definisce come*

$$\Phi_n(x) = \prod_i (x - \omega_i)$$

dove ω_i *sono le radici primitive* n*-esime dell'unità nel campo* $\mathbb{C}$ *dei numeri complessi.*

Le radici n-esime dell'unità in $\mathbb{C}$ sono del tipo $e^{2ir\pi/n}$. Se n e r sono primi tra loro, la radice è primitiva. Quindi il numero di radici primitive dell'unità è uguale al numero di numeri interi minori di n primi con n, ovvero $\phi(n)$, e tale è anche il grado di Φ_n.

Proposizione 1.10. *Per ogni* n *vale*

$$x^n - 1 = \prod_{d|n} \Phi_d(x).$$

In particolare, grazie alla proposizione precedente, si può concludere che se p *è un numero primo allora*

$$\Phi_p(x) = \frac{x^p - 1}{x - 1}.$$

Corollario 1.11. *Esiste un unico sottogruppo di* $\mathbb{F}_{p^n}^\times$ *di ordine* $\Phi_n(p)$.

Dimostrazione. Per la proposizione precedente, $\Phi_n(p)$ divide $p^n - 1$, cardinalità di $\mathbb{F}_{p^n}^\times$. □

Durante tutto il corso di questo testo, chiameremo $G_{p,n}$ il sottogruppo di $\mathbb{F}_{p^n}^\times$ di ordine $\Phi_n(p)$.

Proposizione 1.12. *Sia* q *un fattore primo maggiore di* t *del polinomio* $\Phi_t(p)$. *Allora* q *non divide alcun* $\Phi_s(p)$ *con* s *divisore proprio di* t.

1.1.3 Norme e tracce

Definizione 1.13. *Sia* E/F *un'estensione di campo finita e sia* x *un elemento di* E. *Sia infine* m_x *l'applicazione lineare data da* $m_x(y) = xy$. *Definiamo allora la* ***norma*** N *e la* ***traccia*** T *di* x*, relativamente all'estensione* E/F*, come*

$$\begin{aligned} N_{E/F}(x) &= \det m_x \\ T_{E/F}(x) &= \operatorname{Tr} m_x \end{aligned}$$

Se E/F *è noto, la norma e la traccia di* x *si indicheranno, rispettivamente, con* $N(x)$ *e* $T(x)$.

Definendo norma e traccia in questo modo appare chiaro come esse siano funzioni da E in F. In particolare ci è utile sapere che norma e traccia di un elemento di E è un elemento di F.

Proposizione 1.14. *Sia* E/F *un'estensione di campo separabile di dimensione* n. *Allora si ha*

$$N(x) = \prod_{i=1}^{n} \sigma_i(x) \qquad \text{e} \qquad T(x) = \sum_{i=1}^{n} \sigma_i(x)$$

dove con σ_i *abbiamo indicato gli elementi del gruppo di Galois dell'estensione* E/F. *Conseguentemente, per la norma e la traccia di un elemento* $x \in E$ *valgono*

$$\begin{aligned} N(xy) &= N(x)N(y) \\ T(ax + by) &= aT(x) + bT(y). \end{aligned}$$

1.1.4 Spazi affini e spazi proiettivi

Dato un campo k qualsiasi, possiamo costruire sopra ad esso alcuni tipi di spazi che ci serviranno durante la trattazione.

Definizione 1.15. *Dato un campo* k*, definiamo* ***spazio affine su*** k *la varietà algebrica generata a partire da* k. *Lo spazio affine su un campo* k *si indica con* $\mathbb{A}^1(k)$.

Quindi per i nostri scopi lo spazio affine $\mathbb{A}^1$ sarà una varietà algebrica, in particolare quella che comprende l'intero insieme dei punti di k. Si può estendere senza problemi la definizione a spazi di

dimensioni maggiori $\mathbb{A}^2, \mathbb{A}^3$, e così via, nel qual caso risulta

$$\mathbb{A}^n(k) = \overbrace{\mathbb{A}^1(k) \times \mathbb{A}^1(k) \times \cdots \times \mathbb{A}^1(k)}^{n \text{ volte}}.$$

Definizione 1.16. *Dato un campo* k*, il* ***piano proiettivo*** $\mathbb{P}^2(k)$ *è l'insieme delle classi di equivalenza della relazione* $\sim$ *su* $K^3\backslash\{(0,0,0)\}$*, dove* $(x_1, x_2, x_3) \sim (y_1, y_2, y_3)$ *se e solo se esiste* $\lambda \in k^\times$ *tale che* $(x_1, x_2, x_3) \sim (\lambda y_1, \lambda y_2, \lambda y_3)$*. La classe di equivalenza di* (x_1, x_2, x_3) *si denota con* $(x_1 : x_2 : x_3)$*. Osserviamo che la scrittura* $(0 : 0 : 0)$ *non corrisponde ad un elemento di* $\mathbb{P}^2(k)$*.*

Lo spazio affine (rispettivamente proiettivo) di dimensione n si indicherà semplicemente con $\mathbb{A}^n$ (rispettivamente $\mathbb{P}^n$) se è chiaro dal contesto quale sia il campo k a cui si fa riferimento.

1.1.5 Basi normali ottimali

Sia E/F un'estensione di campi di Galois finita, sia n la dimensione e sia G il rispettivo gruppo di Galois.

Definizione 1.17. *Si definisce* ***normale*** *una base di* E *su* F *della forma* $(\sigma a)_{\sigma \in G}$ *per un fissato* $a \in E\backslash F$ *e per* σ *che varia nel gruppo di Galois* G*.*

È noto che per ogni estensione di Galois finita esiste sempre una base e, in particolare, che il numero dei suoi elementi è la dimensione dell'estensione n.

Sia ora data $(\sigma a)_{\sigma \in G}$ una base normale di E su F e si denotino con $d(\tau, \sigma) \in F$ i coefficienti dell'elemento $a * \sigma a \in E$, ovvero

$$a * \sigma a = \sum_{\tau \in G} d(\tau, \sigma) * \tau a.$$

per ogni $\sigma \in G$. Sommando rispetto a σ otteniamo per il primo membro

$$\sum_{\sigma \in G} a * \sigma a = a * \sum_{\sigma \in G} \sigma a = aT(a),$$

dove la funzione T è la traccia introdotta nella Definizione 1.13, e per il secondo membro

$$\sum_{\sigma \in G} \sum_{\tau \in G} d(\tau, \sigma) * \tau a = \sum_{\tau \in G} \sum_{\sigma \in G} d(\tau, \sigma) * \tau a = \sum_{\tau \in G} \tau a * \sum_{\sigma \in G} d(\tau, \sigma)$$

da cui si ricava che i

$$\sum_{\sigma \in G} d(\tau, \sigma)$$

sono i coefficienti rispetto alla base τa. Eguagliando i termini otteniamo

$$aT(a) = \sum_{\tau \in G} \tau a * \sum_{\sigma \in G} d(\tau, \sigma) = a * \sum_{\sigma \in G} d(1, \sigma) + \sum_{\substack{\tau \in G \\ \tau \neq 1}} \tau a * \sum_{\sigma \in G} d(\tau, \sigma)$$

da cui, visto che la rappresentazione di ciascun elemento di E nella base è unica, otteniamo

$$\begin{aligned} \sum_{\sigma\in G} d(1,\sigma) &= T(a) \\ \sum_{\sigma\in G} d(\tau,\sigma) &= 0 \qquad \text{se } \tau \neq 1. \end{aligned} \tag{1.1}$$

Essendo $(\sigma a)_{\sigma\in G}$ una base, e quindi a un'unità, la matrice dei coefficienti $d(\tau,\sigma)$ è invertibile quindi, per ogni $\tau \in G$ c'è almeno un $d(\tau,\sigma) \neq 0$. Per la (1.1), se $\tau \neq 1$ devono esistere almeno due termini non nulli. In generale, possiamo dire che nella matrice dei coefficienti $d(\tau,\sigma)$ ci saranno almeno $2(n-1)+1 = 2n-1$ termini non nulli. Siamo pronti per la prossima definizione.

Definizione 1.18. *Una base normale è detta* ***ottimale*** *se il numero dei coefficienti non nulli della matrice* $d(\tau,\sigma)$ *è esattamente* $2n-1$.

Diamo ora l'importante risultato proposto in [46] e successivamente dimostrato in [24].

Teorema 1.19. *Sia* E/F *un'estensione di Galois finita con gruppo di Galois* G *e sia* $a \in E$. *Allora* $(\sigma a)_{\sigma\in G}$ *è una base normale ottimale di* E *su* F *se e solo se esiste un numero primo* p*, una radice primitiva* p*-esima dell'unità* ζ *in qualche estensione algebrica di* E *e un elemento* $c \in F^\times$ *tali che valga almeno una delle seguenti:*

(i) il polinomio irriducibile di ζ *su* F *abbia grado* $p-1$*,* $E = F(\zeta)$ *e* $a = c\zeta$*;*

(ii) la caratteristica del campo sia 2, il polinomio irriducibile di $\zeta+\zeta^{-1}$ *su* F *abbia grado* $(p-1)/2$ *e sia* $E = F(\zeta+\zeta^{-1})$ *e* $a = c(\zeta+\zeta^{-1})$.

1.2 Algoritmi e complessità

Poiché nella trattazione attuale di qualsiasi problema crittografico facciamo affidamento ai computer, è divenuto sempre più importante conoscere delle attuali potenzialità dei calcolatori che abbiamo a disposizione. Come vedremo in seguito, i moderni sistemi crittografici si basano sul presupposto fondamentale che trasformare il messaggio nel suo equivalente cifrato sia un'operazione *facile*, mentre risulti estremamente *difficile* ritornare al messaggio di partenza senza avere a disposizione la cosiddetta "chiave". In questo paragrafo metteremo un po' di ordine nelle definizioni di *facile* e *difficile*.

Definizione 1.20. *Definiamo* ***algoritmo*** *una procedura con un numero finito di passi che permette di ottenere la soluzione di un certo problema.*

Ad esempio possiamo considerare algoritmi le regole pratiche per moltiplicare due numeri che vengono insegnate alle elementari, la formula di Cramer per risolvere sistemi lineari, la rappresentazione di un numero in una base diversa da 10, ecc.

Fissato il problema, chiamiamo **istanza** del problema la specificazione effettiva dei suoi parametri. Tornando all'esempio di prima, nell'ambito del problema "moltiplicare due numeri", un'istanza potrebbe essere "moltiplicare 153 per 794", mentre un'altra istanza è "moltiplicare 3 per 5". Ora, sicuramente

le ultime due operazioni, da un punto di vista computazionale, sono molto diverse: è universalmente noto che è più semplice moltiplicare due numeri di una cifra piuttosto che moltiplicarne due di tre. Per questo motivo si parla di **dimensione dell'istanza del problema**, di solito indicata con n. Al problema viene quindi associata una funzione che "conta" il numero di passi elementari di una generica istanza di dimensione n. In particolare $f(n)$ prenderà come valore il numero di passi necessari a risolvere il problema nella sua **istanza peggiore** di dimensione n (ovvero, tra tutte le istanze di dimensione n, quella che necessita del numero di passi maggiore).

Per essere matematicamente più precisi, si introduce usualmente una seconda funzione della forma $O(g(n))$, denominata **ordine di grandezza**, che può essere definita nel seguente modo:

$$f(n) = O(g(n)) \text{ se esistono } \alpha, \overline{n} \text{ tali che } |f(n)| \leqslant \alpha |g(n)| \text{ per } n \geqslant \overline{n}.$$

La notazione indica che $f(n)$ cresce asintoticamente con una velocità non maggiore di $g(n)$.

Detto questo, possiamo finalmente dare una classificazione degli algoritmi secondo la loro **complessità**.

Definizione 1.21. *Diremo che un problema è **trattabile** se esiste un algoritmo in grado di risolvere l'istanza peggiore in tempo polinomiale, ovvero* $g(n) \sim n^r$ *per qualche* r. *I problemi trattabili vengono aggregati nella **classe computazionale P**. I problemi per i quali non si conoscono algoritmi in tempo polinomiale vengono detti **intrattabili**.*

In questa caratterizzazione vengono considerati solamente i cosiddetti **problemi decisionali**, per i quali si hanno risposte del tipo SÌ-NO. Questo approccio non è da considerarsi limitativo, in quanto qualunque problema computazionale può essere ridotto ad uno decisionale in tempo polinomiale, senza quindi perdere la sua appartenenza (o non appartenenza) ad una delle due classi.

Introduciamo ora due nuove classi, di importanza crittografica (e non solo) fondamentale.

Definizione 1.22. *Si definisce **classe computazionale NP** l'insieme di tutti i problemi decisionali per i quali è possibile verificare in tempo polinomiale la correttezza della risposta di tipo SÌ disponendo di un'informazione aggiuntiva denominata certificato (si veda la prossima sezione). Si definisce **classe computazionale co-NP** l'insieme di tutti i problemi decisionali per i quali è possibile verificare in tempo polinomiale la correttezza della risposta di tipo NO, sempre disponendo di un certificato.*

Ogni problema della classe P appartiene banalmente ad entrambe le classi NP e co-NP. Non è noto, attualmente, se le due inclusioni siano strette o meno.

Tra i vari problemi NP ce ne sono alcuni considerati "peggiori" degli altri, che in particolare soddisfano la seguente proprietà.

Definizione 1.23. *Si dice che un problema è **NP-completo** se qualunque problema della classe NP può essere ricondotto ad esso in tempo polinomiale.*

Elenchiamo ora i principali problemi di interesse crittografico appartenenti alla categoria NP-completo.

Problema 1 (delle somme parziali)**.** *Dato un vettore di interi positivi* $a = (a_1, a_2, \ldots, a_n)$ *e un numero intero* s*, determinare se esiste un sottinsieme di* a *che ha come somma* s*, cioè se esiste un vettore* $(x_1, \ldots, x_n)$*, con* $x_i \in \{0, 1\}$ *per ogni* i*, tale che*

$$\sum_{i=1}^{n} x_i a_i = s.$$

Il problema delle somme parziali fu proposto da Merkle e Hellman nel 1978 ed è dimostrato essere NP-completo. È anche stato dimostrato che la sua risoluzione equivale a trovare i numeri a_i che servono per la scomposizione. Tuttavia, nonostante le premesse fossero molto buone, il problema, per poter essere pratico dal punto di vista della crittografia, doveva basarsi su una classe particolare di vettori a, detti **supercrescenti**, per le quali sono stati trovati algoritmi polinomiali che fornissero la soluzione. Si vedano, a tal proposito, [18], [19] e [4].

Problema 2 (del logaritmo discreto)**.** *Dati un campo* F*, un elemento primitivo* $a \in F$ *e un qualsiasi* $b \in F^\times$*, determinare l'unico intero* n *positivo (minore di* $|F|$*) tale che valga*

$$a^n = b \qquad \textit{o, equivalentemente} \qquad x = \log_a b.$$

Nella versione originale, proposta da Diffie ed Hellman nel loro famoso articolo del 1976, il campo in questione era $\mathbb{F}_p$ con p primo, che soddisfa le richieste più generali della definizione del problema che abbiamo fornito noi. Nel cercare algoritmi per rendere più sicuro il sistema, si è iniziata la ricerca di nuovi campi finiti computazionalmente più difficili da trattare da parte di un calcolatore. Tutti gli algoritmi e i cifrari trattati in questo testo si basano sul problema del logaritmo discreto.

Problema 3 (della scomposizione in fattori primi)**.** *Dato un numero* n*, trovare* $p_1, p_2, \ldots, p_k$ *primi tali che*

$$n = p_1^{r_1} p_2^{r_2} \cdots p_k^{r_k}.$$

L'ultimo problema enunciato è quello su cui si basa il cifrario RSA, attualmente utilizzato su internet nel protocollo SSL. Proposto nel 1978 da Rivest, Shamir e Adleman (da cui l'acronimo), il cifrario RSA ha da subito catturato l'attenzione della comunità scientifica per la sua semplicità e la sua sicurezza. Ad oggi non si conoscono algoritmi efficienti che possano decifrarlo in un tempo non esponenziale.

Perché dunque cercare nuovi cifrari se ne esiste già uno sicuro? Le risposte a questa domanda esistono e sono molteplici.

- Innanzi tutto per la sicurezza. Nonostante non sia ancora stato trovato un algoritmo efficiente per decifrare RSA rapidamente, non è detto che in futuro questo non venga trovato. Se ciò accadesse, ci sarebbero già altri cifrari pronti a sostituirlo in maniera veloce e (quasi) indolore.
- RSA necessita allo stato attuale di una chiave molto lunga (1024 bit). Altri sistemi potrebbero garantire un'eguale sicurezza con una chiave più corta, come è scaturito dalle curve ellittiche, considerato attualmente il sistema più conveniente da questo punto di vista.
- Infine non è da nascondere un certo piacere nella ricerca fine a se stessa che potrebbe, chissà, fornire soluzioni ad altri problemi, magari completamente slegati da quello originale.

Nella prossima sezione i problemi NP-completi, e in particolare il logaritmo discreto, avranno un'importanza strategica fondamentale. La crittografia a chiave pubblica basa la sua sicurezza sui problemi "intrattabili" che diventano banali con l'aggiunta del cosiddetto certificato.

1.3 Introduzione alla crittografia

La crittografia è la scienza che persegue lo scopo di assicurare l'invio di un messaggio e la ricezione dello stesso (da parte del legittimo destinatario), senza che un eventuale intruso possa leggerne il contenuto. Nato per scopi militari, il concetto di crittografia si è recentemente affermato grazie alla recente e repentina diffusione di internet e alla conseguente necessità di proteggere le informazioni inviate attraverso il Web.

1.3.1 Terminologia

Per tutto il corso di questo testo, così come accade in qualsiasi recente manuale o documento sulla crittografia, **Alice** sarà il mittente dei nostri messaggi, mentre il legittimo destinatario si chiamerà **Bob**. Infine **Eva** farà la parte della "spia" dalle cattive intenzioni, che cercherà quindi in tutti i modi di impossessarsi delle informazioni presenti nel messaggio.

Il messaggio che Alice vuole spedire a Bob è detto **testo in chiaro**: per poter viaggiare attraverso un canale insicuro esso viene trasformato, attraverso un **sistema di cifratura** e una **chiave**, in un **crittogramma**. Chiave e sistema di cifratura non sono la stessa cosa: in un mondo dove sono sempre più utilizzati canali insicuri come internet, si deve supporre che il sistema di cifratura sia unico e di dominio pubblico, mentre la chiave è un parametro scelto dal mittente (o dal destinatario legittimo) che deve rimanere segreto. L'operazione che trasforma un testo in chiaro in un crittogramma viene detta **cifratura** (o **crittazione**) mentre l'operazione inversa è detta **decifrazione**. Si parla invece di **crittoanalisi** quando si vuole ricostruire il testo in chiaro a partire dal crittogramma senza avere a disposizione la chiave: quest'ultimo è il ruolo della spia (Eva nel nostro caso).

Definizione 1.24. *Un* ***sistema di cifratura****, o* ***cifrario****, è una* 5*-pla* $(\mathcal{M}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$*, dove*

- $\mathcal{M}$ *è l'insieme di tutti i possibili messaggi in chiaro;*
- $\mathcal{C}$ *è l'insieme di tutti i possibili messaggi cifrati;*
- $\mathcal{K}$ *è l'insieme di tutte le possibili chiavi;*
- *Per ogni* $k \in \mathcal{K}$*, esistono una funzione* $e_k \in \mathcal{E}$ *e una funzione* $d_k \in \mathcal{D}$ *tali che* $e_k : \mathcal{M} \to \mathcal{C}$*,* $d_k : \mathcal{C} \to \mathcal{M}$ *e* $d_k(e_k(x)) = x$ *per ogni* $x \in \mathcal{M}$*.*

Alice e Bob scelgono una chiave da utilizzare. Con il termine "scegliere" intendiamo che Alice e Bob si siano messi d'accordo in qualche modo sulla chiave. Vedremo più avanti che esistono metodi per fare questo anche attraverso canali insicuri. Sia $k \in \mathcal{K}$ la chiave scelta al riparo dagli occhi indiscreti di Eva.

Sia $m \in \mathcal{M}$ il messaggio che Alice intende spedire a Bob. Esso sarà una stringa di caratteri alfabetici o numerici. Nella pratica la funzione e_k innanzi tutto divide m in modo che

$$m = m_1 m_2 \dots m_n$$

per qualche intero $n \geqslant 0$ e ogni m_i sia di lunghezza fissata. Ognuno di essi viene ora crittato usando un'opportuna funzione $\overline{e}_k$ relativa alla chiave k scelta. Alice calcola quindi $c_i = \overline{e}_k(m_i)$ e spedisce la stringa

$$c = e_k(m) = c_1 c_2 \dots c_n$$

a Bob. Quest'ultimo utilizzerà d_k in suo possesso per decifrare c ed ottenere il messaggio di partenza m. Se l'operazione è stata svolta in maniera corretta, Eva può venire in possesso solamente del crittogramma c: la sicurezza del messaggio m dipenderà dalla difficoltà per Eva di arrivare da c a m senza avere k.

1.3.2 La crittografia a chiave pubblica

Per introdurre il concetto di **crittografia a chiave pubblica** è bene affiancarlo prima con quello di **crittografia a chiave privata**, utilizzato fino agli anni Ottanta. Si può sintetizzare efficacemente nel seguente modo.

> *Alice si reca da un ferramenta e acquista un lucchetto, il quale è provvisto solo di due chiavi assolutamente identiche. Successivamente Alice incontra Bob e gli consegna una delle due chiavi. Alice scrive quindi un messaggio, lo introduce in una scatola, chiude tale scatola con il lucchetto e invia tutto a Bob, che chiaramente è l'unica persona che possa aprirla. Quest'ultimo risponde utilizzando lo stezzo procedimento. Ciò che può fare Eva è intercettare la scatola e impedire al destinatario di riceverla, ma non venire a conoscenza del contenuto.*

Risulta chiaro dall'esempio che, per poter utilizzare un sistema a chiave privata, Alice e Bob devono prima accordarsi segretamente su una chiave $k \in \mathcal{K}$ che intendono utilizzare. Questo scambio deve avvenire in un modo sicuro, per esempio a voce in un luogo rumoroso, al riparo da orecchie indiscrete (in particolare da quelle di Eva). Una volta superato questo scoglio iniziale, Alice e Bob possono finalmente scambiarsi le informazioni in tutta tranquillità senza che Eva possa leggerle.

Nonostante il sistema appena descritto sia adeguato per molte applicazioni, esso presenta alcuni ostacoli che lo rendono difficilmente applicabile, in particolare se si pensa all'uso "massiccio" della crittografia nel web:

- potrebbe non esistere un canale sicuro attraverso cui comunicare la chiave da utilizzare;
- in una rete (come quella di internet) con n utenti, ogni possibile coppia di utenti deve possedere una chiave, per un totale di $n(n-1)/2$ chiavi, soluzione poco pratica per valori di n molto grandi;
- non è possibile ottenere una firma digitale da un sistema a chiave privata (vedere [49]).

Nel tentativo di risolvere questi inconvenienti, nel 1976 Diffie, Hellman e Merkle scoprirono la crittografia a chiave pubblica. Per introdurla, riprendiamo la metafora del lucchetto esposta in precedenza.

Alice consegna a Bob un lucchetto aperto di cui solo lei possiede la chiave. In seguito Bob scrive il messaggio, lo introduce in una scatola, la chiude con il lucchetto e invia il tutto ad Alice, che può leggere il messaggio. Di nuovo Eva potrà intercettare il lucchetto aperto, la scatola chiusa, ma non avrà la possibilità di decifrare il messaggio.

In pratica il lucchetto è una funzione invertibile che trasforma una scatola chiusa in una aperta: mentre è molto semplice il problema diretto (chiudere la scatola), diventa quasi impossibile quello inverso (aprire la scatola), a meno di avere un'informazione aggiuntiva (la chiave). Con la chiave aprire la scatola diventa banale. L'idea centrale della crittografia a chiave pubblica è quella di separare chiave e lucchetto, in modo che nessuno (fuorché il legittimo destinatario) possa aprire il lucchetto, nemmeno chi l'ha chiuso.

Definizione 1.25. *Una **funzione unidirezionale** è una funzione* $e : \mathcal{M} \to \mathcal{C}$ *invertibile, tale che per ogni* $m \in \mathcal{M}$ *sia facile calcolare* $c = e(m)$*, mentre per ogni* $c \in \mathcal{C}$ *sia molto difficile calcolare* $d(c)$ *dove* $d : \mathcal{C} \to \mathcal{M}$ *è tale che* $m = d(e(m))$.

Più precisamente il problema diretto deve appartenere alla categoria P dei problemi calcolabili in tempo polinomiale, mentre il problema inverso deve essere NP-completo, categoria alla quale appartiene, come già visto, il logaritmo discreto.

