Universitat de Lleida
Escola Politècnica Superior
Màster en Enginyeria en Informàtica

Treball de Final de Màster

Estudi d'esquemes de signatura per a Smart Meters

Autor:
Santiago Risco Amigó

Directors:
Josep M. Miret Biosca
Javier Valera Martín

Setembre de 2014

Índex

Índex de taules

Índex de figures

Capítol 1

Introducció

Des de l'inici de la des-regulació de l'electricitat i la fixació de preus basat en el mercat, els serveis públics han estat buscant un mitjà perquè coincideixi el consum amb la generació. Els mesuradors elèctrics i de gas tradicionals només mesuren el consum total, i per tant no proporcionen informació de quan l'energia es consumeix. A més, aquest sistema és ineficient i poc acurat, ja que depèn de les lectures d'un operari ó està basat en lectures estimades.

La solució que es proposa són els mesuradors intel·ligents, tècnicament anomenats *Smart Meters*. Aquests mesuradors ens proporcionen un sistema de mesura automàtic i eficient que permet realitzar facturacions molt més acurades de l'ús d'electricitat que fan els consumidors, ja que aquests es comuniquen directament amb la central que tracta les dades mitjançant Internet.

Donada aquesta circumstància, les dades que viatgen a través de la xarxa són vulnerables a atacs informàtics, ja que poden revelar indirectament informació detallada del comportament dels consumidors, podent esbrinar els patrons d'estada a les llars. Per tant és important poder ocultar aquesta informació per tal de preservar la privadesa dels usuaris. Per aconseguir-ho cal emprar solucions criptogràfiques que no comportin un consum molt elevat de còmput ni de memòria per tal de minimitzar el cost dels *Smart Meters*.

Una solució criptogràfica proposada és el cas de l'esquema de la signatura Camenisch-Lysyanskaya (CL) basat en la fortalesa del logaritme discret mitjançant corbes el·líptiques emprant aplicacions bilineals no degenerades amb les que el problema bilineal de Diffie-Hellman és difícil. Aquestes aplicacions es poden utilitzar per dissenyar protocols eficients per signar i verificar els paquets de dades.

En aquest treball de final de màster estudiarem i implementarem l'esquema de signatura CL. Donarem una base per entendre com funciona i farem un anàlisi de rendiment de la implementació proposada, on valorarem aquells

aspectes que siguin més favorables per tal d'implementar aquest sistema en els *Smart Meters*.

Agraïments

Vull donar les gracies al meu director, Josep Maria Miret, per oferir-me la possibilitat de realitzar aquest treball de final de màster i vull agrair l'esforç que ha fet per poder-lo duu a terme.

Vull agrair als meus pares, i en especial a la meva dona, els ànims que he rebut d'ells durant aquests mesos.

Vull fer especial menció al meu codirector de treball, Javier Valera, que en les últimes etapes ha dedicat molts esforços en ajudar-me a completar aquest treball.

Capítol 2

Criptografia basada en pairings

La criptografia amb corbes el·líptiques basada en el problema del logaritme discret utilitza un tipus d'aplicacions bilineals, anomenades pairings, per tal de dissenyar protocols de signatura computacionalment eficients i segurs. En aquest capítol donarem una visió general del que són els pairings i perquè s'utilitzen i explicarem alguns dels més coneguts. A més, per tal d'entendre els conceptes, donarem algunes pautes matemàtiques.

2.1 Corbes el·líptiques

En aquesta secció farem una revisió de les característiques principals de les corbes el·líptiques que utilitzarem al llarg del nostre projecte. Aquestes corbes han estat estudiades al llarg dels darrers anys des d'un punt de vista criptogràfic, ja que tenen propietats molt útils a l'hora d'implementar algoritmes basats en el problema del logaritme discret d'una forma molt eficient.

Les primeres propostes en criptografia per tal d'utilitzar el grup de punts d'una corba el·líptica van ser donades per Víctor Miller [14] i Neal Koblitz [9] l'any 1985. El principal argument va ser que les corbes el·líptiques definides sobre un cos finit tenen estructura de grup finit, el que fa que siguin molt adequades en aplicacions de clau pública. A més, una altra propietat que les fa interessants és que per aconseguir un cert grau de seguretat, la criptografia basada en corbes el·líptiques pot utilitzar claus més curtes que d'altres tècniques.