Definizione 1.26. *Una funzione unidirezionale è detta **funzione trappola** se, ottenuta un'informazione aggiuntiva (detta **certificato**), può essere invertita facilmente, ovvero in tempo polinomiale.*

Rifacendoci alle notazioni della definizione 1.24, in un sistema crittografico a chiave pubblica l'algoritmo e_k è di pubblico dominio, mentre l'algoritmo d_k di decifrazione rimane segreto. Il certificato è la chiave k, attraverso cui si può trovare velocemente d_k. La difficoltà per Eva in questo caso risiede nell'ottenere d_k dato e_k.

Come possono Alice e Bob comunicare in sicurezza? Alice sceglie una chiave a e rende pubblico l'algoritmo e_a (chiave pubblica). Bob, analogamente, sceglie una chiave b e rende pubblico e_b. A questo punto ogniqualvolta Alice vuole spedire un messaggio m_1 a Bob calcola $e_b(m_1)$. Bob può facilmente trovare d_b, in quanto possiede la chiave (certificato) b. Può poi rispondere ad Alice con un messaggio m_2 utilizzando la sua chiave pubblica e_a. Eva, invece, intercettando $e_b(m_1)$ e $e_a(m_2)$ senza i rispettivi a e b, si troverà sempre davanti ad un problema di tipo esponenziale. In compenso potrà utilizzare le chiavi pubbliche di Alice e Bob per comunicare con loro. Sta proprio in questo la potenza della chiave pubblica: ogni utente possiede una sua chiave pubblica, attraverso la quale chiunque può comunicare con lui in modo sicuro.

Vediamo ora alcuni sistemi di crittografia a chiave pubblica che utilizzano come funzione trappola il logaritmo discreto, di cui si è parlato nella sezione 1.2.

1.3.3 Lo scambio di chiavi di Diffie ed Hellman

Questo algoritmo è il primo caso di crittografia a chiave pubblica e ha la peculiarità che entrambe le parti (Alice e Bob) partecipino alla creazione della chiave. Esso si può implementare nel modo seguente:

- Alice e Bob scelgono un gruppo finito G e un elemento $\mathbf{g} \in G$, anche attraverso un canale insicuro;

- Alice quindi sceglie un intero casuale a, calcola $\mathbf{c} = \overbrace{\mathbf{g} + \mathbf{g} + \cdots + \mathbf{g}}^{a \text{ volte}} = a\mathbf{g}$ in G;
- Alice trasmette a Bob l'elemento $\mathbf{c}$ così ottenuto;
- Bob sceglie a sua volta un intero b, calcola $\mathbf{d} = b\mathbf{g}$ in G;
- Bob trasmette $\mathbf{d}$ ad Alice;
- Alice riceve $\mathbf{d}$ e calcola $a\mathbf{d} = a(b\mathbf{g}) = (ab)\mathbf{g}$;
- Bob riceve $\mathbf{c}$ e calcola $b\mathbf{c} = b(a\mathbf{g}) = (ba)\mathbf{g}$.

Chiaramente $(ab)\mathbf{g} = (ba)\mathbf{g}$ quindi Alice e Bob hanno in mano lo stesso elemento, tranquillamente utilizzabile come chiave.

Cosa succede se Eva intercetta i messaggi? Quest'ultima avrebbe in mano solamente G, $\mathbf{g}$, $a\mathbf{g}$ e $b\mathbf{g}$. Per poter ottenere $(ab)\mathbf{g}$ dovrebbe riuscire a calcolare a o b, ma questo problema, come visto in precedenza, appartenente alla categoria NP-completo.

1.3.4 Il cifrario di ElGamal

L'algoritmo che segue è stato sviluppato da ElGamal nel 1985.

Innanzi tutto vengono scelti un gruppo finito G e un elemento $\mathbf{g} \in G$. Questi valori possono essere condivisi anche da vari utenti. La riservatezza sta nella scelta dell'intero a da parte del destinatario e dell'intero k da parte del mittente, come si vede dall'algoritmo sottostante. Il messaggio deve essere preventivamente trasformato in un elemento $\mathbf{m}$ di G attraverso un sistema su cui Alice e Bob si accordano (il metodo può essere anche di dominio pubblico).

- Alice sceglie un intero a e calcola $\mathbf{a} = a\mathbf{g}$; a è la chiave privata di Alice, $\mathbf{a}$ è la sua chiave pubblica.

Bob, per inviare un messaggio ad Alice, procede nel seguente modo:

- Bob sceglie un intero casuale k e calcola $k\mathbf{g}$ (ricordiamo che G e $\mathbf{g}$ sono pubblici);
- Bob calcola inoltre $k\mathbf{a}$ e successivamente $\mathbf{m} + k\mathbf{a}$ ($\mathbf{a}$ è la chiave pubblica di Alice);
- Bob invia la coppia $(k\mathbf{g}, \mathbf{m} + k\mathbf{a})$ ad Alice.

È da notare come la crittazione dipenda anche da un valore aleatorio k, scelto dal mittente di volta in volta e sconosciuto anche al destinatario legittimo. Questa particolarità incrementa la sicurezza del sistema: lo stesso messaggio, cifrato in tempi diversi, può dare luogo a crittogrammi diversi. Proprio per questo motivo la coppia $(k\mathbf{g}, \mathbf{m} + k\mathbf{a})$ sembra "ridondante" rispetto al messaggio: in questo modo Alice (destinatario legittimo) può avere informazioni su k attraverso il primo elemento $k\mathbf{g}$ e sul messaggio $\mathbf{m}$ grazie al secondo elemento $\mathbf{m} + k\mathbf{a}$.

Infine la decifrazione da parte di Alice avviene nel modo seguente:

- Alice somma $k\mathbf{g}$ un numero a di volte ottenendo $a(k\mathbf{g}) = k(a\mathbf{g}) = k\mathbf{a}$;

- Alice inverte facilmente l'elemento $k\mathbf{a}$ e trova $\mathbf{b} = -k\mathbf{a}$;
- infine Alice calcola $\mathbf{m}$ in questo modo: $\mathbf{m} = \mathbf{m} + k\mathbf{a} + \mathbf{b} = \mathbf{m} + k\mathbf{a} - k\mathbf{a} = \mathbf{m}$.

Come già osservato sopra, la presenza del parametro k da una parte aumenta il grado di sicurezza, ma dall'altro raddoppia la lunghezza del messaggio criptato. Affinché però il vantaggio sia reale, Bob deve cambiare spesso il valore di k, eventualmente ad ogni messaggio. Ovviamente questo non è un problema se a preoccuparsene è un computer.

2
Nuove soluzioni

2.1 Le curve ellittiche

L'applicazione delle curve ellittiche alla crittografia ha inizio nel 1985, quando viene proposta, indipendentemente, da Neal Koblitz e Victor Miller.

L'idea di base è quella di riprendere cifrari a chiave pubblica già noti (tra quelli che si basano sul problema del logaritmo discreto) utilizzando come gruppo quello formato dai punti di una curva ellittica costruita su un campo finito. Il vantaggio di usare le curve ellittiche costruite sui campi $\mathbb{F}_p$ piuttosto che i campi $\mathbb{F}_p$ stessi è sostanzialmente computazionale: per garantire un grado di sicurezza paragonabile a quello dei cifrari tradizionali sono sufficienti meno bit.

Definizione 2.1. *Sia* k *un campo. Un'equazione della forma*

$$F(x,y,z) = y^2z + a_1xyz + a_3yz^2 - x^3 - a_2x^2z - a_4xz^2 - a_6z^3 = 0 \tag{2.1}$$

dove i parametri $a_1, \cdots, a_6$ *e le variabili* x, y, z *sono elementi del campo* k *è detta* ***equazione di Weierstrass****.*

Definizione 2.2. *Dato un campo* k*, una* ***curva ellittica*** E/k *è l'insieme delle soluzioni in* $\mathbb{P}^2(k)$ *di una equazione di Weierstrass.*

Se ci limitiamo al caso $z \neq 0$, per identificare un punto in $\mathbb{P}^2(k)$, la terza coordinata z è ridondante; per la relazione di equivalenza ereditata dallo spazio proiettivo possiamo sempre moltiplicare un punto (x,y,z) per $1/z$ ottenendo il punto equivalente $(x/z, y/z, 1)$. Questo ci permette di semplificare l'equazione (2.1) attraverso la sostituzione (per $z \neq 0$)

$$(x,y,z) \longmapsto (x/z, y/z, 1)$$

che ci porta alla nuova equazione (detta equazione di Weierstrass non omogenea)

$$f(x,y) = y^2 + a_1xy + a_3y - x^3 - a_2x^2 - a_4x - a_6 = 0 \tag{2.2}$$

dove abbiamo ribattezzato x/z e y/z rispettivamente x e y.

Per $z \neq 0$, quindi, le soluzioni in $\mathbb{P}^2(k)$ dell'equazione di Weierstrass (2.1) possono essere messe in corrispondenza biunivoca con le soluzioni in $\mathbb{A}^2(k)$ della (2.2).

Cosa succede se $z = 0$? Azzerando la terza coordinata nell'equazione (2.1) troviamo $x^3 = 0$. Tutti i punti ottenuti da questa operazione sono quelli dell'insieme $(0, y, 0)$ al variare di y, con $y \neq 0$, il che vuol dire (data la definizione di spazio proiettivo) che tale insieme è composto dall'unico elemento $(0 : 1 : 0)$.

Definizione 2.3. *Sia data una curva ellittica E/k. Il punto $(0 : 1 : 0)$ è detto **punto all'infinito** della curva e si indica con $\mathcal{O}$.*

Possiamo quindi fornire una nuova definizione (più utile per i calcoli) di curva ellittica: dato un campo k, una curva ellittica E/k è in pratica l'insieme delle soluzioni in $\mathbb{A}^2(k)$ dell'equazione (2.2) più il punto all'infinito $\mathcal{O}$ della definizione 2.3.

2.1.1 Curve singolari

Come già visto, una curva ellittica E/k è l'insieme delle soluzioni di una funzione del tipo (2.1) oppure (2.2). Essendo di terzo grado, ci si aspetta che una retta incontri la curva in tre punti (contati con le rispettive molteplicità). Vediamo come evitare curve "anomale", cioè in cui la tangente a qualche punto non è ben definita: questo fatto impedirebbe la legge di gruppo che tra poco andremo a descrivere.

Definizione 2.4. *Sia E/k una curva di equazione (2.1). Un punto $P = (x, y)$ si dice **punto singolare** se e solo se*

$$\frac{\partial F}{\partial x}(P) = \frac{\partial F}{\partial y}(P) = \frac{\partial F}{\partial z}(P) = 0.$$

*La curva E/k si dice **curva non singolare** se non ha punti singolari.*

Silverman, in [57], ci assicura che una "scelta aleatoria" dei coefficienti $a_1, \ldots, a_6$ produce con buona probabilità una curva non singolare.

2.1.2 La legge di gruppo

Sia L una retta in $\mathbb{P}^2(k)$. Se L interseca la curva in due punti, allora sicuramente la intersecherà anche in un terzo punto (due o tre dei punti in questione potrebbero essere coincidenti, nel qual caso L risulterebbe tangente a E/k).

Definizione 2.5. *Sia E/k una curva ellittica e siano $P, Q \in E$, L la retta che congiunge P con Q, e R il terzo punto di intersezione di E con L. Siano ora L' la retta passante per R e $\mathcal{O}$ e R' il terzo punto di intersezione tra E ed L'. Allora R' è la **somma** $\mathbf{P \oplus Q}$ (si veda la figura 2.1).*

R' è in realtà il punto speculare di R rispetto all'asse di simmetria della curva (per uno studio nel caso generale si veda [8]).

Ora che abbiamo definito come si effettuano le operazioni con i punti della curva, possiamo enunciare il seguente teorema.

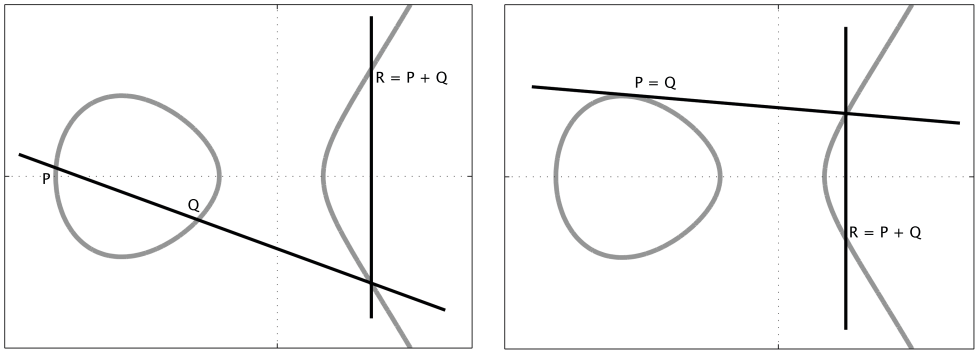

Figura 2.1: Come si sommano i punti (caso incidente e caso tangente).

Teorema 2.6. *Una curva ellittica* E/k *non singolare è un gruppo abeliano rispetto all'operazione* $\oplus$ *con elemento neutro* $\mathcal{O}$*.*

Se il campo k su cui viene costruita la curva è formato da un numero finito di elementi, anche il gruppo godrà della medesima proprietà: in questo modo può essere utilizzato per implementare i cifrari descritti in precedenza che sfruttano le difficoltà derivanti dal logaritmo discreto.

Concludiamo questa sezione con un teorema che fornisce la struttura del gruppo.

Teorema 2.7. *Sia* E/k *una curva ellittica e sia* q *la cardinalità di* k*. Allora il gruppo formato dai suoi punti è isomorfo a* $\mathbb{Z}_{n_1} \oplus \mathbb{Z}_{n_2}$ *dove* $n_1 | n_2$ *e* $n_2 | q - 1$*.*

Ovviamente le curve ellittiche sono soggette a tutti i tipi di attacco "generico" contro il logaritmo discreto; tuttavia, essendo computazionalmente più difficile trattare con esse piuttosto che con i campi finiti $\mathbb{F}_{p^n}$, qualsiasi attacco diretto al problema diventa ancora più oneroso in termini di risorse.

Esistono tuttavia attacchi studiati ad hoc per le curve ellittiche, che sfruttano alcune caratteristiche che la curva può assumere se non vengono fatti opportuni controlli.

2.1.3 Le curve supersingolari e le curve anomale

Definizione 2.8. *Sia* E/k *una curva ellittica e sia* n *un numero positivo. Un punto* P *è detto* **punto di n-torsione se** $nP = \mathcal{O}$**.**

Riferendoci alla figura 2.1, vediamo che ad esempio i 2-punti di torsione sono tutti e soli i punti con tangente verticale (oltre ovviamente il punto $\mathcal{O}$).

Chiaramente, presi due punti di n-torsione, la loro somma mantiene la stessa proprietà. L'insieme $E[n]$ definito da

$$E[n] = \{P \in E/\overline{k} \text{ tale che } nP = \mathcal{O}\}$$

può essere quindi visto come gruppo astratto degli n-punti di torsione di E/k.

Il teorema seguente fornisce la struttura del gruppo degli n-punti di torsione.

Teorema 2.9. *Sia* E/k *una curva ellittica e sia* n *un numero intero positivo. Sia inoltre* p *la caratteristica di* k*.*

- *Se* $p \nmid n$ *oppure* $p = 0$*, allora*
$$E[n] \simeq \mathbb{Z}_n \oplus \mathbb{Z}_n.$$
- *Se* $p|n$*, sia* $n = p^r n'$ *con* $p \nmid n'$*. Allora*
$$E[n] \simeq \mathbb{Z}_{n'} \oplus \mathbb{Z}_{n'} \quad \text{oppure} \quad E[n] \simeq \mathbb{Z}_n \oplus \mathbb{Z}_{n'}.$$

Il secondo punto di questo teorema ci permette di dividere le curve ellittiche utili in crittografia in due categorie.

Definizione 2.10. *Una curva ellittica* E *definita su un campo* K *di caratteristica* p *è detta* ***supersingolare*** *se* $E[p] \simeq 0$*. Altrimenti la curva è detta* ***ordinaria****.*

Diamo invece un'altra categoria cui una curva ellittica può appartenere.

Definizione 2.11. *Una curva ellittica* E *definita su un campo* $k = \mathbb{F}_q$ *è detta* ***anomala*** *se* $|E/\mathbb{F}_q| = q$*.*

Senza entrare nel dettaglio, si dimostra che per queste due categorie di curve il problema del logaritmo discreto sul gruppo dei punti si può ridurre al logaritmo discreto nel campo su cui la curva è costruita, per cui il cifrario perde di convenienza.

2.1.4 Contare i punti

Un particolare attacco (semplificazione di Pohlig-Hellman) per il calcolo del logaritmo discreto funziona tanto meglio quanto la cardinalità del gruppo è scomponibile in fattori primi piccoli. Quello che ci serve è dunque conoscere il numero di punti del gruppo utilizzato.

Definizione 2.12. *Sia data* $E/\mathbb{F}_q$ *curva ellittica e sia* $|E/\mathbb{F}_q| = q+1-t$ *il numero dei suoi punti (cardinalità). L'intero* t *è detto* ***traccia di Frobenius****.*

Un primo risultato fondamentale è il teorema di Hasse, che pone un intervallo in cui cade la cardinalità del gruppo.

Teorema 2.13. *Sia* E/K *una curva ellittica. La traccia di Frobenius* t *soddisfa la disequazione*
$$|t| \leqslant 2\sqrt{q}.$$

Sia E una curva ellittica definita su un campo finito $\mathbb{F}_q$. L'automorfismo di Frobenius (vedere sezione 1.1) agisce sui punti di $E/\overline{\mathbb{F}}_q$ in questo modo:
$$\phi_q(x, y) = (x^q, y^q) \quad \text{e} \quad \phi_q(\mathcal{O}) = \mathcal{O}.$$

Proposizione 2.14. *Sia* E *una curva ellittica definita su* $\mathbb{F}_q$*, e sia* (x, y) *un punto di* $E/\overline{\mathbb{F}}_q$*. Allora valgono:*

(i) $\phi_q(x,y) \in E/\overline{\mathbb{F}}_q$ *cioè* ϕ_q *si restringe ad una mappa da* $E/\overline{\mathbb{F}}_q$ *a* $E/\overline{\mathbb{F}}_q$*;*

(ii) $\phi_q : E/\overline{\mathbb{F}}_q \to E/\overline{\mathbb{F}}_q$ *è un omomorfismo di gruppi e una mappa razionale, cioè è un* ***endomorfismo di*** $E/\overline{\mathbb{F}}_q$*;*

(iii) $(x,y) \in E/\mathbb{F}_q$ *se e solo se* $\phi_q(x,y) = (x,y)$.

Teorema 2.15. *Sia* $E/\mathbb{F}_q$ *una curva ellittica. Allora* t *è la traccia di Frobenius di* E *se e solo se vale*

$$\phi_q^2 - t\phi_q + q = 0 \tag{2.3}$$

come endomorfismo di $E/\overline{\mathbb{F}}_q$*, dove* ϕ_q *è mappa di Frobenius. Inoltre* t *è l'unico intero che rende vera la (2.3).*

In altre parole, se $(x,y) \in E/\overline{\mathbb{F}}_q$*, allora*

$$\left(x^{q^2}, y^{q^2}\right) - t(x^q, y^q) + q(x,y) = \mathcal{O}$$

e t *è l'unico intero che rende vera questa equazione per ogni punto* (x,y) *della curva.*

Il teorema 2.13 fornisce un intervallo entro il quale cade il numero di punti di una generica curva ellittica su un campo $\mathbb{F}_q$. Tuttavia se $q = p^m$ con p sufficientemente piccolo risulta particolarmente pratico un teorema che calcola il valore di t per $\mathbb{F}_q$ dato il valore t per $\mathbb{F}_p$.

Teorema 2.16. *Sia* $E/\mathbb{F}_p$ *una curva ellittica e sia* t_p *la sua traccia di Frobenius. Si risolva l'equazione di secondo grado data da* $x^2 - t_p x + p = (x-\alpha)(x-\beta)$*. Allora*

$$t_{p^m} = \alpha^m + \beta^m$$

dove t_{p^m} *è la traccia di Frobenius di* $E/\mathbb{F}_{p^m}$.

2.1.5 Scegliere la curva

Dopo aver descritto in breve quali sono i tipi di curve ellittiche utilizzabili teoricamente per scopi crittografici, vediamo un algoritmo che risponda nella pratica alle nostre esigenze.

Un algoritmo per selezionare una buona curva può essere riassunto come segue.

(i) scegliere un intero piccolo $\overline{s}$ e un campo finito $\mathbb{F}_q$;

(ii) scrivere l'equazione di una curva E con coefficienti aleatori in $\mathbb{F}_q$;

(iii) calcolare $|E/\mathbb{F}_q|$;

(iv) controllare la supersingolarità, in caso positivo tornare al punto (ii);

(v) controllare che la curva non sia anomala, in caso contrario tornare al punto 2;

(vi) eseguire il test di fattorizzazione con l'intero $\overline{s}$; se non si riesce tornare al punto (ii);

(vii) sia $|E/\mathbb{F}_q| = sr$ con r il primo più grande che divide $|E/\mathbb{F}_q|$; se $s > \overline{s}$ tornare al punto (ii);

(viii) E è la curva cercata.

L'intero $\overline{s}$ serve per stabilire quando il test di fattorizzazione del punto 6 fallisce o meno. Si procede in questo modo: si prova a dividere $|E/\mathbb{F}_q|$ per tutti i primi più piccoli di $\overline{s}$ finché si ottiene un numero r, divisore di $|E/\mathbb{F}_q|$, che non è più diviso da nessun primo minore di $\overline{s}$; si esegue quindi un test di primalità su r; se il test fallisce si torna al punto (ii).

2.2 LUC

Il cifrario LUC prende il nome dalle funzioni di Lucas ed è stato proposto per la prima volta nel 1993 da Peter J. Smith e Michael J. J. Lennon come soluzione a un problema di sicurezza nell'utilizzo del cifrario RSA per la firma digitale.

Tuttavia, studi successivi (cfr. [9]) hanno messo in evidenza come il cifrario LUC sia in realtà la punta di un "iceberg", quello dei cifrari basati sui polinomi ciclotomici.

A differenza delle curve ellittiche, il sistema LUC non si basa sull'utilizzo di nuovi gruppi in cui le operazioni sono più difficili e quindi i tentativi di comprometterne la sicurezza ancora più infattibili da un punto di vista computazionale. Al contrario, LUC ha come obiettivo quello di cercare campi finiti ad hoc, attraverso i quali si riescono ad inviare meno bit di informazioni mantenendo inalterato il livello di sicurezza.

Sia dunque p un numero primo di almeno 512 bit, tale che $p+1$ contenga un fattore primo q di almeno 160 bit. Consideriamo ora il campo $\mathbb{F}_{p^2}$, estensione di $\mathbb{F}_p$, il cui gruppo moltiplicativo è di ordine $p^2-1 = (p+1)(p-1)$. Poiché q divide $p+1$, possiamo trovare un elemento $g \in \mathbb{F}_{p^2}$ di ordine q. L'elemento g e i numeri primi p e q sono la chiave pubblica del sistema.

Ora si procede secondo lo scambio di Diffie-Hellman. Alice sceglie un numero $x_A < q$, costruisce $A = g^{x_A} + g^{-x_A}$ e lo spedisce a Bob. Quest'ultimo, allo stesso modo, sceglie $x_B < q$, costruisce $B = g^{x_B} + g^{-x_B}$ e lo invia ad Alice. Entrambi avranno a disposizione la chiave privata $S = g^{x_A x_B} + g^{-x_A x_B}$.

Prima di descrivere in che modo Alice e Bob possano ottenere S a partire da A e B, apriamo un'importante parentesi su questi due termini. Sia h un qualsiasi elemento del gruppo generato da g. Scrivendo $h = g^n$ per qualche n, notiamo che

$$h^{p-1} = h^{qk} = g^{nqk} = (g^q)^n k = 1$$

da cui deriva immediatamente la relazione $h^p = h^{-1}$. Elevando A alla potenza p si ottiene

$$A^p = (g^{x_A} + g^{-x_A})^p = g^{x_A p} + g^{-x_A p} = g^{-x_A} + g^{x_A} = A,$$

ovvero A è un elemento di $\mathbb{F}_p$. Scambiando A con B si ottiene lo stesso algoritmo per Bob. Grazie a questa proprietà, Alice e Bob possono scambiarsi solamente p, q e $(g + g^{-1})$ (invece che g). Essendo

quest'ultimo un elemento di $\mathbb{F}_p$, può essere trasmesso utilizzando al massimo solamente 512 bit. Valgono gli stessi risultati se al posto dello scambio di Diffie-Hellman viene utilizzato il cifrario RSA.