2.1.1 Introducció a les corbes el·líptiques

Una corba el·líptica sobre un cos $\mathbb{K}$ és una corba sense punts singulars que ve donada per una equació de la forma

$$E/\mathbb{K} : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6, \qquad a_i \in \mathbb{K}, \tag{2.1}$$

anomenada equació general de Weierstrass. Si la característica de $\mathbb{K}$ és diferent de 2 i 3, l'equació (2.1) es pot reduir a una equació de la forma

$$E/\mathbb{K} : y^2 = x^3 + ax + b, \qquad a, b \in \mathbb{K}, \tag{2.2}$$

anomenada equació reduïda de Weierstrass. Aquesta equació no té punts singulars si, i només si, el seu discriminant $\Delta = 4a^3 + 27b^2$ és diferent de 0. Nosaltres utilitzarem l'equació (2.2) en lloc de l'equació (2.1).

El conjunt de punts racionals d'una corba el·líptica $E/\mathbb{K}$ es defineix com

$$E(\mathbb{K}) = \{(x, y) \in \mathbb{K} \times \mathbb{K} \mid y^2 = x^3 + ax + b\} \cup \{\mathcal{O}\},$$

on $\mathcal{O}$ és l'anomenat punt de l'infinit. Aquest punt és el $(0 : 1 : 0)$ en coordenades projectives. Al conjunt de punts $E(\mathbb{K})$ se'l pot dotar d'estructura de grup abelià amb una operació de suma on l'element neutre es el punt $\mathcal{O}$ (veure [20]).

Utilitzant l'operació de suma es pot definir l'operació producte d'un enter n per un punt P de la següent manera:

$$nP = \begin{cases} \underbrace{P + P + \cdots + P}_{n \text{ vegades}} & \text{si } n > 0, \\ \mathcal{O} & \text{si } n = 0, \\ \underbrace{(-P) + (-P) + \cdots + (-P)}_{n \text{ vegades}} & \text{si } n < 0. \end{cases}$$

Sigui $\mathbb{F}_q$ un cos finit de q elements, $q = p^k$, p primer (característica) i $k \in \mathbb{N}$. Si $\#E(\mathbb{F}_q)$ és el cardinal d'una corba el·líptica $E/\mathbb{F}_q$, llavors

$$\#E(\mathbb{F}_q) = q + 1 - t$$

amb $|t| \leq 2\sqrt{q}$. A l'enter t se l'anomena traça de l'endomorfisme de Frobenius de $E/\mathbb{F}_q$. Es diu que $E/\mathbb{F}_q$ és supersingular si $t \equiv 0 \pmod{p}$. En cas contrari es diu que és ordinària.

L'ordre d'un punt $P \in E(\mathbb{F}_q)$ és l'enter positiu més petit n tal que $nP = \mathcal{O}$. El subgrup de punts racionals de n-torsió es defineix com

$$E(\mathbb{F}_q)[n] = \{P \in E(\mathbb{F}_q) \mid nP = \mathcal{O}\}.$$

Aquest subgrup o bé és igual a $\{\mathcal{O}\}$ o bé és cíclic o de rang 2. Si és cíclic llavors està generat per un punt d'ordre n. Si és de rang 2 aleshores està generat per dos punts d'ordre n linealment independents. Sobre la clausura algebraica de $\mathbb{F}_q$, $E(\overline{\mathbb{F}_q})[n]$ sempre és de rang 2, és a dir,

$$E(\overline{\mathbb{F}_q})[n] \simeq \mathbb{Z}/n\mathbb{Z} \times \mathbb{Z}/n\mathbb{Z}.$$

Per a més informació consultar [20].

2.2 Problema del logaritme discret

Sigui g un generador del grup multiplicatiu $\mathbb{F}_q^*$. Llavors, per a cada $a \in \mathbb{F}_q$ existeix un enter x tal que $a = g^x$. El mínim enter x que satisfà aquesta equació s'anomena el logaritme discret de a en base g. El problema del logaritme discret (DLP – *Discrete Logarithm Problem*) tracta de trobar, donats un generador g i un element a, el logaritme discret de a en base g, és a dir, la x tal que $a = g^x$.

Sigui $E(\mathbb{F}_q)$ el conjunt de punts d'una corba el·líptica $E/\mathbb{F}_q$ amb $\#E(\mathbb{F}_q) = h \cdot l$, sent l un nombre primer gran. Donat un punt P, generador d'un subgrup cíclic d'ordre l de $E(\mathbb{F}_q)$, i donat un punt $Q \in \langle P \rangle$, l'enter k tal que $Q = kP$ s'anomena el logaritme discret de Q en base P. El problema del logaritme discret sobre corbes el·líptiques (ECDLP – *Elliptic Curve Discrete Logarithm Problem*) tracta de trobar l'enter k tal que $Q = kP$.

A continuació donarem els beneficis d'utilitzar corbes el·líptiques en comptes de cossos finits, però en primer lloc cal saber que l'operació de grup, el *hashing* i la generació aleatòria d'elements son considerablement més costoses que les mateixes operacions en cossos finits.

La fortalesa deriva del fet de que no s'ha descobert un algoritme eficient per solucionar el ECDLP per a corbes el·líptiques. Els millors mètodes estan basats en la paradoxa de birthday, com la Rho de Pollard [1], que tenen un temps d'execució estimat de l'orde $O(\sqrt{n})$, on n es l'ordre del grup. En el cas de les corbes el·líptiques, un ordre de grup al voltant de 160 bits és suficient per evitar aquests atacs, en canvi, pel cas del DLP, existeixen atacs en temps subexponencials, el que es tradueix en que s'han d'emprar cossos finits d'almenys 1024 bits. El fet d'utilitzar cossos sis vegades més petits fa que es compensin les operacions més complexes ja que utilitzarem menys còmput i memòria, fet que és molt important pels *Smart Meters*.

2.3 Pairings

Siguin $\mathbb{G}_1$, $\mathbb{G}_2$ i $\mathbb{H}$ tres grups cíclics d'ordre un nombre primer r i siguin g_1 i g_2 dos generadors, respectivament, de $\mathbb{G}_1$ i $\mathbb{G}_2$. Un pairing o aparellament e és una aplicació

$$e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{H}$$

tal que:

1. (**Bilinealitat**) $\forall a, b \in \mathbb{Z}_r$ se satisfà que $e(g_1^a, g_2^b) = e(g_1, g_2)^{ab}$.
2. (**No degeneració**) $e(g_1, g_2) \neq 1$.

La tupla $(r, g_1, g_2, \mathbb{G}_1, \mathbb{G}_2, \mathbb{H})$ es diu que és una configuració bilineal asimètrica. Si $\mathbb{G}_1 = \mathbb{G}_2 = \mathbb{G}$ i g és un generador de $\mathbb{G}$, la tupla $(r, g, \mathbb{G}, \mathbb{H})$ és una configuració bilineal simètrica.

Per a que un pairing e sigui útil en criptografia cal que sigui computacionalment eficient i que el problema bilineal de Diffie-Hellman sigui difícil. Aquest problema demana, donats un pairing e, un generador g del grup $\mathbb{G}$ i elements g^a, g^b i g^c, calcular $e(g, g)^{abc}$.

Es conegut que es poden definir pairings amb corbes el·líptiques, com el de Weil o el de Tate [3]. Donat que diferents corbes ens poden donar diferents aplicacions bilineals, ens fixem en una propietat molt important de les corbes el·líptiques, el grau d'immersió k, que ens dóna la resistència de la corba enfront atacs contra el logaritme discret sobre cossos finits. Contra més gran es el valor de k, més resistència, però les operacions són computacionalment més complexes i menys eficients.

Definició (Grau d'immersió)
Sigui $E/\mathbb{F}_q$ una corba elíptica i sigui l un nombre primer gran tal que $l \mid \#E(\mathbb{F}_q)$. S'anomena grau d'immersió de $E/\mathbb{F}_q$ respecte a l al mínim enter positiu k tal que

$$l \mid (q^k - 1).$$

En cas de que l sigui el major divisor primer de $\#E(\mathbb{F}_q)$, k es denomina simplement grau d'immersió de $E/\mathbb{F}_q$.

És per aquest motiu que és molt important escollir de forma apropiada la corba perquè aquesta sigui prou forta als atacs i perquè les operacions siguin computacionalment eficients. En particular, les corbes supersingulars són idònies per aquest propòsit ja que sempre tenen grau d'immersió $k \leq 6$ [12]. Per contra, les corbes ordinàries amb grau d'immersió petit són molt complicades de trobar ja que la seva caracterització es molt complexa i són una minoria [7].

La llibreria que utilitzarem per tal d'implementar el nostre esquema de signatura té definits cinc tipus de corbes el·líptiques:

- **Tipus A:** Són corbes supersingulars de la forma $y^2 = x^3 + ax$ i amb grau d'immersió $k = 2$.