Per determinare S, dunque, si procede nel seguente modo. Alice riceve B da Bob, e deve riuscire a ottenere g^{x_B}. Denotando $u = g^{x_B}$, il lavoro di Alice consiste nel risolvere l'equazione $B = u + u^{-1}$. Per fare questo è sufficiente, con metodi standard, risolvere l'equazione $u^2 - Bu + 1 = 0$ in $\mathbb{F}_p$ (per quanto detto sopra). Una volta ottenuti i valori $u_{1,2}$ in $\mathbb{F}_{p^2}$, Alice calcola $S = u_1^{x_A} + u_2^{x_A}$, con u_1 e u_2 nell'ordine che preferisce visto che il campo è commutativo. Similmente si comporta Bob (basta sostituire A con B).

2.2.1 La sicurezza del cifrario LUC

Per dimostrare la sicurezza del cifrario LUC, facciamo vedere che se Eva riesce a decifrare LUC, allora riesce anche a decifrare Diffie-Hellman nel gruppo $\langle g \rangle$. In questo modo possiamo concludere che LUC è sicuro almeno quanto il Diffie-Hellman classico.

Ricordando che la chiave privata del cifrario di Diffie-Hellman è $g^{x_A x_B}$, immaginiamo che Eva abbia ottenuto in qualche modo $\alpha = g^{x_A}$ e $\beta = g^{x_B}$. Eva dovrebbe quindi procedere costruendo $\alpha + \alpha^p$ e $\beta + \beta^p$, determinando $S_1 = g^{x_A x_B} + g^{x_A x_B p}$ utilizzando il procedimento descritto in precedenza. Applicando lo stesso discorso a $\beta g = g^{x_B + 1}$ si può determinare $S_2 = g^{x_A(x_B+1)} + g^{x_A(x_B+1)p} = \alpha g^{x_A x_B} + \alpha^p g^{x_A x_B p}$. Il tutto si tramuta nel sistema

$$\begin{pmatrix} 1 & 1 \\ \alpha & \alpha^p \end{pmatrix} \cdot \begin{pmatrix} g^{x_A x_B} \\ g^{x_A x_B p} \end{pmatrix} = \begin{pmatrix} S_1 \\ S_2 \end{pmatrix}$$

dal quale si può facilmente estrarre la chiave $g^{x_A x_B}$ cercata. La matrice è regolare in quanto $\alpha \neq \alpha^p$ giacché α non è un elemento di $\mathbb{F}_p$ per costruzione.

2.3 Oltre LUC

Guardando attentamente il cifrario LUC, viene da chiedersi se non si può migliorare utilizzando campi $\mathbb{F}_{p^n}$ con n maggiore di 2. Hanno risposto positivamente a questa domanda Brouwer, Pellikaan e Verheul nel 1999 (cfr. [9]).

Lo scopo dei tre autori è di estendere il metodo LUC passando da $\mathbb{F}_{p^2}$ a $\mathbb{F}_{p^6}$ e dimostrando che è sufficiente conoscere 2 dei 6 coefficienti del polinomio minimo dell'elemento scelto.

Sia p un numero primo di almeno 171 bit tale che il polinomio ciclotomico $\Phi_6(p) = p^2 - p + 1$ contenga un fattore primo q di almeno 160 bit. Poiché $\Phi_6(p)$ divide $p^6 - 1$, possiamo trovare un elemento g in $\mathbb{F}_{p^6}^{\times}$ di ordine q. Come per il LUC, i parametri del nostro sistema sono p, q e g.

Alice sceglie un numero x_A minore di q e forma il polinomio $P_A(X)$ dato da

$$P_A(X) = \prod_{i=0}^{5} (X - g^{x_A p^i}) = X^6 + A_5 X^5 + A_4 X^4 + A_3 X^3 + A_2 X^2 + A_1 X + 1$$

dove tutti gli A_i sono in $\mathbb{F}_p$ per costruzione. Alice spedirà quindi a Bob la coppia (A_1, B_1). Allo stesso modo Bob sceglierà un x_B minore di q, genererà P_B e invierà ad Alice la coppia (B_1, B_2). La chiave

privata di Alice e Bob sarà data dalla coppia di coefficienti (C_1, C_2) del polinomio

$$P_C(X) = \prod_{i=0}^{5}(X - g^{x_A x_B p^i}) = X^6 + C_5X^5 + C_4X^4 + C_3X^3 + C_2X^2 + C_1X + 1.$$

Alice e Bob, per arrivare a determinarlo, hanno bisogno dei coefficienti mancanti (B_3, B_4, B_5 e A_3, A_4, A_5 rispettivamente). Come nel caso precedente, essi possono essere generati dalla coppia di valori inviati di indici 1 e 2. Vediamo come dal punto di vista di Alice (Bob eseguirà le stesse operazioni sul polinomio P_A).

Innanzi tutto, poiché $p^2 - p + 1$ divide $p^3 - 1$, il polinomio P_B è simmetrico, da cui $B_5 = B_1$ e $B_4 = B_2$. Meno immediato è il calcolo di B_3. Si pone $\beta = g^{x_B}$ e, per ogni i da 0 a 5, $\beta_i = \beta^{p^i}$. Si ha pertanto

$$\beta_0 = \beta \qquad \beta_1 = \beta^p \qquad \beta_2 = \beta^{p-1} \qquad \beta_3 = \beta^{-1} \qquad \beta_4 = \beta^{-p} \qquad \beta_5 = \beta^{1-p}.$$

Sapendo che

$$P_B(X) = \prod_{i=0}^{5}(X - g^{x_B p^i}) = X^6 + B_5X^5 + B_4X^4 + B_3X^3 + B_2X^2 + B_1X + 1,$$

e sviluppando il prodotto $\prod_{i=0}^{5}(X - g^{x_B p^i})$ possiamo scrivere B_3 a partire dai β_i, ottenendo

$$B_3 = -2\sum_{i=1}^{5}\beta_i - \sum_{i=0}^{5}\beta_i^2 - 2.$$

Quest'ultimo è un polinomio simmetrico di grado 2 nelle 6 variabili β_i che quindi può essere scritto, grazie alle uguaglianze di Newton, come

$$B_3 = -2 + 2B_1 - B_1^2 + 2B_2.$$

Ora che Alice ha ricavato l'intero polinomio $P_B(X)$ può prendere una radice ρ di $P_B(X)$ per generare $\mathbb{F}_{p^6}$. Successivamente calcola il polinomio minimo di ρ^{x_A}, che è uguale a $P_C(X)$. Dopo che Bob avrà svolto gli stessi calcoli a partire da (A_1, A_2) e da x_B, entrambi avranno la coppia (C_1, C_2), utilizzabile come chiave privata.

2.3.1 Pregi e difetti

Con le stesse argomentazioni utilizzate per dimostrare la sicurezza del cifrario LUC, si riesce a dimostrare che se Eva fosse in grado di decifrare i messaggi criptati con questo sistema, allora sarebbe in grado di decifrare qualsiasi messaggio criptato utilizzando il sistema di Diffie-Hellman.

Rispetto al LUC è sufficiente trasmettere solamente i due terzi delle informazioni, rendendolo quindi migliore per le applicazioni su web. Come per il LUC, inoltre, questo sistema può essere utilizzato per qualsiasi cifrario che utilizzi l'esponenziale in campi finiti (ad esempio ElGamal, cfr. [16]).

L'unica nota di demerito riguarda l'implementazione: oltre ad essere "ingombrante" nelle sue richieste di memoria, non è particolarmente efficiente e richiede mediamente più tempo di un normale RSA. Proprio per correggere questi problemi, Lenstra e Verheul hanno sviluppato il sistema noto come XTR, protagonista del prossimo capitolo.

3
XTR

XTR è un acronimo che significa Efficient and Compact Subgroup Trace Representation. Già dal nome si può dedurre che XTR non sia, in realtà, un cifrario tradizionale, ma un nuovo metodo per rappresentare "efficientemente" gli elementi di un sottogruppo di un dato gruppo. L'idea alla base è la seguente: data un'estensione E/F, invece di coinvolgere nella generazione della chiave l'intero campo E, il sistema utilizza la **traccia** introdotta nella Definizione 1.13, ovvero una specie di "ombra" che ciascun elemento di E lascia sul campo F. Più in particolare l'estensione presa in considerazione è $\mathbb{F}_{p^6}/\mathbb{F}_{p^2}$.

3.1 Rappresentazione e aritmetica dei sottogruppi

Per poter usufruire dei vantaggi portati dalla rappresentazione comoda del sistema XTR, abbiamo bisogno di trovare p per costruire campi $\mathbb{F}_{p^6}/\mathbb{F}_{p^2}$ ad hoc.

Come per il metodo trattato nella sezione precedente, è necessario un numero primo p "abbastanza grande" tale che il polinomio ciclotomico di ordine 6 calcolato in p abbia un fattore primo q, anch'esso "abbastanza grande". Per i nostri scopi aggiungiamo la condizione (che sarà utile in seguito): $p \equiv 2 \bmod 3$. Sia infine g un elemento di ordine q (il quale esiste sicuramente, per come è stato scelto q).

Grazie alla Proposizione 1.12, q non divide alcun $p^s - 1$ per ogni s divisore di 6, ovvero per $s = 1, 2, 3$. Quindi g non è contenuto in alcun sottocampo proprio di $\mathbb{F}_{p^6}$. L'elemento g, tuttavia, genera il sottogruppo $G_{p,6}$ di $\mathbb{F}_{p^6}^\times$, che sicuramente esiste, introdotto precedentemente.

Poiché $p \equiv 2 \bmod 3$, segue che p genera $\mathbb{F}_3^\times$ e che gli zeri α e α^p del polinomio $X^2 + X + 1$ formano una base normale ottimale di $\mathbb{F}_{p^2}$ su $\mathbb{F}_p$ (rispetta infatti le richieste della Definizione 1.18). Ogni elemento x di $\mathbb{F}_{p^2}$ può quindi essere rappresentato come combinazione lineare di α e $\alpha^p = \alpha^2$ (grazie alla scelta oculata di p), ovvero $x = x_1\alpha + x_2\alpha^2$ con x_1 e x_2 in $\mathbb{F}_p$.

Lemma 3.1. *Dato $x \in \mathbb{F}_{p^2}$, calcolare x^p non costa nulla da un punto di vista computazionale.*

Dimostrazione. Possiamo scrivere $x = x_1\alpha + x_2\alpha^2$ con $x_1, x_2 \in \mathbb{F}_p$. Conseguentemente

$$x^p = x_1^p\alpha^p + x_2^p\alpha^{2p} = x_1\alpha^2 + x_2\alpha$$

quindi basta scambiare i coefficienti per calcolare x^p. □

Procedendo in modo analogo, si può arrivare alle seguenti conclusioni (considereremo computazionalmente insignificante calcolare la somma di due elementi). Denotiamo con $*$ la moltiplicazione in $\mathbb{F}_{p^2}$.

Lemma 3.2. *Siano* $x, y, z \in \mathbb{F}_{p^2}$ *con* $p \equiv 2 \bmod 3$.

(i) Calcolare x^2 *costa due moltiplicazioni in* $\mathbb{F}_p$.

(ii) Calcolare $x * y$ *costa tre moltiplicazioni in* $\mathbb{F}_p$.

(iii) Calcolare $x * z - y * z^p$ *costa quattro moltiplicazioni in* $\mathbb{F}_p$.

Dimostrazione. Scriviamo innanzi tutto le espressioni complete di x, y, z, ovvero

$$x = x_1\alpha + x_2\alpha^2 \qquad y = y_1\alpha + y_2\alpha^2 \qquad z = z_1\alpha + z_2\alpha^2.$$

Si vede facilmente che $x^2 = x_2(x_2 - 2x_1)\alpha + x_1(x_1 - 2x_2)\alpha^2$, da cui segue direttamente il primo punto. Per il secondo punto, si noti che

$$x * y = (x_2y_2 - x_1y_2 - x_2y_1)\alpha + (x_1y_1 - x_1y_2 - x_2y_1)\alpha^2.$$

Ci basta calcolare x_1y_1, x_2y_2 e $(x_1 + x_2)(y_1 + y_2)$ per avere $x_1y_2 + x_2y_1$. Con queste tre moltiplicazioni e quattro sottrazioni arriviamo a $x * y$. Infine, per l'ultimo punto possiamo scrivere

$$x * z - y * z^p = (z_1(y_1 - x_2 - y_2) + z_2(x_2 - x_1 + y_2))\alpha + (z_1(x_1 - x_2 + y_1) + z_2(y_2 - x_1 + y_1))\alpha^2$$

da cui, con un ragionamento simile al precedente, segue la tesi. □

Enunciamo poi un risultato noto relativo alle operazioni in $\mathbb{F}_{p^6}$, utile per fare un confronto (si veda, a tale proposito, [12]).

Lemma 3.3. *Siano* $x, y \in \mathbb{F}_{p^6}$ *con* $p \equiv 2 \bmod 3$ *e siano* $a, b \in \mathbb{Z}$ *positivi e minori di* p. *Assumiamo che elevare al quadrato in* $\mathbb{F}_p$ *costi l'80% rispetto a moltiplicare due elementi di* $\mathbb{F}_p$.

(i) Calcolare x^2 *costa* 14.4 *moltiplicazioni in* $\mathbb{F}_p$.

(ii) Calcolare $x * y$ *costa* 18 *moltiplicazioni in* $\mathbb{F}_p$.

(iii) Calcolare x^a *costa circa* $23.4 \log_2(a)$ *moltiplicazioni in* $\mathbb{F}_p$.

(iv) Calcolare $x^a - y^b$ *costa circa* $27.9 \log_2(\max(a, b))$ *moltiplicazioni in* $\mathbb{F}_p$.

Riprendiamo ora il concetto di traccia di un elemento introdotto nella Definizione 1.13. Nel nostro caso specifico l'estensione cui faremo riferimento è ovviamente $\mathbb{F}_{p^6}/\mathbb{F}_{p^2}$, per cui per ogni $h \in \mathbb{F}_{p^6}$, la traccia $T(h)$ è in $\mathbb{F}_{p^2}$ e vale la relazione

$$T(h) = h + h^{p^2} + h^{p^4}$$

grazie alla Proposizione 1.14.

Lemma 3.4. *Sia* g *un elemento di* $\mathbb{F}_{p^6}^{\times}$ *di ordine* q *(vedi sopra). Allora le radici di* $X^3 - T(g)X^2 + T(g)^p X - 1$ *sono i coniugati di* g.

Dimostrazione. Ricordiamo che i coniugati di g sono, oltre a g stesso, g^{p^2} e g^{p^4}. Dato che stiamo lavorando nel sottogruppo di $\mathbb{F}_{p^6}^{\times}$ di ordine $p^2 - p + 1$, allora $g^{p^2} = g^{p-1}$ e $g^{p^4} = g^{-p}$. Per ottenere questi risultati basta ridurre p^2 e p^4 modulo $p^2 - p + 1$. Per dimostrare il lemma, è ora sufficiente calcolare manualmente i coefficienti di $(X - g)(X - g^{p-1})(X - g^{-p})$. □

Abbiamo in pratica appena dimostrato che il polinomio minimo di g e i suoi coniugati sono completamente determinati da $T(g)$. Il passo successivo è dimostrare che la stessa proprietà vale anche per g^n, ovvero lui e i suoi coniugati, radici di $X^3 - T(g^n)X^2 + T(g^n)^p X - 1$, saranno completamente determinati da $T(g^n)$. Quello che quindi ci interessa affinché il sistema abbia delle implicazioni crittografiche utili è riuscire a calcolare $T(g^n)$ a partire da $T(g)$. Per arrivare a questo, faremo ora alcune considerazioni sul comportamento delle generiche soluzioni del polinomio appena enunciato, generalizzandolo nella prossima definizione.

Definizione 3.5. *Dato* $c \in \mathbb{F}_{p^2}$*, definiamo* $F(c, X)$ *il polinomio* $X^3 - cX^2 + c^p X - 1$ *e* h_0, h_1, h_2 *le radici (non necessariamente distinte) del polinomio in* $\mathbb{F}_{p^6}$*. Infine useremo scrivere* $c_n = h_0^n + h_1^n + h_2^n$.

Enunciamo ora qualche interessante proprietà del polinomio F e dei coefficienti c_n appena introdotti.

Teorema 3.6. *Le radici di* $F(c, X)$ *soddisfano una e una sola delle seguenti proprietà:*

- *hanno tutte ordine maggiore di 3 che divide* $p^2 - p + 1$*;*
- *appartengono tutte a* $\mathbb{F}_{p^2}$.

Dimostrazione. Innanzi tutto si osservi che $h_0 * h_1 * h_2 = 1$, da cui $h_i \neq 0$. Vale quindi la catena di uguaglianze

$$0 = F(c, h_j)^p = h_j^{3p} - c h_j^{2p} + c^p h_j^p - 1 = h_j^{3p} f(c, h_j^{-p})$$

da cui deduciamo che, se h_j è una radice, tale è h_j^{-p}. Possono presentarsi tre casi:

- ciascuna radice è potenza $-p$-esima di se stessa;
- una delle tre radici è potenza $-p$-esima di un'altra e la terza lo è di se stessa;
- le radici sono potenza $-p$-esima in maniera ciclica: la prima della seconda, la seconda della terza e la terza della prima.

Nel primo caso si ha $h_j^{p+1} = h_j * h_j^p = h_j * h_j^{-1} = 1$, per cui $h_j \in \mathbb{F}_{p^2}$. Analogamente si ottiene lo stesso risultato per il secondo. Per l'ultimo punto osserviamo che

$$1 = h_0 * h_1 * h_2 = h_0 * h_0^{-p} * h_2^{(-p)^2} = h_0^{p^2 - p + 1}$$

e concludiamo che il periodo di h_0 (e analogamente anche di h_1 e h_2) divide $p^2 - p + 1$. Se inoltre una radice, ad esempio h_0, ha ordine minore o uguale a 3, allora sicuramente questo valore sarebbe 1 oppure 3, visto che $p^2 - p + 1$ è dispari. Poiché 3 divide $p^2 - 1$, e banalmente anche 1, possiamo concludere che $h_0 \in \mathbb{F}_{p^2}$. Ma allora anche h_1 e h_2, in quanto potenze di h_0, saranno elementi di $\mathbb{F}_{p^2}$. □

Lemma 3.7. *Valgono le seguenti proprietà di* c_n*, per* $n, u, v \in \mathbb{Z}$*:*

(i) $c_0 = 3$*;*

(ii) $c_1 = c$*;*

(iii) $c_{-n} = c_{np} = c_n^p$*;*

(iv) $c_n \in \mathbb{F}_{p^2}$*;*

(v) $c_{u+v} = c_u * c_v - c_v^p * c_{u-v} + c_{u-2v}$.

Dimostrazione. I punti (i) e (ii) seguono immediatamente dalla definizione di c_n. Riprendendo i tre casi del teorema precedente e notando

$$c_{np} = h_0^{np} + h_1^{np} + h_2^{np} = (h_0^n + h_1^n + h_2^n)^p = c_n^p$$

il punto (iii) è dimostrato. Sempre basandoci sulle considerazioni del teorema precedente, se tutti i termini h_j appartengono a $\mathbb{F}_{p^2}$ il punto (iv) è banalmente dimostrato; nel caso complementare si ha $c_n = T(h_0^n) \in \mathbb{F}_{p^2}$ per quanto detto in precedenza. L'ultimo punto è facilmente verificato eseguendo i calcoli. □

Il prossimo teorema è il nodo cruciale su cui si basa la teoria del cifrario XTR (cfr. sezione 2.3 di [37]).

Teorema 3.8. *Vale l'uguaglianza*

$$F(c_n, h_j^n) = 0$$

per $j = 0, 1, 2$ *e per* $n \in \mathbb{Z}$*. Inoltre sono equivalenti le seguenti proprietà:*

(i) $F(c, X)$ *è riducibile su* $\mathbb{F}_{p^2}$*;*

(ii) le radici di $F(c, X)$ *sono tutte in* $G_{p,6}$*;*

(iii) $c_{p+1} \in \mathbb{F}_p$.

Dimostrazione. Per provare che $F(c_n, h_j^n) = 0$ basta calcolare manualmente i coefficienti del polinomio che azzera i termini h_j^n

$$(X - h_0^n)(X - h_1^n)(X - h_2^n)$$

e verificare che sono uguali a quelli di $F(c_n, X)$.

Riguardo alla seconda parte del teorema, si noti che l'equivalenza di (i) e (ii) è già stata dimostrata nel Teorema 3.6. Procediamo ora dimostrando che (i) e (iii) sono equivalenti. Se il polinomio $F(c, x)$ è riducibile, allora $h_j \in \mathbb{F}_{p^2}$ per $j = 0, 1, 2$. Segue che

$$(h_j^{p+1})^p = h_j^{p^2+p} = h_j^{p+1} \quad \Rightarrow \quad h_j^{p+1} \in \mathbb{F}_p$$

e dunque anche la loro somma, ovvero c_{p+1}, sta in $\mathbb{F}_p$. Viceversa, se $c_{p+1} \in \mathbb{F}_p$, allora p è l'ordine di c_{p+1} e vale

$$F(c_{p+1}, X) = X^3 - c_{p+1}X^2 + c_{p+1}^p X - 1 = X^3 - c_{p+1}X^2 + c_{p+1}X - 1$$

per cui 1 è radice di $F(c_{p+1}, X)$. Per la prima parte del Teorema, le radici di $F(c_{p+1}, X)$ sono potenze $(p+1)$-esime delle radici di $F(c, X)$, quindi $F(c, X)$ ha una radice di periodo che divide $p+1$ e, più in particolare, $p^2 - 1$, risultando conseguentemente riducibile in $\mathbb{F}_{p^2}$. □

Il caso di nostro interesse è chiaramente quello in cui c è la traccia $T(g)$. In particolare i risultati precedenti ci permettono di trovare g e i suoi coniugati avendo solamente a disposizione $T(g)$. Inoltre la proprietà non vale solamente per g, ma per tutte le sue potenze g^n, per cui lui stesso e i suoi coniugati sono completamente determinati da $T(g^n)$. In pratica, rinunciando alla distinzione tra g e i suoi coniugati, possiamo avere una rappresentazione compatta di g stesso e di tutte le sue potenze n-esime utilizzato $T(g^n)$ al posto di g^n, rappresentazione che risulta 3 volte più compatta di quella dell'elemento g^n.

3.2 Algoritmi di calcolo

Dopo aver descritto l'algebra che sta dietro al sistema XTR, vediamo come svolgere effettivamente i calcoli necessari alla sua implementazione.

3.2.1 Calcolo dei c_n

Procediamo innanzitutto con il calcolare i coefficienti c_n a partire da c ed n. Enunciamo un risultato comodo per l'Algoritmo 1, la cui dimostrazione discende direttamente dal Lemma 3.7.

Proposizione 3.9. *Siano dati c e $S_n(c)$.*

(i) Calcolare $c_{2n} = c_n^2 - 2c_n^p$ richiede due moltiplicazioni in $\mathbb{F}_p$.

(ii) Calcolare $c_{n+2} = c * c_{n+1} - c^p * c_n + c_{n-1}$ richiede quattro moltiplicazioni in $\mathbb{F}_p$.

(iii) Calcolare $c_{2n-1} = c_n * c_{n-1} - c^p * c_n^p + c_{n+1}^p$ richiede quattro moltiplicazioni in $\mathbb{F}_p$.

(iv) Calcolare $c_{2n+1} = c_n * c_{n+1} - c * c_n^p + c_{n-1}^p$ richiede quattro moltiplicazioni in $\mathbb{F}_p$.

In futuro utilizzeremo spesso i coefficienti c_{n-1}, c_n, c_{n+1} contemporaneamente, per cui è utile introdurre una notazione comoda.

Definizione 3.10. *Definiamo il vettore $S_n(c) = (c_{n-1}, c_n, c_{n+1}) \in (\mathbb{F}_{p^2})^3$.*

Descriviamo ora l'algoritmo proposto da Lenstra e Verheul per il calcolo esplicito di $S_n(c)$ dati c ed n.

Algoritmo 1. *Siano dati dunque c ed n. Analizziamo separatamente i casi possibili.*

(i) Se $n < 0$ si può applicare l'algoritmo a $-n$ e, successivamente, applicare il punto (iii) del Lemma 3.7.

(ii) Se $n = 0$, allora si conclude immediatamente $S_n(c) = (c^p, 3, c)$ (basta svolgere i calcoli e considerare la parte (iv) del Lemma 3.7).

(iii) Se $n = 1$, allora per la Proposizione 3.9 si ha $S_1(c) = (3, c, c^2 - 2c^p)$.

(iv) Se $n = 2$, sempre grazie alla Proposizione 3.9 posso calcolare c_3 che, insieme con $S_1(c)$, mi fornisce $S_2(c)$.

(v) *Se* $n > 2$ *poniamo* $m = n$. *Se* m *è pari allora sostituiamo* m *con* $m - 1$. *Si indichi con* $\overline{S}_t(c) = S_{2t+1}(c), t \in \mathbb{Z}$ *e si ponga* $k = 1$. *Utilizzando sempre la proposizione precedente e il vettore* $S_2(c)$ *già calcolato, possiamo trovare* $\overline{S}_k(c) = S_3(c)$. *Scriviamo ora* $(m-1)/2 = 1m_{r-1}m_{r-2}\dots m_1m_0$ *espresso in forma binaria. Per ogni* $j = r-1, r-2, \dots, 0$ *in successione si procede in questo modo:*

- *se* $m_j = 0$ *allora usiamo* $\overline{S}_k(c)$ *per calcolare* $\overline{S}_{2k}(c) = (c_{4k}, c_{4k+1}, c_{4k+2})$ *utilizzando i risultati computazionali ottenuti dal Lemma 3.7 e dalla Proposizione 3.9. Si sostituisce poi* k *con* $2k$.
- *se* $m_j = 1$ *allora usiamo* $\overline{S}_k(c)$ *per calcolare* $\overline{S}_{2k+1}(c) = (c_{4k+2}, c_{4k+3}, c_{4k+4})$. *Si sostituisce poi* k *con* $2k + 1$.

Alla fine delle iterazioni avremo $k = m$ *e, di conseguenza,* $S_m(c) = \overline{S}_k(c)$. *Se* n *è pari usiamo* $S_m(c)$ *e calcoliamo* $S_{m+1}(c)$ *(usando la Proposizione 3.9 per il mancante* c_{m+2}*), dopo di che rimpiazziamo* m *con* $m + 1$. *Abbiamo così ottenuto* $S_n(c) = S_m(c)$.

I due casi del punto (v) dell'algoritmo appena enunciato si differenziano nell'applicazione dei due punti (iii) e (iv) della Proposizione 3.9. I due calcoli, oltre che essere computazionalmente equivalenti, sono anche matematicamente molto simili. Tale particolarità, insolita per routine di questo tipo, rende l'algoritmo meno suscettibile agli attacchi volti a determinare n sulla base dei tempi di funzionamento dell'algoritmo.

Riassumiamo gli oneri computazionali dell'Algoritmo 1 nel seguente teorema.

Teorema 3.11. *Data* c*, la somma delle radici di* $F(c, X)$*, per il calcolo della somma delle potenze* n*-esime delle sue radici* c_n *occorrono* $8\log_2(n)$ *moltiplicazioni in* $\mathbb{F}_p$.

3.2.2 Ricerca di p e q

Anche la ricerca di p e q con le caratteristiche enunciate in precedenza deve essere fatta con criterio in modo che sia computazionalmente conveniente. Sempre basandoci sugli studi di Lenstra e Verheul, descriviamo un algoritmo che fa al caso nostro, ricordando che:

- p deve essere un numero primo;
- q deve essere primo e deve dividere $p^2 - p + 1$;
- $p \equiv 2 \bmod 3$.

Un algoritmo che fa questo potrebbe essere molto semplice: si sceglie $p \equiv 2 \bmod 3$ tale che $p^2 - p + 1$ sia del tipo qs con q primo e s sufficientemente piccolo (oppure, direttamente, q sufficientemente grande). Tuttavia algoritmi di questo tipo sono molto onerosi e conviene quindi fare ipotesi aggiuntive sul numero di partenza. Inoltre è più veloce iniziare la ricerca partendo da q.

Algoritmo 2. *Scegliamo innanzi tutto un numero primo* q *grande quanto ci serve, tale però che sia* $q \equiv 7 \bmod 12$. *Successivamente, calcoliamo le due radici* r_1 *e* r_2 *del polinomio* $X^2 - X + 1 \bmod q$. *Infine cerchiamo* $k \in \mathbb{Z}$ *tale che* $p = r_i + k * q$ *sia un primo congruo a* $2 \bmod 3$ *per* $i = 1$ *o* 2 *della grandezza richiesta.*

Il problema apparentemente più complesso è trovare un numero primo $q \equiv 7 \bmod 12$. Un metodo semplice ed efficace è partire da un qualsiasi numero (non primo) $a \equiv 7 \bmod 12$, per poi aggiungere 12 ad a finché $a + 12h$ non sia primo.

3.2.3 Trovare g

Come detto in precedenza, per trovare g non occorre scandagliare tutto il campo $\mathbb{F}_{p^6}$, ma è sufficiente trovare un $c = T(g)$ tale che g abbia ordine q. Una volta ottenuto c, una qualsiasi delle tre radici di $F(c, X)$ genera il sottogruppo $G_{p,6}$.

Una maniera semplice per trovare $c = T(g)$ utile ai nostri scopi è prendere un polinomio irriducibile su $\mathbb{F}_{p^2}$, usarlo per rappresentare $\mathbb{F}_{p^6}$, prelevare un elemento $h \in \mathbb{F}_{p^6}$ tale che $h^{(p^6-1)/q}$, chiamare $g = h^{(p^6-1)/q}$ (purché $g \neq 1$) e calcolare $T(g)$. Nonostante questo metodo sia apparentemente semplice, così non è da un punto di vista computazionale. Fortunatamente ci viene in aiuto il seguente lemma.

Lemma 3.12. *Dato $c \in \mathbb{F}_{p^2}$, la probabilità che $F(c, X)$ sia irriducibile è circa un terzo.*

Dimostrazione. Per dimostrare questo risultato è sufficiente "contare" manualmente i casi. Sappiamo che circa $p^2 - p$ elementi del sottogruppo $G_{p,6}$ sono radici del polinomio monico irriducibile della forma $F(c, X)$. Poiché ciascuno di essi ha tre radici distinte, ci sono circa $(p^2 - p)/3$ valori differenti di c in $\mathbb{F}_{p^2}/\mathbb{F}_p$ tale che $F(c, X)$ sia irriducibile. □

Dal Teorema 3.6 segue che è sufficiente scegliere un $c \in \mathbb{F}_{p^2}$ finché il relativo $F(c, X)$ è irriducibile e finché $c_{(p^2-p+1)/q} \neq 3$ (si veda il Lemma 3.7). Abbiamo dunque trovato $T(g) = c_{(p^2-p+1)/q}$, traccia di un elemento g di ordine q, senza calcolare esplicitamente g.

Possiamo quindi riassumere quanto detto nel seguente algoritmo.

Algoritmo 3. *Selezione di $c = T(g)$.*

(i) *Si sceglie casualmente $c \in \mathbb{F}_{p^2}/\mathbb{F}_p$.*

(ii) *Si calcola c_{p+1} usando l'Algoritmo 1.*

(iii) *Se $c_{p+1} \in \mathbb{F}_p$ (si veda il Teorema 3.8), tornare al punto (i).*

(iv) *Si calcola $c_{(p^2-p+1)/q}$ usando l'Algoritmo 1.*

(v) *Se $c_{(p^2-p+1)/q} = 3$, tornare al punto (i).*

(vi) *Si pone $T(g) = c_{(p^2-p+1)/q}$.*

Teorema 3.13. *L'Algoritmo 3 riesce a trovare un elemento $c = T(g)$ valido in $3q/(q-1)$ applicazioni dell'Algoritmo 1 in cui $n = p + 1$ e $q/(q-1)$ applicazioni dello stesso algoritmo in cui $n = (p^2 - p + 1)/q$.*

Vediamo nella prossima sezione come è possibile applicare gli algoritmi appena descritti ai più comuni cifrari basati sul logaritmo discreto.

3.3 XTR e la crittografia

Vediamo ora come applicare gli algoritmi noti in ambito crittografico alla teoria appena esposta.

3.3.1 Diffie-Hellman

Per implementare attraverso la teoria di XTR il protocollo presentato da Diffie e Hellman, Alice e Bob si accordano su $p, q, T(g)$. Ricordiamo che questo scambio può avvenire anche attraverso un canale insicuro. Dopo questo primo passo, Alice e Bob devono seguire i seguenti passi.

(i) Alice sceglie un intero casuale $a \in \mathbb{Z}$ minore di $q - 2$, usa l'Algoritmo 1 per calcolare $S_a(T(g))$ e invia $T(g^a)$ a Bob.

(ii) Bob riceve $T(g^a)$ da Alice, sceglie un intero casuale $b \in \mathbb{Z}$ minore di $q-2$, usa anch'egli l'Algoritmo 1 per calcolare $S_b(T(g))$ e invia $T(g^b)$ ad Alice.

(iii) Alice riceve $T(g^b)$ e calcola, sempre utilizzando l'Algoritmo 1 $S_a(T(g^b))$ determinando così K basandosi su $T(g^{ab})$.

(iv) In modo analogo Bob calcola $S_b(T(g^a))$ ottenendo lo stesso valore K.

Come già visto in precedenza, in questo modo si riesce ad avere una sicurezza uguale a quella di Diffie-Hellman standard utilizzando solamente un terzo delle risorse.

3.3.2 ElGamal

Come per Diffie-Hellman, il destinatario del messaggio (Alice) rende disponibili pubblicamente p, q e $T(g)$, dopo di che sceglie un intero segreto k, calcola $S_k(T(g))$ e rende pubblico il risultato $T(g^k)$. Alice deve inoltre concordare un sistema simmetrico per codificare i messaggi (si veda oltre). Vediamo ora come Bob può inviare un messaggio ad Alice.

(i) Innanzi tutto Bob sceglie un intero $b \in \mathbb{Z}$ minore di $q - 2$ e calcola $S_b(T(g))$.

(ii) Bob successivamente calcola $S_b(T(g^k))$.

(iii) Bob sceglie una chiave simmetrica K basata sul valore $T(g^{kb})$ appena calcolato.

(iv) Bob utilizza il sistema simmetrico per cifrare il messaggio M nel messaggio cifrato E.

(v) Bob spedisce la coppia $(E, T(g^b))$ ad Alice.

Alice riceve $(E, T(g^b))$ da Bob e decifra il messaggio nel seguente modo.

(i) Alice calcola $S_k(T(g^b))$.

(ii) Alice ricava la chiave simmetrica K a partire da $T(G^{kb})$.

(iii) Alice utilizza la chiave K appena ottenuta per decifrare il messaggio E con il sistema simmetrico da lei inizialmente deciso.

Come per la situazione precedente, anche in questo caso il livello di sicurezza è uguale a quello dello stesso cifrario utilizzato basandosi sui sottogruppi dei gruppi moltiplicativi "standard". La spesa in termini computazionali è invece ridotta a un terzo.

Il sistema appena descritto è basato però su una versione ibrida di ElGamal unito ad un qualsiasi sistema a chiave simmetrica (ad esempio DES). Infatti XTR, per come è strutturato, permette solamente di elevare a potenza all'interno dell'operazione T, il che rende impossibile la banale moltiplicazione di due elementi. Questo è uno dei motivi per cui si sono cercate nuove soluzioni, tra cui i tori trattati nel prossimo capitolo.

3.4 Oltre XTR

Come per Lucas, anche XTR suggerisce delle idee che possono essere utili per estendere il cifrario, magari nel momento in cui l'estensione $\mathbb{F}_{p^6}/\mathbb{F}_{p^2}$ risultasse poco sicura per via di nuovi algoritmi che decifrassero il codice in tempo polinomiale (trasformandolo, di fatto, in un problema di tipo P).

Cosa succederebbe, ad esempio, se prendessimo il prossimo numero "bello" da un punto di vista del rapporto $\phi(n)/n$? Proseguendo per prodotti di primi successivi, il candidato successivo a 6 sarebbe 30, avente un rapporto di compressione pari a

$$\frac{\phi(30)}{30} = \frac{8}{30} = \frac{4}{15},$$

quindi migliore di $\phi(6)/6 = 1/3$.

Il problema, che già trovava posto come suggerimento nella parte finale di [37], viene inizialmente trattato in [7] nel 2002. In quest'ultimo lavoro gli autori Bosma, Hutton e Verheul presentano quattro congetture che avrebbero aperto nuove porte a XTR, di cui tuttavia due smentite nell'articolo stesso. A completare l'opera arrivano Rubin e Siverberg che in [51] dimostrano definitivamente la falsità delle quattro congetture fornendo dei controesempi, e contestualmente consegnando alla comunità matematica un nuovo cifrario basato sui tori. Quest'ultimo, grazie anche agli sforzi successivi, è immune a questo tipo di problemi e permette di applicare ElGamal nella sua versione originale.

4
I tori

I tori hanno fatto la loro comparsa nel mondo della crittografia molto recentemente con l'introduzione del cifrario CEILIDH da parte di Rubin e Silverberg nel 2003 (cfr. [51] e [52]). Come per LUC e XTR, anche CEILIDH utilizza come punto di forza il problema del logaritmo discreto, basandosi sul gruppo generato dai punti di un toro algebrico.

L'idea alla base di CEILIDH è molto simile a quella alla base di XTR, ovvero comprimere i dati utilizzando una funzione che ha come codominio un insieme più piccolo di quello di partenza, ma che permetta allo stesso tempo di garantire la sicurezza dell'insieme più grande.

Oltre a CEILIDH (che si riferisce solamente ai tori di dimensione 6), anche altri cifrari sono stati proposti, in particolare per i tori di dimensione superiore.

4.1 Definizioni

Iniziamo con alcune definizioni e notazioni fondamentali.

Innanzi tutto con la notazione $\mathfrak{g}$ indicheremo il generico gruppo algebrico moltiplicativo, ovvero dato un campo E la notazione $\mathfrak{g}(E)$ indicherà il gruppo moltiplicativo $E^\times$. L'introduzione di questa notazione è comoda per poter indicare con una notazione compatta e comoda un gruppo algebrico qualsiasi, che per le nostre applicazioni diventerà, se applicato a un campo, il gruppo moltiplicativo del campo.

Definizione 4.1. *Un* ***toro algebrico*** T *sul campo* $\mathbb{F}_q$ *è un gruppo algebrico definito su* $\mathbb{F}_q$*, tale che esiste un'estensione di campo in cui* T *è isomorfo a* $\mathfrak{g}^d$*, dove* d *è la dimensione di* T*. Se* T *è isomorfo a* $\mathfrak{g}^d$ *su* $\mathbb{F}_{q^n}$*, allora si dice che* $\mathbb{F}_{q^n}$ *separa* T*.*

4.1.1 La restrizione di Weil degli scalari

Sia E/F un'estensione di campo finita e separabile. Sia V una varietà definita sul campo E. La restrizione di Weil analizza il comportamento della varietà V quando viene considerata relativamente al campo F.

Definizione 4.2. *Sia V una varietà definita sul campo K e G un insieme finito. Scriveremo*

$$V^{|G|} := \bigoplus_{\sigma \in G} V,$$

dove $\oplus$ denota la somma diretta: $A \oplus B = \{(a, b) : a \in A, b \in B\}$.

Chiaramente $V^{|G|}$ è una varietà e ogni suo elemento x può essere visto come vettore di $|G|$ componenti in cui ciascuna di esse è un distinto elemento di V avente come indice un elemento di G.

Se G è un gruppo, questo modo di pensare gli elementi di $V^{|G|}$ ci permette di definire in maniera naturale un'azione di G su $V^{|G|}$. Se $g \in G$, possiamo definire $g(x)$, con $x \in V^{|G|}$, come il vettore di $|G|$ componenti in cui ogni indice delle sue componenti è stato preventivamente composto con g.

Enunciamo ora un teorema che definisce la restrizione di Weil e ne elenca alcune proprietà.

Teorema 4.3. *Sia E/F un'estensione di Galois e sia V una varietà affine definita su E. Allora esiste una varietà affine $\mathrm{Res}_{E/F}V$ definita su F tale che*

(i) esiste una biiezione tra i punti di $(\mathrm{Res}_{E/F}V)(F)$ e i punti di $V(E)$;

(ii) esistono delle proiezioni $\pi_\sigma : \mathrm{Res}_{E/F}V \to V$ tali che la somma diretta $\bigoplus_{\sigma \in G} \pi_\sigma$ sia isomorfa a V^G attraverso un isomorfismo definito su E;

(iii) esiste un'azione del gruppo G su $\mathrm{Res}_{E/F}V$ compatibile con l'isomorfismo del punto (ii) tale da rendere commutativo il seguente diagramma

$$\begin{array}{ccc} \mathrm{Res}_{E/F}V & \xrightarrow{\sim} & V^{|G|} \\ {\scriptstyle G}\downarrow & & \downarrow{\scriptstyle G} \\ \mathrm{Res}_{E/F}V & \xrightarrow[\sim]{} & V^{|G|} \end{array}$$

dove le frecce verticali sinistra e destra indicano, rispettivamente, l'azione naturale di G su $\mathrm{Res}_{E/F}$ punto a punto e l'azione naturale su $V^{|G|}$ sopra descritta.

La prima parte del teorema può essere chiarita pensando ad E come spazio vettoriale su F. Se ad esempio prendiamo $E = \mathbb{C}$, $F = \mathbb{R}$ e V la varietà composta da tutti i punti di $\mathbb{C}$, ovvero lo spazio affine complesso di dimensione 1, si vede chiaramente che $\mathrm{Res}_{E/F}V = \mathrm{Res}_{\mathbb{C}/\mathbb{R}}V$ è banalmente lo spazio affine reale di dimensione 2. La diversità di dimensione è ovviamente proporzionale al numero di elementi di una base dell'estensione.

Occorre invece più attenzione per descrivere la seconda e la terza parte del teorema. Esse descrivono un isomorfismo tra la restrizione di Weil e $|G|$ copie di V, dove G è il gruppo di Galois di E/F. Vediamo che $V^{|G|}$ ha una naturale interpretazione come varietà V più tutti i suoi coniugati ottenuti applicando di volta in volta a V gli elementi di G.

Per ogni elemento $\sigma \in G$ possiamo definire i **coniugati di Galois dei punti della varietà V sotto l'azione di σ** come

$$V^\sigma = \{\sigma(x) : x \in V\}.$$

Chiaramente ciascun V^σ è isomorfo a V, per cui

$$V^{|G|} = \bigoplus_{\sigma \in G} V \text{ è isomorfo a } \bigoplus_{\sigma \in G} V^\sigma.$$

Quindi il teorema 4.3 ci dice che la restrizione di Weil agli scalari $\mathrm{Res}_{E/F} V$ descrive il comportamento dei coniugati di Galois di V.

Riscrivendo la definizione di toro data all'inizio della sezione, possiamo definire il toro algebrico come $\mathrm{Res}_{\mathbb{F}_{q^n}/\mathbb{F}_q} \mathfrak{g}$. Il Teorema 4.3 fornisce quindi un isomorfismo

$$(\mathrm{Res}_{\mathbb{F}_{q^n}/\mathbb{F}_q} \mathfrak{g})(\mathbb{F}_q) \simeq \mathfrak{g}(\mathbb{F}_{q^n}) = \mathbb{F}_{q^n}^\times$$

e, riprendendo la funzione norma introdotta nella Definizioe 1.13, fornisce la mappa

$$\mathrm{Res}_{\mathbb{F}_{q^n}/\mathbb{F}_q} \mathfrak{g} \xrightarrow{N_{\mathbb{F}_{q^n}/\mathbb{F}_{q^m}}} \mathrm{Res}_{\mathbb{F}_{q^m}/\mathbb{F}_q} \mathfrak{g}$$

per ogni m divisore di n, ovvero per ogni campo $\mathbb{F}_{q^m}$ intermedio dell'estensione.

Possiamo finalmente dare la definizione di toro che utilizzeremo nel corso della trattazione.

Definizione 4.4. *Il* ***toro di norma 1 su*** $\mathbb{F}_q$ *è definito da*

$$T_n(\mathbb{F}_q) = \left\{ x \in \mathbb{F}_{q^n}^\times : N_{\mathbb{F}_{q^n}/\mathbb{F}_{q^d}}(x) = 1 \text{ per ogni } d < n, d|n \right\}.$$

L'oggetto appena definito è quindi un sottoinsieme del gruppo moltiplicativo $\mathbb{F}_{q^n}^\times$. La prossima proposizione fornisce ulteriori importanti caratteristiche di $T_n(\mathbb{F}_q)$.

Proposizione 4.5. *Sia* $T_n(\mathbb{F}_q)$ *come nella definizione precedente.*

(i) $T_n(\mathbb{F}_q)$ *è un sottogruppo di* $\mathbb{F}_{q^n}^\times$.

(ii) La dimensione di $T_n(\mathbb{F}_q)$ *è* $\phi(n)$.

(iii) $T_n(\mathbb{F}_q)$ *è isomorfo a* $G_{q,n}$.

(iv) Il numero di punti di $T_n(\mathbb{F}_q)$ *è* $\Phi_n(q)$.

(v) Se $h \in T_n(\mathbb{F}_q)$ *è un elemento di ordine primo che non divide* n*, allora* h *non giace in alcun sottocampo proprio di* $\mathbb{F}_{q^n}$.

Dimostrazione. I primi due punti sono immediati, date la definizione di $T_n(\mathbb{F}_q)$ e il fatto che la funzione norma è un omomorfismo moltiplicativo.

Il gruppo $\mathbb{F}_{q^n}^\times$ è ciclico di ordine $q^n - 1$ e il gruppo di Galois G dell'estensione $\mathbb{F}_{q^n}/\mathbb{F}_q$ è generato dall'automorfismo di Frobenius (si veda la Proposizione 1.7). Quindi, se t divide n, allora

$$N_{\mathbb{F}_{q^n}/\mathbb{F}_{q^t}}(x) = x^{(q^n-1)/(q^t-1)}$$

e, per la Definizione 4.4,

$$T_n(\mathbb{F}_q) \simeq \left\{x \in \mathbb{F}_{q^n}^{\times} : x^c = 1\right\} \qquad \text{dove } c = \text{MCD}\left(\frac{q^n-1}{q^t-1} \text{ tale che } t|n \text{ e } t \neq n\right).$$

Poiché $q^t - 1 = \prod_{j|t} \Phi_j(q)$, possiamo concludere che $\Phi_n(q)$ divide c. Esistono quindi dei polinomi $a_t(u) \in \mathbb{Z}[u]$ tali che

$$\sum_{t|n, t\neq n} a_t(u)\frac{u^n-1}{u^t-1} = \Phi_n(u)$$

da cui si deduce che c divide $\Phi_n(q)$. Si veda [10] per la dimostrazione completa. Quindi $c = \Phi_n(q)$, e di conseguenza $T_n(\mathbb{F}_q) \simeq G_{q,n}$. I punti (iii) e (iv) sono così dimostrati.

Per il punto (v), poiché $q \nmid n$, si ha $\text{MCD}(X^n - 1, nX^{n-1}) = 1$ in $\mathbb{F}_q[X]$ e quindi $X^n - 1$ non ha radici ripetute nella chiusura algebrica di $\mathbb{F}_q$. Dall'uguaglianza

$$X^n - 1 = \prod_{e|n} \Phi_e(X)$$

e per il fatto che $\Phi_n(p) \equiv 0 \bmod q$ otteniamo $\Phi_e(p) \not\equiv 0 \bmod q$ per $e \mid n, e < n$. Ma per ogni divisore proprio d di n vale

$$X^d - 1 = \prod_{e|d} \Phi_e(X) \;\Bigg|\; \prod_{\substack{e|n \\ e<n}} \Phi_e(X)$$

e segue che $p^d - 1 \not\equiv 0 \bmod q$, da cui la tesi. □

In pratica con la proposizione precedente abbiamo dimostrato che i punti del toro algebrico $T_6(\mathbb{F}_q)$ formano lo stesso sottogruppo utilizzato nel cifrario XTR. Vedremo più avanti che CEILIDH utilizzerà una compressione simile alla traccia.

4.2 I tori razionali

Una classe particolare di tori sono quelli detti "razionali". La loro interessante proprietà aggiuntiva rispetto ai tori arbitrari è la possibilità di comprimere la rappresentazione dei suoi punti di un fattore $\phi(n)/n$, esattamente come già visto con la traccia.

Definizione 4.6. *Sia* T *un toro algebrico su* $\mathbb{F}_q$ *di dimensione* d. *Allora* T *è detto* ***razionale*** *se esiste una mappa birazionale* $\rho : T \longrightarrow \mathbb{A}^d$ *definita su* $\mathbb{F}_q$.

Le **mappe birazionali**, definite tra due varietà, mettono in relazione le applicazioni invece che i punti, come accade con le applicazioni standard. In particolare una mappa birazionale tra due varietà X e Y è classe di equivalenza delle coppie $\langle U, \varphi_U\rangle$, dove U è un intervallo aperto di X e $\varphi : U \longrightarrow Y$[1]è un'applicazione regolare dotata di inversa $\psi : Y \longrightarrow X$ tale che le due funzioni siano componibili e la loro composizione fornisca l'identità. Due coppie $\langle U, \varphi_U\rangle$ e $\langle V, \varphi_V\rangle$ sono equivalenti se le funzioni φ_U e φ_V coincidono in $U \cap V$. Se non esiste l'inversa ψ, la mappa è chiamata semplicemente **razionale**.

[1]Poiché in realtà la funzione φ è definita su un insieme denso di X, sarebbe più corretto utilizzare ad esempio una freccia tratteggiata al posto di quella usuale. Poiché questa teoria esula dagli scopi del testo, abuseremo un po' delle notazioni e utilizzeremo le frecce standard anche in riferimento alle mappe razionali.