- **Tipus D:** Emprant el mètode CM [2] és possible construir corbes amb grau d'immersió $k = 6$ [16] [19].

- **Tipus E:** Emprant el mateix mètode que en les corbes de tipus D, podem obtenir corbes amb grau d'immersió $k = 1$, on els còmputs necessaris per tal de fer el pairing es poden realitzar sobre $\mathbb{F}_q$.

- **Tipus F:** Tanmateix, emprant el mètode CM i considerant els polinomis ciclotòmics, Barreto i Naehring van construir una família de corbes amb grau d'immersió $k = 12$.

- **Tipus G:** Són corbes de grau d'immersió $k = 10$ descobertes per Freeman [6].

2.4 Pairings de Tate i Weil

En aquesta secció definirem els dos tipus d'aparellament que s'utilitzen en la llibreria que emprarem per implementar l'esquema de signatura CL. Aquests aparellaments són aplicacions bilineals que assignen a un parell de punts de la corba E una arrel d'ordre l en un cos d'extensió $\mathbb{F}_{q^k}$, on k és el grau d'immersió de la corba i l és un nombre primer i divisor del cardinal de la corba. Prenent per entrada punts de l-torsió, es defineixen mitjançant funcions racionals i la sortida són elements d'un cos finit.

Pairing de Tate

L'aparellament de Tate pren els punts P i Q, ambdós de l-torsió, que si bé no tenen perquè estar definits en el cos base, si que ho han de estar en $\mathbb{F}_{q^k}$. I es calcula com:

$$t_l(P, Q) = f_{l,P}(Q + R),$$

on R és un punt auxiliar de la corba i la funció $f_{l,P}$ és un quocient de dos polinomis en dos variables.

La dificultat d'aquest càlcul resideix en la construcció d'aquestes funcions, però l'algoritme de Miller [13] [15] permet realitzar aquesta construcció de

forma eficient. Podem obtenir les funcions $f_{l,T}$, per qualsevol punt T, mitjançant les següents identitats:

$$f_{0,T} = f_{1,T} = 1$$

$$f_{m+n,T} = f_{m,T} f_{n_t} g_{mT,nT}$$

on $g_{U,V} = \frac{L_{U,V}}{L_{(U+V),-(U+V)}}$ i $L_{U,V} = 0$ és l'equació de la recta que passa pels punts U i V.

Pairing de Weil

L'aparellament de Weil pot considerar-se una variant de l'aparellament de Tate. Com aquest últim pren els punts P i Q de l-torsió d'una corba el·líptica.

El valor de l'aparellament w_l es calcula com el quocient:

$$e_l(P,Q) = \frac{f_{l,P}(Q+R) f_{l,Q}(S)}{f_{l,P}(R) f_{l,Q}(P+S)},$$

on R, S són punts auxiliars de la corba i les funcions $f_{l,P}$ i $f_{l,Q}$ són quocient de dos polinomis en dos variables, que poden calcular-se de forma eficient amb l'algoritme de Miller, de la mateixa forma que hem explicat en la definició de l'aparellament de Tate.

Capítol 3

Signatura Camenisch-Lysyanskaya

En aquest capítol presentarem l'esquema de signatura proposat per Jan Camenisch i Anna Lysyanskaya [4]. Aquest model criptogràfic s'ha proposat per la implementació del protocol que utilitzen els *Smart Meters* per comunicar-se de forma segura [17].

3.1 Introducció als Smart Meters

Per tal de proporcionar el context en el que hem fet el nostre treball, descriurem breument que són els *Smart Meters*.

Aquests dispositius es diferencien dels mesuradors autòmats en què introdueixen sensors per fer lectures en temps real i, a més, poden incloure mesures de sobretensions i distorsió harmònica, cosa que permet el diagnòstic dels problemes de qualitat de l'energia que se subministra.

Els *Smart Meters* van néixer de la necessitat d'automatitzar el sistema de lectura del consum d'electricitat per tal de fer les facturacions molt més acurades de l'ús que fan els consumidors. Fins i tot poden permetre als consumidors no solament triar les millors tarifes, sinó també distingir entre les hores de consum, el que alhora permetria un millor ús de la xarxa.

Per fer-ho possible, aquests dispositius formen part d'una complexa infraestructura que, mitjançant tecnologies de la informació i comunicació, tracta d'intercomunicar totes les fases que intervenen, des de la generació de l'energia fins que aquesta es consumeix. Aquesta infraestructura és coneguda amb el nom de *Smart Grid*.