Una proposizione molto nota fornisce una caratterizzazione della birazionalità tra due varietà.

Proposizione 4.7. *Le seguenti condizioni sono equivalenti:*

(i) X *e* Y *sono birazionali;*

(ii) $K(X) \simeq K(Y)$*, dove, data una varietà* V*,* $K(V)$ *è l'insieme delle mappe razionali* $\varphi : V \longrightarrow \mathbb{P}^1$*;*

(iii) esistono due sottisiemi aperti $U \subset X$ *e* $V \subset Y$ *tali che* $U \simeq V$*.*

Tornando ai tori, T è razionale se e solo se, immergendo T nello spazio affine $\mathbb{A}^t$, esistono due insiemi aperti $W \subset T$ e $U \subset \mathbb{A}^d$ e delle mappe razionali $\rho_1 \dots \rho_d \in \mathbb{F}_q(x_1 \dots x_t)$ e $\varphi_1 \dots \varphi_t \in \mathbb{F}_q(y_1 \dots y_d)$ tali che $\rho = (\rho_1 \dots \rho_d) : W \longrightarrow U$ e $\varphi = (\varphi_1 \dots \varphi_t) : U \longrightarrow W$ sono isomorfismi tali che uno è inverso dell'altro. ρ è detta **parametrizzazione razionale** di T.

Dalla definizione risulta quindi evidente come per i tori razionali si possa avere una rappresentazione compatta dei suoi elementi. In generale questa situazione è la migliore possibile, in quanto una varietà razionale di dimensione d ha approssimativamente q^d punti su $\mathbb{F}_q$, e quindi un punto di $T(\mathbb{F}_q)$ non può essere rappresentato con meno di d elementi.

La domanda che sorge spontanea a questo punto è: quando un toro è algebrico? Purtroppo una risposta precisa a questo quesito non esiste, ma esso fa parte di quel "calderone" di problemi irrisolti di cui la matematica ancora è ricca.

Congettura 1. *Il toro* T_n *è razionale.*

Questa congettura, proposto tra gli altri da Voskresenskii in [61], è stato dimostrata quando n è la potenza di un numero primo o il prodotto di due potenze di primi (eventualmente diversi tra loro). Per gli interessi crittografici, tuttavia, sarebbe opportuno che la congettura fosse vera quando n è un prodotto di numeri primi distinti, in modo da utilizzare i "famosi" prodotti dei primi numeri primi $2, 6, 30, \dots$ che hanno il miglior rapporto $\phi(n)/n$.

In realtà da questo punto di vista gli studi successivi hanno portato buone notizie. In [15] viene introdotta la nozione di toro stabilmente razionale, una versione debole di quello razionale, che però ha tutte le carte in regola per poter essere utilizzato in crittografia. In questa sua versione debole la Congettura 1 vale per ogni toro di interesse crittografico.

4.3 Il cifrario CEILIDH

Prima di introdurre il cifrario vero e proprio, vediamo come parametrizzare a dovere il toro T_6, quello utilizzato da Rubin e Silverberg in [51].

4.3.1 Parametrizzazione razionale di T_6

Come suggerito dalla Definizione 4.6, vediamo che il toro T_6 è birazionalmente isomorfo allo spazio affine $\mathbb{A}^d$. In particolare, $d = 2$, cioè potremo rappresentare gli elementi di $T_6(\mathbb{F}_q)$ come elementi di $\mathbb{A}^2(\mathbb{F}_q)$, con la stessa compressione data da XTR.

Il nostro scopo è quello di dimostrare il seguente teorema.

Teorema 4.8. $T_6(\mathbb{F}_q)$ *e* $\mathbb{A}^2(\mathbb{F}_q)$ *sono birazionalmente equivalenti.*

Dimostrazione. Cerchiamo due mappe ρ e φ, una inversa dell'altra, che relizzino l'equivalenza birazionale.

Siano $x \in \mathbb{F}_{q^2} - \mathbb{F}_q$ e $(\alpha_1, \alpha_2, \alpha_3)$ una base di $\mathbb{F}_{q^3}/\mathbb{F}_q$. Avremo che $\mathbb{F}_{q^2} = \mathbb{F}_q(x)$ e che l'insieme $(\alpha_1, \alpha_2, \alpha_3, x\alpha_1, x\alpha_2, x\alpha_3)$ è una base dell'estensione $\mathbb{F}_{q^6}/\mathbb{F}_q$. Chiamiamo G il suo gruppo di Galois. Sia infine σ un elemento di G di ordine 2. Definiamo la mappa $\varphi_0 : \mathbb{A}^3(\mathbb{F}_q) \hookrightarrow \mathbb{F}_{q^6}^\times$ come

$$\varphi_0(u_1, u_2, u_3) = \frac{\gamma + x}{\gamma + \sigma(x)}$$

dove $\gamma = u_1\alpha_1 + u_s\alpha_2 + u_3\alpha_3$. Quindi $N_{\mathbb{F}_{q^6}/\mathbb{F}_{q^3}}(\varphi_0(\mathbf{u})) = 1$ per ogni $\mathbf{u} = (u_1, u_2, u_3)$. Definiamo ora $U = \left\{\mathbf{u} \in \mathbb{A}^3 : N_{\mathbb{F}_{q^6}/\mathbb{F}_{q^2}}(\varphi_0(\mathbf{u})) = 1\right\}$. Per la definizione di toro algebrico, $\varphi_o(\mathbf{u}) \in T_6(\mathbb{F}_q)$ se e solo se $u \in U$, per cui la restrizione di φ_0 a U fornisce un morfismo $\varphi_0 : U \longrightarrow T_6(\mathbb{F}_q)$. Segue dal Teorema 90 di Hilbert che esiste un isomorfismo

$$\varphi_0 : U \xrightarrow{\sim} T_6(\mathbb{F}_q) \smallsetminus \{1\}.$$

Quello che occorre ora è definire una mappa da U a $\mathbb{A}^2$. Per come è stata definita, U è un'ipersuperficie in $\mathbb{A}^3$ definito da un'equazione quadratica nelle variabili u_1, u_2, u_3. Fissato un punto $\mathbf{a} = (a_1, a_2, a_3)$, possiamo supporre senza perdere in generalità, cambiando la base $(\alpha_1, \alpha_2, \alpha_3)$, che il piano tangente in $\mathbf{a}$ alla superficie U sia quello definito dall'equazione $u_1 = a_1$. Se (v_1, v_2) è un punto di $\mathbb{F}_q^2$, allora l'intersezione di U con la retta $\mathbf{a} + t(1, v_1, v_2)$ consiste di due punti, che in particolare sono

$$\mathbf{a} \quad \text{e un secondo punto } \mathbf{b} \text{ della forma} \quad \mathbf{b} = \mathbf{a} + \frac{1}{f(v_1, v_2)}(1, v_1, v_2)$$

dove $f(v_1, v_2) \in \mathbb{F}_q[v_1, v_2]$ è un polinomio calcolabile esplicitamente. La mappa che manda (v_1, v_2) in $\mathbf{b}$ è un isomorfismo

$$g : \mathbb{A}^2 \smallsetminus V(f) \xrightarrow{\sim} U \smallsetminus \{\mathbf{a}\}$$

dove $V(f)$ è la sottovarietà di $\mathbb{A}^2$ formata dai punti del tipo $f(v_1, v_2) = 0$. La composizione $\varphi_0 \circ g$ definisce un isomorfismo

$$\varphi : \mathbb{A}^2 \smallsetminus V(f) \xrightarrow{\sim} T_6(\mathbb{F}_q) \smallsetminus \{1, \varphi_0(\mathbf{a})\}.$$

Per trovare il suo inverso ρ, prendiamo un elemento β dell'insieme $T_6(\mathbb{F}_q) \smallsetminus \{1, \varphi_0(\mathbf{a})\}$. Quest'ultimo sarà della forma $\beta = \beta_1 + \beta_2 x$ con $\beta_1, \beta_2 \in \mathbb{F}_{q^3}$. Si verifica facilmente che $\beta_2 \neq 0$ e che

$$y = \frac{1 + \beta_1}{\beta_2} \quad \text{implica} \quad \frac{\gamma}{\sigma(\gamma)} = \beta.$$

Una scrittura esplicita per ρ è quindi

$$\rho(\beta) = \left(\frac{u_2 - a_2}{u_1 - a_1}, \frac{u_3 - a_3}{u_1 - a_1}\right)$$

dove u_1, u_2, u_3 sono i coefficienti di γ nella base $(\alpha_1, \alpha_2, \alpha_3)$. Svolgendo i conti, si verifica che la funzione ρ appena trovata è l'isomorfismo inverso

$$\rho : T_6(\mathbb{F}_q) \smallsetminus \{1, \varphi_0(\mathbf{a})\} \xrightarrow{\sim} \mathbb{A}^2 \smallsetminus V(f).$$

della funzione φ precedentemente definita. □

Ora che abbiamo dimostrato che il toro T_6 è razionale, possiamo passare all'applicazione vera e propria ai noti cifrari che basano la loro sicurezza sul logaritmo discreto.

4.3.2 Applicazione ai cifrari noti

Innanzi tutto, per la Proposizione 4.5.(v), utilizzare il toro $T_n(\mathbb{F}_q)$ ci permette di avere la stessa sicurezza di $\mathbb{F}_{q^n}^\times$. Tuttavia, se il toro è razionale, grazie alla compattezza della rappresentazione degli elementi di $T_n(\mathbb{F}_q)$, sono sufficienti solamente $\phi(n)$ punti di $\mathbb{F}_q$, anziché n.

Selezione dei parametri

Per scegliere dei parametri validi dobbiamo avere una potenza q di un numero primo e un intero n in modo che $T_n(\mathbb{F}_q)$ sia razionale (ad esempio $n = 6$). Per ottenere una sicurezza compatibile con quella di RSA e di ElGamal standard, $n \log q \approx 1024$. Inoltre $\Phi_n(q)$ deve avere un divisore primo ℓ "abbastanza grande". Per scegliere questi valori, possiamo rifarci agli algoritmi già utilizzati nel cifrario XTR.

Ora che abbiamo trovato $T_n(\mathbb{F}_q)$ calcoliamo $m = \phi(n)$ e fissiamo una mappa birazionale

$$\rho : T_n(\mathbb{F}_q) \longrightarrow \mathbb{F}_q^m,$$

che avrà una sua inversa φ. Scegliamo infine un elemento $\alpha \in T_n(\mathbb{F}_q)$ di ordine ℓ. Per trovarlo rapidamente si può provare ad elevare alla potenza $(q^n - 1)/\ell$ un elemento qualsiasi di $\mathbb{F}_{q^n}^\times$; in generale funzionerà. Per i protocolli noti che adesso andremo a descrivere, saranno pubblici n, q, ρ, φ, ℓ e α oppure $g = \rho(\alpha)$.

Lo scambio di chiavi di Diffie-Hellman

Per lo scambio di chiavi descritto nella Sezione 1.3.3, si procede nel seguente modo.

(i) Alice sceglie un intero casuale a minore di ℓ, calcola $P_A = \rho(\alpha^a) \in \mathbb{F}_q^m$ e lo spedisce a Bob.

(ii) Bob sceglie anch'egli un intero casuale b minore di ℓ, calcola $P_B = \rho(\alpha^b) \in \mathbb{F}_q^m$ e lo spedisce ad Alice.

(iii) Alice calcola $\rho(\varphi(P_B)^a) \in \mathbb{F}_q^m$.

(iv) Bob calcola $\rho(\varphi(P_A)^b) \in \mathbb{F}_q^m$.

Poiché ρ e φ sono l'una inversa dell'altra, abbiamo

$$\rho(\varphi(P_B)^a) = \rho(\alpha^{ab}) = \rho(\varphi(P_A)^b)$$

che verrà usata da Alice e Bob come chiave.

ElGamal

Come accennato nella Sezione 3.3.2, il cifrario XTR non poteva essere applicato all'algoritmo di ElGamal originale, ma se ne poteva solamente utilizzare una versione ibrida in cui ci si scambiava una chiave da utilizzare poi con un sistema simmetrico (ovvero a chiave privata). Basando il cifrario sui tori, invece, abbiamo a disposizione un gruppo a tutti gli effetti, per cui il cifrario è applicabile nella sua versione originale.

Ricordiamo che il sistema è asimmetrico, ovvero il mittente e il destinatario procedono in modo diverso nel contribuire alla sicurezza del sistema. Nell'esempio Alice sarà la destinataria del messaggio e renderà visibile a tutti la sua chiave pubblica, che poi Bob utilizzerà per cifrare il suo messaggio.

Questo è quello che deve fare Alice.

(i) Alice sceglie un intero a minore di ℓ come sua chiave privata.

(ii) Alice genera $P_A = \rho(\alpha^a)$, la sua chiave pubblica.

Questi sono invece i passaggi che dovrà eseguire Bob per cifrare il messaggio.

(i) Bob rappresenta il suo messaggio M come elemento del gruppo generato da α.

(ii) Bob sceglie un intero casuale k minore di ℓ e calcola $\gamma = \rho(\alpha^k)$ e $\delta = \rho(M\varphi(P_A)^k)$.

(iii) Bob spedisce (γ, δ) ad Alice.

Alice, alla ricezione della coppia (γ, δ), calcola $M = \varphi(\delta)\varphi(\gamma)^{-a}$ ottenendo così il messaggio.

4.3.3 XTR sotto una luce diversa

Vediamo ora quali sono i rapporti tra il cifrario XTR e CEILIDH, apparentemente molto simili, ma in realtà algebricamente diversi.

Il cifrario XTR, invece di considerare $g \in G_{q,6}$, considera la traccia $T(g)$, ovvero la somma dei suoi coniugati. In particolare, per ogni elemento $g \in G_{q,6}$, la traccia di g ci permette di identificare il polinomio caratteristico di g e dei suoi coniugati. Il problema è che non abbiamo alcun modo per distinguere se una particolare soluzione del polinomio è g o uno dei suoi due coniugati. Per meglio comprendere, definiamo

$$C_g = \left\{ g^\sigma : \sigma \in \mathrm{Gal}(\mathbb{F}_{q^6}/\mathbb{F}_{q^2}) \right\}$$

l'insieme dei coniugati di g.

Dato l'insieme $C = (c_1, c_2, c_3)$, definiamo $C^j = (c_1^j, c_2^j, c_3^j)$. Se $C = C_g$, allora banalmente $C^j = C_{g^j}$. La struttura di XTR, però, supera il problema di calcolare g^j dato g calcolando direttamente la terna C_{g^j} a partire da C_g senza distinguere tra i vari elementi di C_g. Avendo quindi due insiemi di coniugati $R = (r_1, r_2, r_3)$ e $S = (s_1, s_2, s_3)$ non possiamo moltiplicarli tra di loro per avere un nuovo insieme di coniugati, perché non possiamo sapere se stiamo trattando con $C_{r_1 s_1}$ oppure con $C_{r_1 s_2}$, che sono sostanzialmente differenti. XTR non ha quindi un algoritmo efficiente per moltiplicare gli elementi che

sia privo di ambiguità. Al contrario, con CEILIDH è possibile. La motivazione matematica di questa sostanziale differenza è insita nella varietà utilizzata dai due cifrari.

In CEILIDH l'informazione scambiata è la rappresentazione compatta di un elemento del gruppo $G_{q,6}$, per cui si può utilizzare la sua struttura di gruppo per eseguire moltiplicazioni. In XTR, al contrario, l'informazione scambiata è un elemento della varietà T_6/S_3. Poiché la moltiplicazione in T_6 non preserva le orbite di S_3, non è possibile definire una moltiplicazione sulle classi di equivalenza.

Quello che rende computazionalmente appetibile il cifrario XTR, è il fatto che il quoziente T_6/S_3 è razionale. La parametrizzazione razionale conseguente permette di immergere T_6/S_3 in $\mathbb{A}^2$, e quindi di avere una rappresentazione compatta dei suoi elementi.

Definendo $\mathrm{XTR}(q)$ l'insieme delle tracce usate in XTR, ovvero

$$\mathrm{XTR}(q) = \left\{ T_{\mathbb{F}_{q^6}/\mathbb{F}_{q^2}}(\alpha) : \alpha \in T_6(\mathbb{F}_q) \right\} \subset \mathbb{F}_{q^2}$$

vale il seguente teorema.

Teorema 4.9. *L'insieme* $\mathrm{XTR}(q)$ *può essere identificato naturalmente con l'immagine di* $T_6(\mathbb{F}_q)$ *nell'insieme* $(T_6/S_3)(\mathbb{F}_q)$. *Più precisamente, esiste un'inclusione birazionale*

$$T_6/S_3 \hookrightarrow \mathrm{Res}_{\mathbb{F}_{q^2}/\mathbb{F}_q}\mathbb{A}^1 \simeq \mathbb{A}^2$$

tale che $\mathrm{XTR}(q)$ *è l'immagine della composizione*

$$T_6(\mathbb{F}_q) \longrightarrow (T_6/S_3)(\mathbb{F}_q)(\mathrm{Res}_{\mathbb{F}_{q^2}/\mathbb{F}_q}\mathbb{A}^1)(\mathbb{F}_q) \simeq \mathbb{F}_{q^2}.$$

Andando ancora oltre, possiamo generalizzare ulteriormente, come descritto in [7] e [51].

Sia $n = de$ un numero intero che non sia un quadrato perfetto. Siano inoltre $E = \mathbb{F}_{q^n}$, $F = \mathbb{F}_{q^d}$ e $k = \mathbb{F}_q$.

Definizione 4.10. *Denotiamo con* $B_{(d,e)}$ *l'immagine di* T_n *in* $(\mathrm{Res}_{E/k}\mathfrak{g})/S_e$.

Ricordiamo che $\mathfrak{g}$ è il generico gruppo moltiplicativo. L'oggetto $B_{(d,e)}$ appena definito è una varietà. Quando $e = 1$, la varietà $B_{(n,1)}$ è il toro T_n che abbiamo già visto essere un gruppo. Tuttavia non sempre è così. In particolare per XTR e LUC, che corrispondono rispettivamente a $(d, e) = (2, 3)$ e $(d, e) = (1, 2)$, le rispettive varietà $B_{(d,e)}$ non sono gruppi e quindi non possiedono una moltiplicazione che sia priva di ogni ambiguità.

4.4 I tori stabilmente razionali

La soluzione proposta da Rubin e Silverberg sui tori algebrici è molto allettante, ma si scontra con una congettura non ancora dimostrata: finché non avremo informazioni maggiori sui tori costruiti a partire da prodotti di primi distinti, non possiamo estendere la costruzione di CEILIDH.

Nel tentativo di dimostrare la congettura o, comunque, di aggirare il problema, Marten van Dijk e David Woodruff hanno scoperto che le idee espresse in [51] possono non essere limitate ai soli casi in cui il toro sia razionale, "rompendo" la fastidiosa limitazione per cui n potesse essere al massimo 6, senza perdere il risvolto pratico che permette di inviare solo $\phi(n)/n$ dei bit richiesti per i cifrari tradizionali.

Così viene l'idea di utilizzare, anziché i tori algebrici razionali, i **tori algebrici stabilmente razionali** (si veda [61] per approfondire l'argomento). Essi permettono di costruire una biiezione θ che faccia le veci della funzione ρ del cifrario CEILIDH.

Iniziamo con la definizione di una funzione nota utile per accorciare le notazioni che utilizzeremo nella nostra trattazione.

Definizione 4.11. *Dato un intero n, definiamo la **funzione di Möbius** μ in questo modo:*

- $\mu(n) = 1$ *se* $n = 1$*;*
- $\mu(n) = 0$ *se nella fattorizzazione di n compaiono dei fattori ripetuti;*
- $\mu(n) = (-1)^k$ *se n è il prodotto di k primi distinti.*

Enunciamo ora una proposizione utile che lega la funzione di Möbius e i polinomi ciclotomici.

Proposizione 4.12. *Vale la seguente relazione:*

$$\Phi_n(x) = \prod_{d|n} (x^d - 1)^{\mu(n/d)}.$$

Durante il resto della sezione, $q = p^n$ con p un numero primo e indicheremo con $[k]$ l'insieme dei primi k numeri naturali, ovvero $[k] = \{1, 2, \ldots, k\}$. L'obiettivo di questa sezione è quello di costruire una funzione biiettiva θ (con inversa θ^{-1}) definita come

$$\theta : T_n(\mathbb{F}_q) \times \left(\bigtimes_{\substack{d|n \\ \mu(n/d)=-1}} \mathbb{F}_{q^d}^{\times} \right) \longrightarrow \bigtimes_{\substack{d|n \\ \mu(n/d)=1}} \mathbb{F}_{q^d}^{\times}. \tag{4.1}$$

Per fare questo ci serviamo di cinque lemmi, di cui non riportiamo le dimostrazioni, presenti in [15]. Indichiamo con C_n il gruppo ciclico di ordine n.

Lemma 4.13. *Supponiamo $n = r_1 \cdot r_2 \cdot \cdots \cdot r_k$ con $r_1, \ldots, r_k$ primi tra loro a due a due. Allora esistono due isomorfismi*

$$\begin{aligned} \rho &: C_n \longrightarrow \bigtimes_{i \in [k]} C_{r_i} \\ \sigma &: \bigtimes_{i \in [k]} C_{r_i} \longrightarrow C_n \end{aligned}$$

facilmente implementabili con operazioni elementari.

Definiamo quindi un isomorfismo utile.

Lemma 4.14. *Sia* U *il più piccolo intero positivo per cui*

$$\mathrm{MCD}\left(\Phi_d(q), \Phi_e(q), \frac{q^n - 1}{U}\right) = 1$$

per ogni d, e *tali che* $d \neq e$, $d \mid n$ *ed* $e \mid n$. *Per* $d \mid n$, *sia*

$$y_d = \mathrm{MCD}\left(\Phi_d(q), \frac{q^n - 1}{U}\right).$$

Allora esiste un isomorfismo

$$\mathbb{F}_{q^n}^{\times} \simeq C_U \times \left(\bigtimes_{d \mid n} C_{y_d}\right)$$

facilmente implementabile con operazioni elementari.

Il prossimo lemma fornisce interessanti stime del lavoro-macchina necessario per calcolare le biiezioni parziali utilizzate per calcolare θ.

Lemma 4.15. *Sempre per* $d \mid n$, *sia* $z_d = \mathrm{MCD}(\Phi_d(q), U)$. *Allora esistono due biiezioni tra* C_U *e* $\times_{d \mid n} C_{z_d}$ *che richiedono* $O(\log U + \log n + \log\log q)$ *operazioni per essere calcolate e* $O(U n^{1+\varepsilon} \log q)$ *bit di spazio per* ε *piccolo a piacere.*

Il prossimo lemma ci fornisce un altro isomorfismo utile.

Lemma 4.16. *Siano* y_d *e* z_d *come nei lemmi precedenti. Allora esiste un isomorfismo*

$$\bigtimes_{d \mid n} T_d(\mathbb{F}_q) \simeq \left(\bigtimes_{d \mid n} C_{y_d}\right) \times \left(\bigtimes_{d \mid n} C_{z_d}\right)$$

facilmente implementabile con operazioni elementari.

Mettiamo insieme tutti i risultati ottenuti nel lemma seguente.

Lemma 4.17. *Assumendo che le applicazioni del Lemma 4.15 siano efficienti, esistono due mappe facilmente calcolabili* γ *e* γ^{-1}, *una inversa dell'altra, definite tra gli insiemi* $\mathbb{F}_{q^n}^{\times}$ *e* $\times_{d \mid n} T_d(\mathbb{F}_q)$.

Siamo ora pronti per dimostrare il teorema seguente.