Un dels problemes que sorgeixen de la utilització de xarxes de comunicació és que les dades són vulnerables a atacs informàtics, ja que obtenir aquestes

dades pot implicar esbrinar patrons d'estada a les llars, el que compromet la privadesa dels usuaris. Per ocultar aquesta informació cal emprar solucions criptogràfiques que no comportin un consum molt elevat de còmput ni de memòria per tal de minimitzar el cost dels *Smart Meters*.

3.2 Construcció dels blocs criptogràfics

Per tal de garantir que les mesures que realitza el *Smart Meter* realment estan ocultes, es poden utilitzar els esquemes de compromís o d'entrega i les proves de coneixement zero.

Els esquemes de compromís són unes primitives criptogràfiques que permeten a una part crear l'equivalent digital d'un missatge secret, és a dir, emmascarar el missatge original. Els compromisos tenen dos propietats importants: l'ocultació i la vinculació, és a dir, que han de protegir en secret el missatge i han d'assegurar que el contingut del missatge rebut és el mateix que l'enviat.

Les proves de coneixement zero són un mètode pel qual una part, generalment la entitat que signa, pot demostrar a l'entitat verificadora que coneix els valors amb els que s'ha emmascarat el missatge sense tenir que mostrar-los.

Un dels esquemes que assegura aquestes dues propietats és l'anomenat *Pedersen commitments* [18], basat en el logaritme discret. Aquest esquema s'utilitza en el cas que el missatge es calculi de la següent forma: $C = g^r h^o$, on g i h són generadors d'un grup $\mathbb{G}$ d'ordre un primer p, el missatge és $r \in \mathbb{Z}_p$ i o és un valor d'apertura que també pertany a $\mathbb{Z}_p$. Basat en aquest, trobem una variant, l'anomenat *Fujisaki-Okamoto commitments* [8], que permet emmascarar valors enters negatius.

Però existeixen dos problemes amb aquests mètodes. D'una banda ens trobem amb que hi ha protocols eficients per poder fer veure a l'entitat verificadora que es coneixen els valors emprats per emmascarar sense tenir que desemmascarar el missatge, dit d'una altra forma, podem superar les proves de coneixement zero. I d'altra banda ens trobem amb la suplantació de canal, ja que és relativament senzill capturar la IP del *Smart Meter* i assignar-la a un altre dispositiu.

Aquestes dues febleses poden permetre a d'altres mesuradors enviar informació falsejada a l'entitat verificadora sense que aquesta detecti que no és el mesurador original. És per quest motiu pel qual s'han obert línies d'investigació que tracten de resoldre, o millor dit, mitigar aquestes febleses. Un dels esquemes que tracta de resoldre aquest problema és l'esquema de les *Signatures Camenisch-Lysyanskaya (CL)* [4].

3.3 Signatura Camenisch-Lysyanskaya (CL)

Els esquemes de signatures digitals, proposats per Diffie i Hellman, proporcionen una base important per a molts protocols criptogràfics, com els plans de credencials anònimes o la votació electrònica. No obstant, l'eficiència d'aquestes construccions generals, i també el fet que aquests esquemes de signatura requereixen del canvi de la clau privada del signant entre invocacions de l'algorisme de signatura, fa que aquestes solucions no siguin desitjables en la pràctica.

Per aquest motiu Camenisch i Lysyanskaya van proposar un nou esquema de signatura [4] que es basa en una suposició del logaritme discret introduït per Lysyanskaya, Rivest, Sahai, i Wolf (LRSW)[11] que utilitza aplicacions bilineals. Aquesta suposició es va crear per tal de poder utilitzar grups genèrics a l'hora d'implementar l'esquema, i ser independent del supòsit de la presa de decisions Diffie-Hellman.

Descriurem a continuació la generació de claus, la signatura i la verificació de la signatura.

3.3.1 Esquema de signatura CL

Sigui n el nombre de missatges per cada bloc. Per generar les claus pública i privada primer generem la tupla

$$(p, \mathbb{G}, \mathbb{H}, g, h, e),$$

on $\mathbb{G} = \langle g \rangle$ i $\mathbb{H} = \langle h \rangle$ són dos grups cíclics d'ordre un nombre primer p i $e\colon \mathbb{G} \times \mathbb{G} \to \mathbb{H}$ és un pairing eficient de computar. En segon lloc triem aleatòriament els paràmetres x, y, z_2, …, $z_n \in \mathbb{Z}_p$. Amb aquests paràmetres calculem

$$X = g^x, Y = g^y,$$

$$Z_i = g^{z_i}, W_i = Y^{z_i} \quad \forall i \in \{2, \ldots, n\}.$$

Llavors:

- La clau pública és la tupla $(p, \mathbb{G}, \mathbb{H}, g, h, e, X, Y, [Z_i], [W_i])$.
- La clau privada és la tupla $(p, \mathbb{G}, \mathbb{H}, g, h, e, X, Y, [Z_i], [W_i], x, y, [z_i])$.