Teorema 4.18. *Assumendo che le applicazioni del Lemma 4.15 siano efficienti, esistono due mappe facilmente calcolabili* θ *e* θ^{-1}, *una inversa dell'altra, definite come in (4.1).*

Dimostrazione. Il Lemma 4.17 fornisce una efficiente biiezione tra gli insiemi

$$T_n(\mathbb{F}_q) \times \left(\bigtimes_{\substack{d \mid n \\ \mu(n/d) = -1}} \mathbb{F}_{q^d}^{\times}\right) \qquad \text{e} \qquad T_n(\mathbb{F}_q) \times \left(\bigtimes_{\substack{d \mid n \\ \mu(n/d) = -1}} \left(\bigtimes_{e \mid d} T_e(\mathbb{F}_q)\right)\right)$$

e anche tra

$$\bigtimes_{\substack{d\mid n\\ \mu(n/d)=1}} \mathbb{F}_{q^d}^{\times} \qquad \text{e} \qquad \bigtimes_{\substack{d\mid n\\ \mu(n/d)=1}} \left(\bigtimes_{e\mid d} T_e(\mathbb{F}_q) \right).$$

Permutando le coordinate, la tesi segue facilmente se dimostriamo l'uguaglianza degli insiemi

$$\{n\} \cup \bigsqcup_{\substack{d\mid n\\ \mu(n/d)=-1}} \{e \text{ tale che } e \mid d\} \qquad \text{e} \qquad \bigsqcup_{\substack{d\mid n\\ \mu(n/d)=1}} \{e \text{ tale che } e \mid d\}.$$

Per fare questo, riprendiamo la Proposizione 4.12 e scriviamo, nell'anello $\mathbb{Q}[x]$, il polinomio

$$\Phi_n(x) \prod_{\mu(n/d)=-1} (x^d - 1) = \prod_{\mu(n/d)=1} (x^d - 1).$$

Se scomponiamo l'equazione appena scritta in polinomi irriducibili, otteniamo

$$\Phi_n(x) \prod_{\mu(n/d)=-1} \prod_{e\mid d} \Phi_e(x) = \prod_{\mu(n/d)=1} \prod_{e\mid d} \Phi_e(x)$$

e, poiché $\mathbb{Q}[x]$ è un dominio a fattorizzazione unica, i due polinomi devono per forza coincidere. Questo fornisce l'uguaglianza degli indici cercata. □

4.5 La nuova costruzione

Il metodo appena visto, proposto in [15], in pratica cerca di ovviare il problema della razionalità dei tori T_n con n prodotto di primi distinti utilizzando dei tori "stabilmente razionali", ovvero trasformando la biiezione ρ nella biiezione

$$\theta : T_n(\mathbb{F}_q) \times \mathbb{F}_q^m \longrightarrow \mathbb{F}_q^{m+\phi(n)}. \tag{4.2}$$

Tuttavia la biiezione θ indicata per il toro T_{30} può essere migliorata. Nell'articolo originale, infatti, la migliore rappresentazione della "stabile razionalità" del toro era

$$\theta : T_{30}(\mathbb{F}_q) \times \mathbb{F}_q^{32} \longrightarrow \mathbb{F}_q^{40},$$

ovvero la (4.2) con $m = 32$, mentre in [14] si è riusciti ad ottenere

$$\theta : T_{30}(\mathbb{F}_q) \times \mathbb{F}_q^{2} \longrightarrow \mathbb{F}_q^{8},$$

la stessa formula di (4.2) con $m = 2$.

Vediamo ora come si articola la nuova idea. Iniziamo con alcune proposizioni utili.

Proposizione 4.19. *Se p è un numero primo e a è un numero intero positivo non divisibile per p, allora*

$$\Phi_{ap}(x) = \Phi_a(x) = \Phi_a(x^p).$$

Il seguente risultato ricalca la proposizione precedente e, in effetti, si dimostra a partire da essa.

Teorema 4.20. *Siano* p *un numero primo,* q *una potenza di un primo e* a *un intero positivo. Se* qa *non è divisibile per* p *e* $\mathrm{MCD}(\Phi_{ap}(q), \Phi_a(q)) = 1$*, allora*

$$T_{ap}(\mathbb{F}_q) \times T_a(\mathbb{F}_q) \simeq (\mathrm{Res}_{\mathbb{F}_{q^p}/\mathbb{F}_q} T_a)(\mathbb{F}_q) \simeq T_a(\mathbb{F}_{q^p}).$$

Sempre basandoci sulla Proposizione 4.19, possiamo dedurre il seguente risultato.

Teorema 4.21. *Se* n *non è un quadrato e se* m *è un divisore di* n*, allora*

$$\Phi_n(x) \prod_{\substack{d \mid \frac{n}{m} \\ \mu(\frac{n}{md}) = -1}} \Phi_m(x^d) = \prod_{\substack{d \mid \frac{n}{m} \\ \mu(\frac{n}{md}) = 1}} \Phi_m(x^d).$$

Enunciamo infine un risultato la cui dimostrazione è simile a quella già data nel Teorema 4.18.

Teorema 4.22. *Se* n *non è un quadrato e se* m *è un divisore di* n*, allora esiste una biiezione*

$$T_n(\mathbb{F}_q) \times \left(\bigtimes_{\substack{d \mid \frac{n}{m} \\ \mu(\frac{n}{md}) = -1}} T_m(\mathbb{F}_{q^d}) \right) \longrightarrow \left(\bigtimes_{\substack{d \mid \frac{n}{m} \\ \mu(\frac{n}{md}) = 1}} T_m(\mathbb{F}_{q^d}) \right)$$

facilmente implementabile con operazioni elementari.

È proprio questo risultato che ci permette di ottimizzare la biiezione θ del Teorema 4.18. Se inoltre i tori indicati con $T_m(\mathbb{F}_{q^d})$ sono razionali (come nel caso $n = 6$) il teorema appena enunciato fornisce

$$T_n \times \mathbb{A}^{D(m,n)} \sim \mathbb{A}^{\phi(n)+D(m.n)}$$

dove

$$D(m.n) = \phi(m) \sum_{\substack{d \mid \frac{n}{m} \\ \mu(\frac{n}{md}) = -1}} d.$$

Chiaramente, più $D(m.n)$ è piccolo, migliore sarà l'applicazione pratica del cifrario. Data l'attuale conoscenza della razionalità dei tori, sappiamo che T_6 è razionale, quindi possiamo prendere $m = 6$, il che ci permette di scrivere le seguenti biiezioni

$$T_{30} \times \mathbb{A}^2 \sim \mathbb{A}^{10} \qquad \text{e} \qquad T_{210} \times \mathbb{A}^{24} \sim \mathbb{A}^{72}.$$

Concludiamo con il seguente teorema.

Teorema 4.23. *Se* $n = p_1 \cdots p_k$ *è un prodotto di* $k \geqslant 2$ *primi distinti, allora*

$$\Phi_n(x) \prod_{i=2}^{k-1} \Phi_{p_1 \cdots p_i}(x^{p_{i+2} \cdots p_k}) = \Phi_{p_1 p_2}(x^{p_3 \cdots p_k}).$$

Applicando il teorema al caso $n = 30$, otteniamo i passaggi finali per la biiezione del cifrario cercato:

$$T_{30}(\mathbb{F}_q) \times \left(\mathbb{A}^2(\mathbb{F}_q) \setminus V(f)\right) \xrightarrow{\theta_0} T_{30}(\mathbb{F}_q) \times T_6(\mathbb{F}_q) \xrightarrow{\sigma} T_6(\mathbb{F}_{q^5}) \xrightarrow{\theta_1} \mathbb{A}^2(\mathbb{F}_{q^5}) \setminus V(f_5)$$

dove $V(f_5)$ indica $V(f)$ sul campo $\mathbb{F}_{q^5}$. Le mappe θ_0 e θ_1 sono basate sulla compressione e sulla decompressione del cifrario CEILIDH, mentre σ è la nuova mappa di decompressione data da

$$\sigma(x, y) = xy.$$

La mappa inversa è

$$\sigma^{-1}(z) = (z^{1-\alpha\Phi_{30}(q)}, z^{\alpha\Phi_{30}(q)})$$

dove α è dato da $\alpha\Phi_{30}(q) + \beta\Phi_6(q) = 1$. Per la verifica che $\mathrm{MCD}(\Phi_{30}(q), \Phi_6(q)) = 1$ si vedano [14] e [15].

Ora abbiamo tutti gli strumenti per procedere ad un'applicazione pratica della crittografia basata sui tori.

5

Una implementazione

Dopo aver descritto il *background* teorico necessario per cifrare usando i tori, vedremo ora come nella pratica avvengono i passaggi di compressione, decompressione, ricerca dei primi necessari, e via dicendo.

5.1 La scelta del linguaggio

Per implementare il nostro sistema di cifratura basato sui tori abbiamo deciso di utilizzare il linguaggio Java di Sun Microsystems. Hanno contribuito alla scelta diversi fattori.

Sintassi Il linguaggio Java è un derivato del C++, quindi ha una sintassi nota e facilmente comprensibile anche ai meno esperti di programmazione.

Librerie Nel linguaggio sono presenti in modo nativo le classi `BigInteger` e `Random`, che permettono rispettivamente di lavorare con numeri interi arbitrariamente grandi e di gestire in modo efficiente l'estrazione casuale di numeri con certe caratteristiche.

Comunità Java ha alle sue spalle una enorme comunità di sviluppatori, la quale ci ha permesso di ottenere tramite internet alcune librerie aggiuntive che hanno notevolmente accorciato il lavoro su quei problemi non direttamente relazionati con l'argomento della tesi. In particolare abbiamo utilizzato la classe `ecm` di Dario Alejandro Alpern per fattorizzare numeri interi molto grandi con il metodo delle curve ellittiche (scaricabile dal sito web http://www.alpertron.com.ar/ECM.HTM) e la classe `Primes` di Marty Hall, una raccolta di funzioni utili per lavorare con numeri molto grandi (scaricabile dal sito http://www.coreservlets.com/).

Estendibilità Il codice che abbiamo realizzato per il toro T_{30} può essere esteso senza problemi anche a tori di dimensione superiore. È sufficiente aggiungere, all'interno delle opportune classi, gli algoritmi di moltiplicazione e inversione di elementi ad esempio di $\mathbb{F}_{q^7}/\mathbb{F}_q$ (si veda più avanti).

5.1.1 Il codice sorgente

Nell'Appendice sono presenti i sorgenti completi dei programmi utilizzati. Per richiamare una determinata porzione di codice, abbiamo utilizzato la convenzione `nomefile.riga`, per cui `Main.java.46` rappresenta il codice a partire dalla riga 46 del file sorgente `Main.java` (che, nel caso specifico, è vuota).

5.1.2 Alcune note

Java è un **linguaggio a oggetti**. Questo significa che ogni "cosa" utilizzata al suo interno è considerato un oggetto che ha sue proprietà, metodi, funzioni. Questo approccio ha permesso una miglior progettazione del codice, nonché la possibilità che questo fosse estendibile a tori di dimensione più elevata. Tuttavia da un punto di vista prettamente "estetico", gli oggetti risultano spesso poco comprensibili. Ci sembra quindi doveroso spendere qualche riga per meglio comprendere la filosofia della programmazione ad oggetti.

Nel corso del testo trattiamo spesso con elementi di un gruppo. Per questo motivo è stata creata una **classe** `Elemento` (il cui codice si trova in `Elemento.java`). Una classe è in pratica la visione astratta di un oggetto, ovvero l'insieme di regole alle quali il singolo oggetto si atterrà scrupolosamente.

Ogni classe/oggetto ha al suo interno i cosiddetti **metodi**, ovvero delle funzioni intrinseche dell'oggetto stesso. Ad esempio il nostro `Elemento` possiede il metodo `add` che somma se stesso ad un altro `Elemento` e restituisce, come risultato, ancora un `Elemento`. Possiamo scrivere, ad esempio,

```
sommadeidue = primoaddendo.add(secondoaddendo);
```

per indicare che l'oggetto `sommadeidue` (di tipo `Elemento`) prenderà il valore che in "matematichese" scriveremmo `primoaddendo + secondoaddendo`. Questa costruzione è molto pratica per poter eseguire più operazioni concatenate su un oggetto. Quello che in "matematichese" scriviamo come $y = \sigma(\rho(x))$ in termini di oggetti diventerebbe qualcosa del tipo

```
y = x.rho().sigma();
```

come si può vedere, per fare un esempio, in `Elemento.java.584`.

Fatte queste veloci premesse, procediamo con la presentazione dell'implementazione.

5.2 La ricerca di p, q ed ℓ

Il passo iniziale di quasi ogni cifrario moderno è quello della ricerca dei numeri primi *ad hoc*, da utilizzare poi nelle operazione di cifratura descritte nella Sezione 4.3.2.

Da un punto di vista pratico, i numeri vanno scelti secondo alcuni criteri che li rendono particolarmente "comodi" per i calcoli successivi. I dati forniti equivalgono a una richiesta di sicurezza di un cifrario RSA a 1024 bit, mentre tra parentesi sono indicate le richieste per raggiungere la sicurezza di un cifrario RSA a 2048 bit.

(i) p deve essere un numero primo di circa 30 (risp. 61) bit;

(ii) q deve essere un numero primo di circa 35 (risp. 64) bit;

(iii) $p \equiv 1 \bmod 30$;

(iv) $q \equiv 2 \bmod 9$;

(v) $q \equiv 7 \bmod 11$;

(vi) $q^{30} \equiv 1 \bmod p$;

(vii) $\Phi_{30}(q)/p$ deve avere un divisore ℓ primo di almeno 160 (risp. 200) bit.

Nonostante le richieste possano sembrare stringenti, un normale personal computer fornisce molto probabilmente una soluzione nel giro di qualche minuto (risp. qualche ora). Ricordiamo che la scelta di p, q ed ℓ è da fare una sola volta e, anche se forse ciò non sarebbe raccomandabile, una sola buona scelta di questi valori sarebbe teoricamente sufficiente per tutte le comunicazioni di tutte le persone del nostro pianeta.

Il programma `Main.java` ha lo scopo di scegliere questi tre valori. L'unico punto su cui vale la pena soffermarci è la richiesta (vi). Partendo da due valori di p e q completamente casuali, è molto improbabile soddisfare la richiesta $q^{30} \equiv 1 \bmod p$, per cui è utile ricorrere ad uno stratagemma, che sarà poi utilizzato per trovare un elemento g di ordine ℓ. In pratica costruiamo preventivamente un h *ad hoc* che abbia la proprietà richiesta (e solo quella), alla quale poi applichiamo tutti i controlli rimanenti, di gran lunga più probabili. Nel nostro caso, a partire da un intero i qualsiasi minore di p, si è scelto

$$h = i^{\frac{p-1}{30}} \bmod p$$

il quale, effettivamente, ha ordine 30 mod p.

q	ℓ
18310583243	726080625060352451636238930710804663087684915821
12653992523	104128902949142690335051226558343892852956983 11
25470840809	240335418005505940836254594845770167128520372303161
26194868003	457271684802163130635298608010149043584951755416 11
8246747153	684403247187897887404220460304861174587046712050321
11244210743	108324286371803537493120912359195898692788255 81

Tabella 5.1: Esempi di parametri trovati dall'algoritmo (q da circa 35 bit, ℓ da circa 160 bit)

q	ℓ
5609734962762914507	751437495348591117756961901038302520195820525890361006486531556751
3310773474020223251	932183137385588255428930252750250500245433903113481548361539 31
17640843363573723641	250854801604027863364553926517440938160066778759083163138243332561
4271421036414508841	127674759789242332759731558071990502562008422090425548990123597 1
11349293286843367049	509179124324659223741065122561681158168067814203012855466383276 1

Tabella 5.2: Esempi di parametri trovati dall'algoritmo (q da circa 64 bit, ℓ da circa 200 bit)

Per verificare che abbia un divisore di 160 (risp. 200) bit, il numero viene preventivamente fattorizzato, dopo di che si controlla l'esistenza del fattore di ordine 160 (risp. 200) bit. La fattorizzazione viene in ogni caso interrotta dopo una decina di secondi (risp. qualche ora), perché in questo caso significa che il numero è formato da due fattori molto grandi, ciascuno dei quali è quasi sicuramente più piccolo dei 160 (risp. 200) bit richiesti. Per fare i calcoli è stato usato un computer Apple XServe G5, 2 GHz, 2GB DDR RAM.

5.3 L'aritmetica in $\mathbb{F}_{q^{30}}^{\times}$

Il problema più grosso, da un punto di vista informatico, è stata la realizzazione delle operazioni in un campo molto grande (di $q^{30} - 1$ elementi) nel modo più efficiente possibile, ovvero facendo sì che il tempo di esecuzione non fosse improponibile per un utilizzo concreto.

Innanzi tutto dobbiamo scegliere una base per il campo, per cui trattare con i suoi elementi equivalga a riscriverli come vettori di 30 interi positivi minori di q. Poiché $30 = 2 \cdot 3 \cdot 5$, abbiamo preso tre basi delle tre estensioni, programmando di volta in volta le operazioni nelle diverse basi. Vediamo innanzi tutto le operazioni per cui l'esecuzione non dipende dall'estensione utilizzata, per poi passare in dettaglio a quelle per le quali, invece, cambia la modalità di esecuzione.

5.3.1 Le operazioni comuni a tutte le estensioni

Addizione e sottrazione

L'addizione e la sottrazione sono le operazioni più facili, in quanto prevedono la somma (o sottrazione) modulo q degli elementi componente per componente, come nel caso dei vettori tradizionali. Le rispettive funzioni si trovano in `Elemento.java.155` e in `Elemento.java.169`.

Moltiplicazione per una costante

Anche in questo caso l'operazione è estremamente semplice. Ogni componente del vettore viene moltiplicato per la costante, il tutto ovviamente modulo q. La funzione si trova in `Elemento.java.207`.

Elevamento a potenza

L'elevamento a potenza utilizza l'algoritmo *left-to-right*, per il quale il risultato si ottiene moltiplicando ed elevando al quadrato di volta in volta, a seconda del valore del bit dell'esponente (convertito preventivamente in base 2).

Dati i valori $a, b, \alpha = (\alpha_{n-1}\alpha_{n-2}\dots\alpha_1\alpha_0)_2$, vogliamo ottenere $s = a^e \bmod b$. Si procede come segue (i è il contatore dei bit e $\leftarrow$ è l'operatore di assegnazione).

(i) Si pongono $s = 1$ e $i = n - 1$.

(ii) $s \leftarrow s^2 \bmod b$.

(iii) Se $\alpha_i = 1$ allora $s \leftarrow s \cdot a \bmod b$.

(iv) $i \leftarrow i - 1$.

(v) Se $i \geqslant 0$, torna al punto (iii).

Questo algoritmo riduce notevolmente il numero di moltiplicazioni rispetto all'esecuzione di un ciclo in cui ad ogni iterazione si moltiplica a per il numero ottenuto all'iterazione precedente (modulo b). Si sarebbe potuto ottimizzare ulteriormente il calcolo dell'elevamento alla potenza 2 ma questo non avrebbe portato a una rilevante diminuzione del tempo totale di esecuzione dell'algoritmo.

5.3.2 Le operazioni in $\mathbb{F}_{q^{30}}/\mathbb{F}_{q^{15}}$

Gli elementi di questa estensione sono due vettori, ciascuno di 15 componenti di $\mathbb{F}_q$. La base utilizzata è $(1, x)$, dove x ha polinomio minimo $x^2 + x + 1$ irriducibile per la scelta $q \equiv 2 \bmod 3$. Vediamo come ottimizzare moltiplicazione e inversione.

Moltiplicazione

Siano $c = c_0 + c_1 x$ e $d = d_0 + d_1 x$. Abbiamo $cd = (c_0 d_0 - c_1 d_1) + (c_0 d_1 + c_1 d_0 - c_1 d_1)x$. Per minimizzare i calcoli, troviamo prima $t_{00} = c_0 d_0$, $t_{11} = c_1 d_1$ e $t_{01} = (c_0 + c_1)(d_0 + d_1)$. Quindi

$$cd = (t_{00} - t_{11}) + (t_{01} - t_{00} - 2t_{11})x.$$

Si veda `Elemento.java.392`.

Inversione

A partire dall'elemento $c = c_0 + c_1 x$, sia $d = d_0 + d_1 x = (c_0 + c_1 x)^{-1}$. L'approccio diretto ci porta alla formula

$$d_0 + d_1 x = \frac{1}{c_0^2 - c_0 c_1 + c_1^2} \begin{pmatrix} c_0 - c_1 \\ -c_1 \end{pmatrix}.$$

Possiamo calcolare $t_0 = c_1 - c_0$, $t_{01} = c_0 c_1$ e $\Delta = t_0^2 + t_{01}$ e concludere

$$d = -\Delta^{-1} t_0 - \Delta^{-1} c_1 x.$$

Si veda `Elemento.java.281`.

5.3.3 Le operazioni in $\mathbb{F}_{q^{15}}/\mathbb{F}_{q^5}$

Gli elementi di questa estensione sono tre vettori, ciascuno di 5 componenti di $\mathbb{F}_q$. La base utilizzata è $(1, y, y^2 - 2)$, dove y ha polinomio minimo $y^3 - y + 1$.

Moltiplicazione

Siano $a = a_0 + a_1 y + a_2(y^2 - 2)$ e $b = b_0 + b_1 y + b_2(y^2 - 2)$. Allora si vede facilmente che

$$\begin{aligned} ab &= (a_0b_0 + 2a_1b_1 + 2a_2b_2 - a_1b_2 - a_2b_1) + (a_0b_1 + a_1b_0 + a_1b_2 + a_2b_1 - a_2b_2)y \\ &+ (a_0b_2 + a_2b_0 + a_1b_1 - a_2b_2)(y^2 - 2). \end{aligned}$$

Precalcolando $t_{00} = a_0b_0$, $t_{11} = a_1b_1$, $t_{22} = a_2b_2$, $t_{01} = (a_0 + a_1)(b_0 + b_1)$, $t_{12} = (a_1 - a_2)(b_2 - b_1)$ e $t_{20} = (a_2 - a_0)(b_0 - b_2)$, si ha

$$ab = (t_{00} + t_{11} + t_{22} - t_{12}) + (t_{01} + t_{12} - t_{00})y + (t_{20} + t_{00} + t_{11})(y^2 - 2).$$

Si veda `Elemento.java.361`.

Inversione

Sia $a = a_0 + a_1 y + a_2(y^2 - 2)$ e sia $b = b_0 + b_1 y + b_2(y^2 - 2) = (a_0 + a_1 y + a_2(y^2 - 2))^{-1}$. Allora

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix} = \frac{1}{\Delta} \cdot \begin{pmatrix} -a_0^2 + a_1^2 + a_2^2 - a_1a_2 \\ a_2^2 + a_0a_1 - 2a_1a_2 \\ -a_1^2 + a_2^2 + a_0a_2 \end{pmatrix}$$

dove $\Delta = -a_0^3 + a_1^3 + a_2^3 + 3a_0a_2^2 + 3a_0a_1^2 + 3a_1a_2^2 - 6a_1^2a_2 - 3a_0a_1a_2$. Ponendo $t_{ij} = a_ia_j$, $t_{012} = a_0 + a_1 + a_2$ e $t = t_{12}(a_0 + a_1)$, allora $\Delta = t_{012}^3 - t_{00}(3t_{012} - a_0) - 9t$. Per concludere l'inversione, si ha

$$\begin{aligned} b_0 &= \Delta^{-1}(t_{11} + t_{22} - t_{00} - t_{12}) \\ b_1 &= \Delta^{-1}(t_{01} - 2t_{12} + t_{22}) \\ b_2 &= \Delta^{-1}(t_{20} + t_{22} - t_{11}) \end{aligned}$$

Si veda `Elemento.java.250`.

5.3.4 Le operazioni in $\mathbb{F}_{q^5}/\mathbb{F}_q$

Gli elementi di questa estensione sono cinque vettori di una componente in $\mathbb{F}_q$. È stata scelta la base normale gaussiana $(t+t^{10}, t^7+t^4, t^5+t^6, t^2+t^9, t^3+t^8)$ della sottoestensione di grado 5 di $\mathbb{F}_q[t]/\Phi_{11}(t)$, che ha grado 10.

Moltiplicazione

Per la moltiplicazione abbiamo usato il tensore di ordine 3 visibile nel sorgente (in particolare nella posizione `Elemento.java.10`) che associa agli elementi di partenza $a = (a_1, a_2, a_3, a_4, a_5)$ e $b = (b_1, b_2, b_3, b_4, b_5)$ i coefficienti del prodotto $c = ab = (c_1, c_2, c_3, c_4, c_5)$. La parte di codice relativa si trova in `Elemento.java.335`.

Inversione

Per l'inversione in questa estensione, vista la difficoltà di ottenere una rappresentazione "attraente" della formula, abbiamo utilizzato l'algoritmo di Itoh-Tsujii (si veda [27]). Per calcolare a^{-1} con $a \in \mathbb{F}_{q^5}$ si procede nel seguente modo.

(i) Si calcola $r = (q^m - 1)/(q - 1)$.

(ii) Si calcola a^{r-1}.

(iii) Si calcola $a^r = a \cdot a^{r-1}$.

(iv) Si inverte a^r, ottenendo $(a^r)^{-1}$.

(v) Si calcola infine $a^{-1} = (a^r)^{-1} \cdot a^{r-1}$.

Il vantaggio di questo algoritmo è l'inversione del punto (iv) che risulta facilmente eseguibile poiché $a^r \in \mathbb{F}_q$ per costruzione (si può ad esempio trovare $(a^r)^{-1}$ calcolando $(a^r)^{q-2}$ in $\mathbb{F}_q$). Si veda `Elemento.java.243`.

5.4 Compressione e decompressione

Dopo aver presentato l'aritmetica che sta dietro alle estensioni di campo del cifrario, presentiamo la vera novità, ovvero l'utilizzo dei tori. Come abbiamo già detto nel capitolo precedente, l'innovazione di questo cifrario è stata la scoperta che la compressione già nota per i tori algebrici razionali può essere ottenuta su tori di dimensioni maggiori attraverso il prodotto cartesiano con un toro razionale (nell'equazione sottostante, la funzione σ). Nel nostro caso abbiamo utilizzato per cifrare la successione di funzioni

$$T_{30}(\mathbb{F}_q) \times \left(\mathbb{A}^2(\mathbb{F}_q) \setminus V(f)\right) \xrightarrow{\theta_0} T_{30}(\mathbb{F}_q) \times T_6(\mathbb{F}_q) \xrightarrow{\sigma} T_6(\mathbb{F}_{q^5}) \xrightarrow{\theta_1} \mathbb{A}^2(\mathbb{F}_{q^5}) \setminus V(f_5).$$

Chi riceve il messaggio non dovrà fare altro che applicare le funzioni al contrario. È doveroso (ma computazionalmente irrilevante) aggiungere che questi isomorfismi sono veri a meno di alcuni punti (indicati da $V(f)$) che nel caso concreto si riducono a 2. Se anche dovesse capitare questa sfortunata evenienza, è sufficiente cambiare il coefficiente k (si veda oltre) per risolvere il problema.

Nella pratica viene applicata inizialmente la funzione di decompressione φ di CEILIDH, seguita dalla nuova funzione σ e infine la compressione ρ, ereditata da CEILIDH. Gli isomorfismi utilizzati nella nostra implementazione sono quelli suggeriti in [14] e [15].

Le funzioni ρ e φ, una inversa dell'altra, sono

$$\begin{aligned} \rho(\beta_1, \beta_2) &= \left(\frac{v_2}{v_1}, \frac{v_3}{v_1}\right) \\ \varphi(\alpha_1, \alpha_2) &= \frac{1 + \alpha_1 y + \alpha_2(y^2 - 2) + f(\alpha_1, \alpha_2)x}{1 + \alpha_1 y + \alpha_2(y^2 - 2) + f(\alpha_1, \alpha_2)x^2} \end{aligned}$$

dove i coefficienti ν_i sono definiti da $\nu_1 + \nu_2 y + \nu_3(y^2 - 2) = (1 + \beta_1)/\beta_2$, mentre f è definita da $f(\alpha_1, \alpha_2) = 1 - \alpha_1^2 - \alpha_2^2 + \alpha_1\alpha_2$. In realtà da un punto di vista computazionale la funzione φ non è espressa in modo comodo, poiché x^2 vale $-x - 1$ e quindi prevede un rimescolamento dei coefficienti delle due basi $(1, x)$ e $(1, y, y^2 - 2)$. Si veda `Elemento.java.418`. Queste due applicazioni vengono utilizzate due volte durante il lavoro di Alice e Bob, una volta su $\mathbb{A}^2(\mathbb{F}_q)$ e una volta su $\mathbb{A}^2(\mathbb{F}_{q^5})$. Grazie al codice estremamente flessibile, è stato sufficiente scriverlo una sola volta, in quanto la struttura ad oggetti di Java esegue poi le operazioni di addizione, sottrazione, moltiplicazione e inversione nel campo di appartenenza dell'elemento.

Per quanto riguarda la funzione σ, molto semplice, essa è definita da

$$\begin{aligned} \sigma(x, y) &= xy \\ \sigma^{-1}(z) &= (zw^{-1}, w) \end{aligned}$$

dove $w = z^{\alpha\Phi_{30}(q)}$. Il coefficiente α è descritto nel punto (i) della prossima sezione. Il codice si trova in `Elemento.java.480`.

5.5 Un riassunto utile

Vediamo nel dettaglio cosa devono fare Alice e Bob per inviare un messaggio attraverso le compressioni appena descritte applicate al cifrario di ElGamal. Nell'esempio Alice sarà il mittente e Bob il destinatario. Il codice per queste operazioni inizia in `Elemento.java.507`.

(i) Alice e Bob si mettono d'accordo su p, q, ℓ e g. Ricordiamo che questi valori possono essere unici in tutto il mondo. Dovrà essere calcolato un coefficiente α applicando l'algoritmo di Euclide a $\Phi_{30}(q)$ e $\Phi_6(q)$. Poiché il loro MCD è 1 (si veda [14]), esistono α e β tali che $\alpha\Phi_{30}(q) + \beta\Phi_6(q) = 1$. Anche questo calcolo potrà essere eseguito una volta soltanto.

(ii) Bob sceglie un intero $a < l$ come sua chiave privata. Questo numero è personale di Bob e non è necessario cambiarlo per i messaggi futuri.

(iii) Bob calcola $A = g^a$, che sarà la sua chiave pubblica.

(iv) Bob invia A ad Alice.

(v) Alice trasforma il messaggio M in un numero relativamente piccolo m (si veda oltre per comprendere la motivazione di questa scelta).

(vi) Alice sceglie un intero $k < l$ da utilizzare per questo messaggio. Il valore deve essere diverso per ciascun messaggio.

(vii) Alice calcola $d = g^k$ ed $e = g^m A^k$. In un cifrario di ElGamal tradizionale Alice invierebbe la coppia (d, e) a Bob.

(viii) Alice comprime d con la funzione ρ di CEILIDH, ottenendo un elemento $\rho(d)$ in $\mathbb{A}^2(\mathbb{F}_{q^5}) \simeq \mathbb{A}^{10}(\mathbb{F}_q)$.

(ix) Alice separa $\rho(d) = (a_1, \ldots, a_{10})$ in $d_1 = (a_1, a_2)$ e $d_2 = (a_3, \ldots, a_{10})$.

(x) Alice applica φ, inversa di ρ, a d_1, ottenendo (γ_1, γ_2), coppia di elementi di $\mathbb{F}_{q^3}$, corrispondente a un elemento di $\mathbb{F}_{q^6}$ in realtà appartenente a $T_6(\mathbb{F}_q)$.

(xi) Alice immerge (γ_1, γ_2) in $T_6(\mathbb{F}_{q^5})$.

(xii) Alice applica $\sigma \circ \rho$ alla coppia $(e, \gamma = (\gamma_1, \gamma_2))$, ottenendo un elemento $\Gamma = \rho(\sigma(e, \gamma))$.

(xiii) Alice invia la coppia (Γ, d_2) a Bob.

(xiv) Bob riceve (Γ, d_2) e ricalcola $(e, \gamma) = \varphi(\sigma^{-1}(\Gamma))$.

(xv) Bob applica ρ a γ e ottiene d_1.

(xvi) Bob calcola $d^{-a} \cdot e = g^{-ka} \cdot g^m \cdot g^{ka} = g^m$.

(xvii) Poiché m era stato scelto sufficientemente piccolo, Bob può facilmente calcolare il logaritmo discreto di g^m per ottenere m.

5.6 Tempi di esecuzione

Nonostante i nostri algoritmi non siano stati ottimizzati al massimo (ad esempio non è stata realizzato al meglio il calcolo dell'esponenziazione, cfr. [14]), i tempi sono risultati appetibili, perfettamente in linea con quanto ci si aspettava e soprattutto compatibili con i computer attualmente in circolazione.

Gli unici due algoritmi lenti sono quelli utilizzati per trovare p, q ed ℓ i quali tuttavia devono essere eseguiti solo una volta, eventualmente addirittura per tutte le comunicazioni che tutte le persone si scambiano. Come già scritto in precedenza, nel caso di sicurezza equivalente ad un RSA di 1024 bit (come gli attuali protocolli SSL utilizzati per la sicurezza in internet), l'algoritmo fornisce una coppia valida ogni due-tre minuti. Per avere la sicurezza di un RSA a 2048 bit, in qualche giorno di lavoro il computer ci ha fornito una manciata di terne valide, visibili nella tabella 5.2.

Passando alla fase successiva, risulta non velocissima la ricerca di un elemento g valido di ordine ℓ nel campo $\mathbb{F}_{30}^{\times}$. Nel nostro caso la sua ricerca ha impegnato il computer per $3,314$ secondi nel caso di sicurezza RSA a 1024 bit e $10,369$ secondi nel caso di sicurezza RSA a 2048 bit. Anche g, come p, q ed ℓ, potrebbe essere uno solo nel mondo.

Diventano decisamente più rapidi i tempi di cifratura e decifratura. Per cifrare un messaggio M di circa 30 bit sono occorsi $1,385$ secondi per la sicurezza minore e $2,965$ per quella maggiore. La decifrazione ha richiesto, rispettivamente, $1,764$ e $5,266$ secondi.

Come dato aggiuntivo diciamo che cifrare richiede 1356500 (risp. 1882250) moltiplicazioni in $\mathbb{F}_q$ e decifrare ne richiede 1695500 (risp. 3027000). Sarebbe quindi opportuno ottimizzare la moltiplicazione degli elementi di $\mathbb{F}_{q^5}$ per ridurre in modo rilevante i tempi di esecuzione.

5.7 Conclusioni e possibili estensioni

Gli algoritmi descritti sono realmente funzionanti e, anche se attualmente ancora un po' lenti per applicazioni pratiche, promettono bene per il futuro delle nostre comunicazioni. Il codice utilizzato consultabile nell'Appendice è estendibile senza problemi a tutti gli eventuali casi futuri di utilizzo di cifrari basati su tori (ad esempio il caso $n = 210$), nonché riutilizzabile per cifrari di ElGamal standard (è sufficiente ignorare le funzioni ρ, φ e σ).

Dal punto di vista di un'ottimizzazione del numero di calcoli effettuati, il nostro lavoro risulta incompleto anche alla luce delle nuove soluzioni proposte nel 2005 dagli autori di [14], che suggeriscono, sul fronte dell'esponenziazione, una variante del nostro algoritmo per cui l'esponente viene scritto in base q invece che nella semplice base 2. Andrebbe anche migliorato (ma nell'articolo non si fa cenno di una possibile soluzione) l'algoritmo di moltiplicazione di elementi di $\mathbb{F}_{q^5}$, attualmente basato sul tensore di ordine 3 in `Elemento.java.10`.

Ultimo miglioramento futuro, probabilmente il più importante, è la soluzione del problema del logaritmo discreto che Bob deve eseguire per ottenere m a partire da g^m. Allo stato attuale, il cifrario è utile e direttamente utilizzabile per la soluzione ibrida di ElGamal (si veda la sezione dedicata a XTR), ma decisamente impraticabile nel caso di un m molto grande (ovvero un messaggio molto lungo), se non a prezzo di utilizzare l'algoritmo più volte, allungando di conseguenza il tempo di invio, e quindi di fatto annullando il vantaggio ottenuto con la compressione dei dati grazie agli isomorfismi ρ, φ e σ.

A

I sorgenti completi dei programmi

A.1 Il file **Main.java**

```
1  package paccheggio;
2
3  import java.math.BigInteger;
4
5  public class Main {
6
7    final static int TORO = 30;
8
9    final static int pBIT_SIZE = 61;
10   final static int qBIT_SIZE = 64;
11   final static int TARGET_SIZE = 210;
12   final static int TARGET_SIZE_pm = 15;
13
14   final static int PASSI = 67;
15
16   final static BigInteger UNO = BigInteger.valueOf(1);
17   final static BigInteger DUE = BigInteger.valueOf(2);
18   final static BigInteger SETTE = BigInteger.valueOf(7);
19   final static BigInteger NOVE = BigInteger.valueOf(9);
20   final static BigInteger UNDICI = BigInteger.valueOf(11);
21   final static BigInteger TRENTA = BigInteger.valueOf(TORO);
22
23   final static int K = (int) Math.pow(2, qBIT_SIZE - pBIT_SIZE);
24
25   public static BigInteger dammiFattore(BigInteger numero) {
26     ecm fatt = new ecm();
27     fatt.onlyFactoring = true;
28     int num_fattori;
29     int[] esponenti = new int[100];
30     BigInteger[] fattori = new BigInteger[100];
31
32     num_fattori = fatt.getFactors(numero, fattori, esponenti);
```

```
33      if (num_fattori == 1) {
34        return BigInteger.ONE;
35      }
36      for (int i = 0; i < num_fattori; i++)
37        if (
38            fattori[i].bitLength() > TARGET_SIZE - TARGET_SIZE_pm
39            && fattori[i].bitLength() < TARGET_SIZE + TARGET_SIZE_pm
40            ) {
41          return fattori[i];
42        }
43
44      return BigInteger.ONE;
45    }
46
47    public static void main(String[] args) {
48      BigInteger p, q = BigInteger.ZERO, h;
49      long adesso = System.currentTimeMillis();
50      boolean trovato = false;
51
52      while (true) {
53        p = DUE;
54        while (!NumeroPrimo.restoSeLoDividoPer(p, TRENTA).equals(UNO)) {
55          p = NumeroPrimo.dammiUnPrimo(pBIT_SIZE);
56        }
57        long passo = p.longValue() / PASSI;
58        for (long i = 2; i < p.longValue(); i += passo) {
59          h = NumeroPrimo.elevaMod(BigInteger.valueOf(i),
60              p.subtract(UNO).divide(TRENTA), p);
61          for (int k = 1; k <= K; k++) {
62            q = h.add(p.multiply(BigInteger.valueOf(k)));
63            if (NumeroPrimo.restoSeLoDividoPer(q, NOVE).equals(DUE) &&
64                NumeroPrimo.restoSeLoDividoPer(q, UNDICI).equals(SETTE) &&
65                q.isProbablePrime(100)) {
66              trovato = true;
67              for (int a = 1; a < TORO; a++) {
68                if (TORO % a == 0) {
69                  trovato = trovato &&
70                    !q.modPow(BigInteger.valueOf(a), p).equals(UNO);
71                }
72              }
73              if (trovato) {
74                System.out.println("Forse ho trovato una coppia...");
75                System.out.println("p = " + p + " q = " + q);
76                BigInteger phi = NumeroPrimo.calcolaPhi30(q);
77                BigInteger phip = phi.divide(p);
78                BigInteger f = dammiFattore(phip);
79                if (!f.equals(BigInteger.ONE)) {
80                  System.out.println();
81                  System.out.println("p = " + p + " q = " + q);
82                  System.out.println("Phi30 = " + phi);
83                  System.out.println("Phi30/p = " + phip);
84                  System.out.println("l = " + f);
85                  System.out.println("Secondi trascorsi = " +
86                      ((System.currentTimeMillis() - adesso)/1000));
```

```
87                  adesso = System.currentTimeMillis();
88                }
89                else {
90                  System.out.println("E' andata male!");
91                }
92                System.out.println();
93              }
94              break;
95            }
96          }
97          if (trovato) {
98            break;
99          }
100       }
101     }
102   }
103 }
```

A.2 Il file **NumeroPrimo.java**

```
1  package paccheggio;
2
3  import java.math.BigInteger;
4  import java.util.Random;
5
6  public class NumeroPrimo {
7
8    public static BigInteger alfaBeta(BigInteger primo,
9      BigInteger secondo, int quale) {
10     BigInteger[] x = new BigInteger[3],
11       y = new BigInteger[3],
12       z = new BigInteger[3];
13     BigInteger a;
14
15     x[0] = BigInteger.ONE;
16     x[1] = BigInteger.ZERO;
17
18     x[2] = x[0].multiply(primo).add(x[1].multiply(secondo));
19
20     y[0] = BigInteger.ZERO;
21     y[1] = BigInteger.ONE;
22
23     y[2] = y[0].multiply(primo).add(y[1].multiply(secondo));
24
25     while (!y[2].equals(BigInteger.ZERO)) {
26       a = x[2].divide(y[2]);
27       for (int i = 0; i < 2; i++) {
28         z[i] = NumeroPrimo.copia(y[i]);
29         y[i] = NumeroPrimo.copia(x[i].subtract(a.multiply(y[i])));
30         x[i] = NumeroPrimo.copia(z[i]);
31       }
32       x[2] = x[0].multiply(primo).add(x[1].multiply(secondo));
```

```
33        y[2] = y[0].multiply(primo).add(y[1].multiply(secondo));
34      }
35
36      return NumeroPrimo.copia(x[quale]);
37    }
38
39    static BigInteger dammiUnPrimo(int dim) {
40      return BigInteger.probablePrime(dim,new Random());
41    }
42
43    static BigInteger copia(BigInteger numero) {
44      return numero.add(BigInteger.ZERO);
45    }
46
47    static BigInteger sommaPrimi(BigInteger primo, BigInteger secondo) {
48      return primo.add(secondo);
49    }
50
51    static BigInteger elevaMod(BigInteger primo,
52      BigInteger secondo, BigInteger terzo) {
53      return primo.modPow(secondo, terzo);
54    }
55
56    static BigInteger restoSeLoDividoPer(BigInteger primo, BigInteger secondo) {
57      return primo.remainder(secondo);
58    }
59
60    static BigInteger calcolaPhi30(BigInteger numero) {
61      return numero.pow(8)
62        .add(numero.pow(7))
63        .subtract(numero.pow(5))
64        .subtract(numero.pow(4))
65        .subtract(numero.pow(3))
66        .add(numero).add(BigInteger.valueOf(1));
67    }
68
69    static BigInteger calcolaPhi6(BigInteger numero) {
70      return numero.pow(2)
71        .subtract(numero).add(BigInteger.valueOf(1));
72    }
73
74  }
```

A.3 Il file **Elemento.java**

```
1  package paccheggio;
2
3  import java.math.BigInteger;
4  import java.util.Vector;
5
6  public class Elemento {
7    private static BigInteger q, qalfa, qbeta;
```

```
8   private static int Molt = 0;
9
10  private static int multi5vett[][][]= {
11    {
12      {-2, -2, -2, -1, -2},
13      {0, 0, 1, 0, 1},
14      {0, 1, 1, 0, 0},
15      {1, 0, 0, 0, 1},
16      {0, 1, 0, 1, 0}
17    },
18    {
19      {0, 0, 1, 0, 1},
20      {-2, -2, -2, -2, -1},
21      {1, 0, 0, 1, 0},
22      {0, 0, 1, 1, 0},
23      {1, 1, 0, 0, 0}
24    },
25    {
26      {0, 1, 1, 0, 0},
27      {1, 0, 0, 1, 0},
28      {-1, -2, -2, -2, -2},
29      {0, 1, 0, 0, 1},
30      {0, 0, 0, 1, 1}
31    },
32    {
33      {1, 0, 0, 0, 1},
34      {0, 0, 1, 1, 0},
35      {0, 1, 0, 0, 1},
36      {-2, -1, -2, -2, -2},
37      {1, 0, 1, 0, 0}
38    },
39    {
40      {0, 1, 0, 1, 0},
41      {1, 1, 0, 0, 0},
42      {0, 0, 0, 1, 1},
43      {1, 0, 1, 0, 0},
44      {-2, -2, -1, -2, -2}
45    }
46  };
47  private Vector valori;
48
49
50  public static void setQ(BigInteger numero) {
51    q = numero.add(BigInteger.ZERO);
52  }
53
54
55  public static BigInteger getQ() {
56    return q;
57  }
58
59
60  public static Elemento ZERO(int dimensione) {
61    Vector nuovo = new Vector();
```

```
62      for (int i = 0; i < dimensione; i++) {
63        nuovo.add(BigInteger.ZERO);
64      }
65      return new Elemento(nuovo);
66    }
67
68
69    public static Elemento UNO(int dimensione) {
70
71      Elemento nuovo = new Elemento(new Vector());
72
73      if (dimensione < 5) {
74        nuovo.addValue(BigInteger.ONE);
75        nuovo = nuovo.affianca(Elemento.ZERO(dimensione - 1));
76      }
77      else {
78        nuovo.addValue(BigInteger.ZERO.subtract(BigInteger.ONE));
79        nuovo.addValue(BigInteger.ZERO.subtract(BigInteger.ONE));
80        nuovo.addValue(BigInteger.ZERO.subtract(BigInteger.ONE));
81        nuovo.addValue(BigInteger.ZERO.subtract(BigInteger.ONE));
82        nuovo.addValue(BigInteger.ZERO.subtract(BigInteger.ONE));
83        nuovo = nuovo.affianca(Elemento.ZERO(dimensione - 5));
84      }
85      return nuovo;
86    }
87
88
89    public static Elemento aCaso(int dimensione) {
90      Elemento nuovo = new Elemento(new Vector());
91
92      for (int i = 0; i < dimensione; i++) {
93        nuovo.addValue(Primes.random(q.toString().length() - 1, false));
94      }
95      return nuovo;
96    }
97
98
99    public String toString() {
100     return this.valori.toString();
101   }
102
103
104   private void mettiValori(Vector valori) {
105     this.valori = new Vector();
106     int size = valori.size();
107
108     for (int i = 0; i < size; i++) {
109       this.valori.add(((BigInteger) valori.get(i)).mod(q));
110     }
111   }
112
113
114   public Elemento(BigInteger numero, Vector valori) {
115     Elemento.setQ(numero);
```

```
116     mettiValori(valori);
117   }
118
119
120   public Elemento(Vector valori) {
121     mettiValori(valori);
122   }
123
124
125   public int getSize() {
126     return this.valori.size();
127   }
128
129
130   public void addValue(BigInteger numero) {
131     this.valori.add(numero);
132   }
133
134
135   public void putValue(BigInteger numero, int posizione) {
136     Vector nuovo = new Vector();
137     for (int i = 0; i < this.getSize(); i++) {
138       if (i != posizione) {
139         nuovo.add(this.getValue(i));
140       }
141       else {
142         nuovo.add(numero);
143       }
144     }
145     this.valori.clear();
146     this.valori = (Vector) nuovo.clone();
147   }
148
149
150   public BigInteger getValue(int posizione) {
151     return (BigInteger) this.valori.get(posizione);
152   }
153
154
155   public Elemento add(Elemento numero) {
156     int size = numero.getSize();
157     Vector nuovo = new Vector();
158
159     for (int i = 0; i < size; i++) {
160       BigInteger primo = (BigInteger) this.valori.get(i);
161       BigInteger secondo = (BigInteger) numero.valori.get(i);
162       nuovo.add(primo.add(secondo).mod(q));
163     }
164
165     return new Elemento(nuovo);
166   }
167
168
169   public Elemento sub(Elemento numero) {
```

```
170     int size = numero.getSize();
171     Vector nuovo = new Vector();
172 
173     for (int i = 0; i < size; i++) {
174       BigInteger primo = (BigInteger) this.valori.get(i);
175       BigInteger secondo = (BigInteger) numero.valori.get(i);
176       nuovo.add(primo.subtract(secondo).mod(q));
177     }
178 
179     return new Elemento(nuovo);
180   }
181 
182 
183   public Elemento affianca(Elemento numero) {
184     int i;
185     Elemento nuovo = new Elemento(new Vector());
186     for (i = 0; i < this.getSize(); i++) {
187       nuovo.addValue(this.getValue(i));
188     }
189     for (i = 0; i < numero.getSize(); i++) {
190       nuovo.addValue(numero.getValue(i));
191     }
192     return nuovo;
193   }
194 
195 
196   public Elemento affianca(BigInteger numero) {
197     int i;
198     Elemento nuovo = new Elemento(new Vector());
199     for (i = 0; i < this.getSize(); i++) {
200       nuovo.addValue(this.getValue(i));
201     }
202     nuovo.addValue(numero);
203     return nuovo;
204   }
205 
206 
207   public Elemento mulCost(BigInteger costante) {
208     Vector nuovo = new Vector();
209 
210     for (int i = 0; i < this.getSize(); i++) {
211       BigInteger primo = (BigInteger) this.valori.get(i);
212       nuovo.add(primo.multiply(costante).mod(q));
213     }
214 
215     return new Elemento(nuovo);
216   }
217 
218 
219   public Elemento pow2() {
220     return this.mult(this);
221   }
222 
223 
```

```
224   public Elemento pow(BigInteger esponente) {
225     String numero = esponente.toString(2);
226     Elemento s = Elemento.UNO(this.getSize());
227     for (int i = 0; i < numero.length(); i++) {
228       s = s.pow2();
229       if (numero.charAt(i) == "1".charAt(0)) {
230         s = s.mult(this);
231       }
232     }
233     return s;
234   }
235
236
237   public Elemento inv() {
238     int i;
239     int dim = this.getSize();
240     switch (dim) {
241     case 1:
242       return this.pow(q.subtract(BigInteger.valueOf(2)));
243     case 5:
244       BigInteger r = q.pow(5).subtract(BigInteger.ONE)
245         .divide(q.subtract(BigInteger.ONE));
246       Elemento ar_1 = this.pow(r.subtract(BigInteger.ONE));
247       Elemento ar = this.mult(ar_1);
248       Elemento invar = ar.f_i().inv().f_i();
249       return invar.mult(ar_1);
250     case 15:
251     case 3:
252       Elemento a0 = new Elemento(new Vector());
253       Elemento a1 = new Elemento(new Vector());
254       Elemento a2 = new Elemento(new Vector());
255       for (i = 0; i < dim; i++) {
256         if (i < (dim / 3)) {
257           a0.addValue((BigInteger) this.valori.get(i));
258         }
259         else if (i < ((dim * 2) / 3)) {
260           a1.addValue((BigInteger) this.valori.get(i));
261         }
262         else {
263           a2.addValue((BigInteger) this.valori.get(i));
264         }
265       }
266       Elemento t00 = a0.pow2();
267       Elemento t11 = a1.pow2();
268       Elemento t22 = a2.pow2();
269       Elemento t01 = a0.mult(a1);
270       Elemento t12 = a1.mult(a2);
271       Elemento t20 = a2.mult(a0);
272       Elemento t012 = a0.add(a1).add(a2);
273       Elemento t = t12.mult(a0.add(a1));
274       Elemento det = t012.pow(BigInteger.valueOf(3))
275         .sub(t00.mult(t012.mulCost(BigInteger.valueOf(3)).sub(a0)))
276         .sub(t.mulCost(BigInteger.valueOf(9)));
277       det = det.inv();
```

```
278       return det.mult(t11.add(t22).sub(t00).sub(t12))
279         .affianca(det.mult(t01.sub(t12.mulCost(BigInteger.valueOf(2))).add(t22)))
280         .affianca(det.mult(t20.add(t22).sub(t11)));
281     case 30:
282     case 6:
283       Elemento c0 = new Elemento(new Vector());
284       Elemento c1 = new Elemento(new Vector());
285       for (i = 0; i < dim; i++) {
286         if (i < (dim / 2)) {
287           c0.addValue((BigInteger) this.valori.get(i));
288         }
289         else {
290           c1.addValue((BigInteger) this.valori.get(i));
291         }
292       }
293       Elemento t0 = c1.sub(c0);
294       Elemento t1 = c0.mult(c1);
295       Elemento delta = t0.pow2().add(t1);
296       delta = delta.inv().neg();
297       return delta.mult(t0).affianca(delta.mult(c1));
298     default:
299       return null;
300     }
301   }
302
303
304   public Elemento neg() {
305     return this.mulCost(BigInteger.valueOf(-1));
306   }
307
308
309   public Elemento f_i() {
310     if (this.getSize() == 5) {
311       Vector mnuovo = new Vector();
312       mnuovo.add(BigInteger.ZERO.subtract((BigInteger)this.valori.get(0)));
313       return new Elemento(mnuovo);
314     }
315     else {
316       return Elemento.UNO(5).mulCost(this.getValue(0));
317     }
318   }
319
320
321   public Elemento mult(Elemento numero) {
322     int dim = this.valori.size();
323     BigInteger[] primo, secondo, risultato;
324     Vector nuovo = new Vector();
325     int i = 0, j = 0, k = 0;
326
327     switch (dim) {
328     case 1:
329       primo = new BigInteger[1];
330       secondo = new BigInteger[1];
331       primo[0] = (BigInteger) this.valori.get(0);
```

```
332       secondo[0] = (BigInteger) numero.valori.get(0);
333       nuovo.add(primo[0].multiply(secondo[0]).mod(q));
334       return new Elemento(nuovo);
335     case 5:
336       primo = new BigInteger[5];
337       secondo = new BigInteger[5];
338       risultato = new BigInteger[5];
339
340       for (i = 0; i < 5; i++) {
341         primo[i] = (BigInteger) this.valori.get(i);
342         secondo[i] = (BigInteger) numero.valori.get(i);
343         risultato[i] = BigInteger.ZERO;
344       }
345       for (i = 0; i < 5; i++) {
346         for (j = 0; j < 5; j++) {
347           for (k = 0; k < 5; k++) {
348             risultato[k] = risultato[k].add(
349                 primo[i].multiply(secondo[j]).multiply(
350                     BigInteger.valueOf(multi5vett[i][j][k])
351                     )
352                 ).mod(q);
353             Molt++;
354           }
355         }
356       }
357       for (i = 0; i < 5; i++) {
358         nuovo.add(risultato[i]);
359       }
360       return new Elemento(nuovo);
361     case 15:
362     case 3:
363       Elemento a0 = new Elemento(new Vector());
364       Elemento a1 = new Elemento(new Vector());
365       Elemento a2 = new Elemento(new Vector());
366       Elemento b0 = new Elemento(new Vector());
367       Elemento b1 = new Elemento(new Vector());
368       Elemento b2 = new Elemento(new Vector());
369       for (i = 0; i < dim; i++) {
370         if (i < (dim / 3)) {
371           a0.addValue((BigInteger) this.valori.get(i));
372           b0.addValue((BigInteger) numero.valori.get(i));
373         }
374         else if (i < ((dim * 2) / 3)) {
375           a1.addValue((BigInteger) this.valori.get(i));
376           b1.addValue((BigInteger) numero.valori.get(i));
377         }
378         else {
379           a2.addValue((BigInteger) this.valori.get(i));
380           b2.addValue((BigInteger) numero.valori.get(i));
381         }
382       }
383       Elemento t00 = a0.mult(b0);
384       Elemento t11 = a1.mult(b1);
385       Elemento t22 = a2.mult(b2);
```

```
386       Elemento t01 = a0.add(a1).mult(b0.add(b1));
387       Elemento t12 = a1.sub(a2).mult(b2.sub(b1));
388       Elemento t20 = a2.sub(a0).mult(b0.sub(b2));
389       return t00.add(t11).add(t22).sub(t12)
390         .affianca(t01.add(t12).sub(t00))
391         .affianca(t20.add(t00).add(t11));
392     case 30:
393     case 6:
394       Elemento c0 = new Elemento(new Vector());
395       Elemento c1 = new Elemento(new Vector());
396       Elemento d0 = new Elemento(new Vector());
397       Elemento d1 = new Elemento(new Vector());
398       for (i = 0; i < dim; i++) {
399         if (i < (dim / 2)) {
400           c0.addValue((BigInteger) this.valori.get(i));
401           d0.addValue((BigInteger) numero.valori.get(i));
402         }
403         else {
404           c1.addValue((BigInteger) this.valori.get(i));
405           d1.addValue((BigInteger) numero.valori.get(i));
406         }
407       }
408       Elemento tp00 = c0.mult(d0);
409       Elemento tp11 = c1.mult(d1);
410       Elemento tp01 = c0.add(c1).mult(d0.add(d1));
411       return tp00.sub(tp11).affianca(tp01.sub(tp00).sub(tp11.add(tp11)));
412     default:
413       return null;
414     }
415   }
416
417
418   public Elemento rhoPhi() {
419     int i;
420     int dim = this.getSize();
421
422     switch (dim) {
423     case 6:
424     case 30:
425       // rho
426       Elemento b1 = new Elemento(new Vector());
427       Elemento b2 = new Elemento(new Vector());
428       for (i = 0; i < dim; i++) {
429         if (i < (dim / 2)) {
430           b1.addValue((BigInteger) this.valori.get(i));
431         }
432         else {
433           b2.addValue((BigInteger) this.valori.get(i));
434         }
435       }
436       Elemento b = b1.add(Elemento.UNO(dim / 2)).mult(b2.inv());
437       Elemento v1 = new Elemento(new Vector());
438       Elemento v2 = new Elemento(new Vector());
439       Elemento v3 = new Elemento(new Vector());
```

```
440       for (i = 0; i < (dim / 2); i++) {
441         if (i < (dim / 6)) {
442           v1.addValue(b.getValue(i));
443         }
444         else if (i < (dim / 3)) {
445           v2.addValue(b.getValue(i));
446         }
447         else {
448           v3.addValue(b.getValue(i));
449         }
450       }
451       return v2.mult(v1.inv()).affianca(v3.mult(v1.inv()));
452     case 2:
453     case 10:
454       // phi
455       Elemento a1 = new Elemento(new Vector());
456       Elemento a2 = new Elemento(new Vector());
457       for (i = 0; i < dim; i++) {
458         if (i < (dim / 2)) {
459           a1.addValue((BigInteger) this.valori.get(i));
460         }
461         else {
462           a2.addValue((BigInteger) this.valori.get(i));
463         }
464       }
465       Elemento fa = Elemento.UNO(dim / 2)
466         .sub(a1.pow2()).sub(a2.pow2()).add(a1.mult(a2));
467       Elemento sotto = Elemento.UNO(dim / 2).sub(fa)
468         .affianca(a1).affianca(a2)
469         .affianca(fa.neg()).affianca(Elemento.ZERO(dim));
470       Elemento sopra = Elemento.UNO(dim / 2)
471         .affianca(a1).affianca(a2)
472         .affianca(fa).affianca(Elemento.ZERO(dim));
473       return sopra.mult(sotto.inv());
474     default:
475       return null;
476     }
477   }
478
479
480   public Elemento sigma() {
481     int i;
482
483     switch (this.getSize()) {
484     case 30:
485       Elemento z = new Elemento(new Vector());
486       z = this.pow(qalfa.multiply(NumeroPrimo.calcolaPhi30(q)));
487       return this.mult(z.inv()).affianca(z);
488     case 60:
489       Elemento x = new Elemento(new Vector());
490       Elemento y = new Elemento(new Vector());
491       for (i = 0; i < 60; i++) {
492         if (i < 30) {
493           x.addValue((BigInteger) this.valori.get(i));
```

```
494         }
495         else {
496           y.addValue((BigInteger) this.valori.get(i));
497         }
498       }
499       return x.mult(y);
500     }
501     return null;
502   }
503
504   public static void main(String[] args) {
505     long adesso;
506
507     adesso = System.currentTimeMillis();
508
509     // Elementi noti, eventualmente unici per tutto il mondo
510     q = new BigInteger("1986798557");
511     BigInteger l = new
512       BigInteger("21672500148679066100334925909312112976495633458941");
513     BigInteger esp = q.pow(30).subtract(BigInteger.ONE).divide(l);
514     Elemento h = Elemento.aCaso(30);
515     Elemento g = h.pow(esp);
516     qalfa = new BigInteger("1578947401246833766");
517
518     // La mia chiave privata, un numero a caso tra 1 e l
519     BigInteger a = new
520       BigInteger("385331897195938659226172100857300914611270286962");
521
522     // La mia chiave pubblica
523     Elemento ga = g.pow(a);
524
525     adesso = (System.currentTimeMillis() - adesso);
526     System.out.println("Tempo impiegato prima parte: " + adesso + " ms");
527     System.out.println("Moltiplicazioni in Fq: " + Molt);
528     System.out.println();
529     Molt = 0;
530
531     // Scelta dei parametri per il singolo messaggio
532     adesso = System.currentTimeMillis();
533     BigInteger k = Primes.random(l.toString().length() - 1, false);
534     BigInteger M = new BigInteger("25");
535     Elemento d = g.pow(k);
536
537     // Messaggio criptato
538     Elemento m = g.pow(M);
539     Elemento e = m.mult(ga.pow(k));
540     // System.out.println("e = " + e);
541
542     // Applico la prima compressione (CEILIDH)
543     Elemento dr = d.rhoPhi();
544
545     // Separo il vettore dr compresso formato da 10 elementi
546     Elemento d1 = new Elemento(new Vector());
547     Elemento d2 = new Elemento(new Vector());
```

```
548     d1.addValue(dr.getValue(0));
549     d1.addValue(dr.getValue(1));
550     d2.addValue(dr.getValue(2));
551     d2.addValue(dr.getValue(3));
552     d2.addValue(dr.getValue(4));
553     d2.addValue(dr.getValue(5));
554     d2.addValue(dr.getValue(6));
555     d2.addValue(dr.getValue(7));
556     d2.addValue(dr.getValue(8));
557     d2.addValue(dr.getValue(9));
558
559     // Espansione di d1
560     Elemento gamma12 = d1.rhoPhi();
561     Elemento gamma = new Elemento(new Vector());
562
563     for (int i = 0; i < 6; i++) {
564       Elemento gammatmp = new Elemento(new Vector());
565       gamma = gamma.affianca(gammatmp.affianca(gamma12.getValue(i)).f_i());
566     }
567
568     // Compressione finale
569     Elemento dainviare = e.affianca(gamma).sigma().rhoPhi();
570     adesso = (System.currentTimeMillis() - adesso);
571     System.out.println("Tempo impiegato seconda parte: " + adesso + " ms");
572     System.out.println("Moltiplicazioni in Fq: " + Molt);
573     System.out.println();
574     System.out.println("Da inviare:");
575     System.out.println(dainviare);
576     System.out.println(d2);
577     System.out.println();
578     System.out.println("Messaggio inviato: " + m);
579     System.out.println();
580
581     // Espansione
582     adesso = System.currentTimeMillis();
583     Molt = 0;
584     Elemento lambdamu = dainviare.rhoPhi().sigma();
585
586     Elemento newe = new Elemento(new Vector());
587     Elemento g12 = new Elemento(new Vector());
588     for (int i = 0; i < 60; i++) {
589       if (i < 30) {
590         newe.addValue((BigInteger) lambdamu.valori.get(i));
591       }
592       else {
593         g12.addValue((BigInteger) lambdamu.valori.get(i));
594       }
595     }
596
597     Elemento newgamma = new Elemento(new Vector());
598     for (int i = 0; i < 6; i++) {
599       Elemento gammatmp = new Elemento(new Vector());
600       for (int j = 0; j < 5; j++) {
601         gammatmp.addValue(g12.getValue(j + 5 * i));
```

```
602         }
603         newgamma = newgamma.affianca(gammatmp.f_i());
604       }
605 
606       Elemento newd = newgamma.rhoPhi().affianca(d2).rhoPhi();
607       Elemento newm = newe.mult(d.inv().pow(a));
608       adesso = (System.currentTimeMillis() - adesso);
609       System.out.println("-------------------------------------------------");
610       System.out.println();
611       System.out.println("Tempo impiegato terza parte: " + adesso + " ms");
612       System.out.println("Moltiplicazioni in Fq: " + Molt);
613       System.out.println();
614       System.out.println("Messaggio ricevuto: " + newm);
615     }
616 }
```

Bibliografia

[1] Leonard M. Adleman e Jonathan Demarrais. A subexponential algorithm for discrete logarithms over all finite fields, *Math. Comput.* **61** (1993), n. 203, 1–15.

[2] Nicholas C. Alexander. *Algebraic tori in cryptography*, inedito (2005).

[3] Robert B. Ash. *Abstract algebra: the basic gratuate year*, inedito (2002).

[4] Carlo Bertoluzza. *Dispense del corso di Crittografia*, inedito (2003).

[5] Ian Blake, Gadiel Seroussi e Nigel Smart. *Elliptic curves in cryptography*, Cambridge University Press, 1999.

[6] Daniel Bleichenbacher, Wieb Bosma e Arjen K. Lenstra. Some remarks on Lucas-based cryptosystems, *Advances in cryptology—CRYPTO '95 (Santa Barbara, CA, August 27–31)*, Lect. Notes in Comp. Sci. **963** (1995), 386–396.

[7] Wieb Bosma, James Hutton e Eric R. Verheul. The XTR public key system, *Advances in cryptology—ASIACRYPT 2002 (Queenstown, New Zealand, December 1–5)*, Lect. Notes in Comp. Sci. **2501** (2002), 46–63.

[8] Egbert Brieskorn e Horst Knörrer. *Plane algebraic curves*, Birkhäuser, 1986.

[9] Andries E. Brouwer, Ruud Pellikaan e Eric R. Verheul. Doing more with fewer bits, *Advances in cryptology—ASIACRYPT '99 (Singapore, November 14–18)*, Lect. Notes in Comp. Sci. **1716** (1999), 321–332.

[10] N. G. de Bruijn. On the factorization of cyclic groups. *Indagationes Mathematicae* **15** (1953), 370–377.

[11] Emanuele Cesena. *Crittografia con XTR*, inedito, 2003.

[12] H. Cohen, A. Miyaji e T. Ono. Efficient elliptic elliptic curve exponentiation using mixed coordinates, *Advances in cryptology—ASIACRYPT '98 (Beijing, China, October 18–22)*, Lect. Notes in Comp. Sci. **1514** (1998), 51–65.

[13] Whitfield Diffie e Martin E. Hellman. New directions in cryptography, *IEEE Trans. Information Theory* **IT-22** (1976), no. 6, 644–654.

[14] Marten van Dijk, Robert Granger, Dan Page, Karl Rubin, Alice Silverberg, Martijn Stam e David Woodruff. Practical cryptography in high dimensional tori, *Advances in cryptology—EUROCRYPT 2005 (Aarhus, Denmark, May 22-26)*, Lect. Notes in Comp. Sci. **3494** (2005), 234–250.

[15] Marten van Dijk e David Woodruff. Asymptotically optimal communication for torus-based cryptography, *Advances in cryptology—CRYPTO 2004 (Santa Barbara, CA, August 15–19)*, Lect. Notes in Comp. Sci. **3152** (2004), 157–178.

[16] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms, *IEEE Trans. Information Theory* **31** (1985), no. 4, 469–472.

[17] Taher ElGamal. A subexponential-time algorithm for computing discrete logarithms over $GF(p^2)$, *IEEE Trans. Information Theory* **31** (1985), no. 4, 473–481.

[18] Francesco Fabris. *Teoria dell'informazione, codici, cifrari*, Bollati Boringhieri, 2001.

[19] Paolo Ferragina e Fabrizio Luccio. *Crittografia. Principi, algoritmi, applicazioni*, Bollati Boringhieri, 2001.

[20] Clemens Fuchs, Attila Pethö e Robert F. Tichy. On the diophantine equation $G_n(x) = G_m(P(x))$: higher-order recurrences, *Trans. Amer. Math. Soc* **355** (2003), no. 11, 4657–4681.

[21] Steven D. Galbraith e Nigel P. Smart. A cryptographic application of Weil descent, *IMA—Cryptography and Coding '99 (Cirencester, UK, December 20–22)*, Lect. Notes in Comp. Sci. **1746** (1999), 191–200.

[22] Steven D. Galbraith. Weil descent of jacobians, *Discrete Appl. Math.* **128** (2003), 165–180.

[23] Yves Gallot. *Cyclotomic polynomials and prime numbers*, inedito, 2001.

[24] Shuhong Gao e Hendrik W. Lenstra. Optimal normal bases, *Des. Codes Cryptogr.* **2** (1992), 315–323.

[25] Robert Granger, Dan Page e Martijn Stam. A comparison of CEILIDH and XTR, *Algorithmic number theory 2004 (Vermont, USA, June 13–18)*, Lect. Notes in Comp. Sci. **3076** (2004), 235–249.

[26] Robert Granger, Frederik Vercauteren. On the discrete logarithm problem on algebraic tori, *Advances in cryptology—CRYPTO 2005 (Santa Barbara, CA, August 14–18)*, Lect. Notes in Comp. Sci. **3621** (2005), 66–85.

[27] Jorge Guajardo e Christof Paar. Itoh-Tsujii inversion in standard basis and its application in cryptography and codes, *Des. Codes Cryptogr.* **25** (2002), 207–216.

[28] Patrick Horster, Markus Michels e Holger Petersen. *Digital signature schemes based on Lucas functions*, inedito, 1995.

[29] Neal Koblitz. *A course in number theory and cryptography*, Springer-Verlag, 1987.

[30] Neal Koblitz. Elliptic curve cryptosystems, *Math. Comput.* **48** (1987), no. 177, 203–209.

[31] Michael J. J. Lennon e Peter J. Smith. LUC: a new public key system, *Ninth IFIP Symposium on Computer Security* (1993), 103–117.

[32] Arjen K. Lenstra. Using cyclotomic polynomials to construct efficient discrete logarithm cryptosystems over finite fields, *Australasian Conference on Information Security and Privacy '97 (Sydney, Australia, July 7–9)*, Lect. Notes in Comp. Sci. **1270** (1997), 127–138.

[33] Arjen K. Lenstra e Martijn Stam. Speeding up XTR, *Advances in cryptology—ASIACRYPT 2001 (Gold Coast, Australia, December 9–13)*, Lect. Notes in Comp. Sci. **2248** (2001), 125–143.

[34] Arjen K. Lenstra e Martijn Stam. Efficient subgroup exponentiation in quadratic and sixth degree extensions, *Cryptographic Hardware and Embedded Systems 2002 (San Francisco Bay, USA, August 13–15)*, Lect. Notes in Comp. Sci. **2523** (2003), 318–332.

[35] Arjen K. Lenstra e Eric R. Verheul. Selecting cryptographic key sizes, *J. Cryptology* **14** (2001), no. 4, 255–293.

[36] Arjen K. Lenstra e Eric R. Verheul. *An overview of the XTR public key system*, inedito, 2001.

[37] Arjen K. Lenstra e Eric R. Verheul. The XTR public key system, *Advances in cryptology—CRYPTO 2000 (Santa Barbara, CA, August 20–24)*, Lect. Notes in Comp. Sci. **1880** (2000), 1–19.

[38] Hendrik W. Lenstra, Jr. Factoring integers with elliptic curves, *Ann. of Math.* **126** (1987), 649–673.

[39] Rudolf Lidl e Harald Niederreiter. *Introduction to finte fields and their applications*, Cambridge University Press, 1986.

[40] Alfred J. Menezes. *Elliptic curve public key cryptosystems*, Kluwer Academic Publishers, 1993.

[41] Alfred J. Menezes, Tatsuaki Okamoto e Scott A. Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field, *IEEE Trans. Information Theory* **39** (1993), no. 5, 1639–1646.

[42] Alfred J. Menezes, Scott A. Vanstone e Robert J. Zuccherato. Counting points on elliptic curves over $\mathbb{F}_{2^m}$, *Math. Comput.* **60** (1993), no. 201, 407–420.

[43] Ralph C. Merkle. Secure communications over insecure channels, *CACM* **21** (1978), no. 4, 294–299.

[44] Victor S. Miller. Use of elliptic curves in cryptography, *Advances in cryptology—CRYPTO '85 (Santa Barbara, CA, August 18–22)*, Lect. Notes in Comp. Sci. **218** (1986), 417–426.

[45] Peter L. Montgomery, Modular multiplication without trial division, *Math. Comput.* **44** (1985), 519–521.

[46] Ronald C. Mullin, I. M. Onyszchuk, Scott A. Vanstone e Richard M. Wilson. Optimal normal bases in $GF(p^n)$, *Discrete Appl. Math.* **22** (1988), 149–161.

[47] Andrew M. Odlyzko. Discrete logarithm: the past and the future, *Des. Codes Cryptogr.* **19** (2000), 129–145.

[48] Takashi Ono. Arithmetic of algebraic tori, *Ann. of Math.* **74** (1961), 101–139.

[49] Alessio Palmero. *Le curve ellittiche e la loro applicazione alla crittografia*, inedito, 2004.

[50] John M. Pollard. Monte Carlo methods for index computation mod p, *Math. Comput.* **32** (1978), no. 143, 918–924.

[51] Karl Rubin e Alice Silverberg. Torus-based cryptography, *Advances in cryptology—CRYPTO 2003 (Santa Barbara, CA, August 17–21)*, Lect. Notes in Comp. Sci. **2729** (2003), 349–365.

[52] Karl Rubin e Alice Silverberg. Algebraic tori in cryptography, *High primes and misdemeanours: lectures in honour of the 60th birthday of Hugh Cowie Williams* **41**, Fields Institute Communications Series, 317–326, American Mathematical Society, Providence, RI, 2004.

[53] Karl Rubin e Alice Silverberg. *Miscellaneous results on algebraic tori*, inedito.

[54] Karl Rubin e Alice Silverberg. Using primitive subgroups to do more with fewer bits, *Algorithmic number theory 2004 (Vermont, USA, June 13–18)*, Lect. Notes in Comp. Sci. **3076** (2004), 18–41.

[55] Edoardo Sernesi. *Geometria 1*, Bollati Boringhieri, 2000.

[56] René Schoof e Lambertus van Geemen. *Note per il corso di algebra*, inedito, 2001.

[57] Joseph H. Silverman. *The arithmetic of elliptic curves*, Springer-Verlag, 1986.

[58] Peter Smith e Christopher Skinner. A public-key cryptosystem and a digital signature based on the lucas function analogue to discrete logarithms, *Advances in cryptology—ASIACRYPT '94 (Wollongong, Australia, November 28–December 1)*, Lect. Notes in Comp. Sci. **917** (1995), 357–364.

[59] Douglas R. Stinson, *Cryptography: theory and practice*. Chapman & Hall/CRC, 2002.

[60] Valentin E. Voskresenskii. Stably rational algebraic tori, *J. Theor. Nombres Bordeaux* **11** (1999), 263–268.

[61] Valentin E. Voskresenskii. Algebraic groups and their birational invariants, *Translations of Mathematical Monographs*, **179**, American Mathematical Society, 1998.

[62] Lawrence C. Washington. *Elliptic curves: number theory and cryptography*, Chapman & Hall/CRC, 2003.

[63] William C. Waterhouse. Abelian varieties over finite fields, *Ann. Sci. Ecole Norm. Sup.* **2** (1969), no. 4, 521–560.

[64] Hugh C. Williams. On a generalization of the Lucas functions, *Acta Arith.* **29** (1972), 33–52.

[65] Andrew Wilson. Birational maps and blowing things up, *UoE Geometry Club*, inedito, 2006.