Cal remarcar que la seqüència $[W_i]$ només s'utilitza per a la prova de coneixement zero.

Generació de la signatura

Per signar un bloc de n missatges m_i, $i \in \{1, \ldots, n\}$, en primer lloc triem aleatòriament un element $a \in \mathbb{G}$ per a després calcular $b = a^y$. En segon lloc calculem

$$A_i = a^{z_i}, B_i = A_i^y \quad \forall i \in \{2, \ldots, n\}.$$

Finalment, calculant

$$\sigma = a^{x+xym_1} \prod_{i=2}^{n} A_i^{xym_i},$$

tenim que la signatura del bloc és la tupla

$$(a, [A_i], b, [B_i], \sigma).$$

Verificació de la signatura

Donada la clau pública, el bloc $[m_i]$ i la signatura $(a, [A_i], b, [B_i], \sigma)$, aquesta s'accepta si es compleixen les següents igualtats:

$$e(a, Y) = e(b, g),$$

$$e(a, Z_i) = e(A_i, g) \quad \forall i \in \{2, \ldots, n\},$$
$$e(A_i, Y) = e(B_i, g) \quad \forall i \in \{2, \ldots, n\},$$

$$e(g, \sigma) = e(X, a) \cdot e(X, b)^{m_1} \cdot \prod_{i=2}^{n} e(X, B_i)^{m_i}.$$

Demostrem, mitjançant les propietats dels pairings que hem vist en la secció 2.3, que aquestes igualtats es compleixen.

$$e(a, Y) = e(a, g^y) = e(a^y, g) = e(b, g),$$

$$e(a, Z_i) = e(a, g^{z_i}) = e(a^{z_i}, g) = e(A_i, g),$$
$$e(A_i, Y) = e(A_i, g^y) = e(A_i^y, g) = e(B_i, g),$$

$$e(g, \sigma) = e(g, a^{x+xym_1} \prod_{i=2}^{n} A_i^{xym_i}) =$$

$$= e(g, a^x) \cdot e(g, a^{xym_1}) \cdot \prod_{i=2}^{n} e(g, A_i^{xym_i})$$

$$= e(g^x, a) \cdot e(g^x, a^y)^{m_1} \cdot \prod_{i=2}^{n} e(g^x, A_i^y)^{m_i}$$

$$= e(X, a) \cdot e(X, b)^{m_1} \cdot \prod_{i=2}^{n} e(X, B_i)^{m_i}.$$

Per a més informació consultar [4].

Capítol 4

Implementació

En aquest capítol, en primer lloc, introduirem el software utilitzat per tal d'implementar l'esquema de signatura de Camenisch-Lysyanskaya (CL). Una vegada vist això, explicarem breument com l'hem implementat.

4.1 Software utilitzat

Per tal d'implementar l'esquema de signatura CL ens hem servit del llenguatge de programació Java emprant l'entorn de desenvolupament Netbeans. Hem escollit aquest llenguatge perquè és multiplataforma i perquè existeix una llibreria anomenada Java Pairing Based Cryptography (jPBC) [5] que ens permet treballar fàcilment amb pairings. Aquesta llibreria es basa en la llibreria PBC que va desenvolupar en llenguatge C Ben Lynn com a part de la seva tesis [10]. La llibreria jPBC importa tot el codi de la llibreria PBC i li dóna una mena de capa perquè aquesta sigui més intuïtiva per als programadors. Quan és possible, degut a que C és més òptim que Java, jPBC crida a la llibreria PBC. jPBC també disposa de la possibilitat de fer un pre-processament de les operacions d'exponenciació i de pairings. Això s'ha fet per una banda perquè Java és un llenguatge "lent" i per l'altra perquè normalment aquestes operacions es realitzen més d'una vegada. Finalment, cal dir que jPBC és LGPL.

4.1.1 Operacions bàsiques per a pairings

En aquesta secció donem les operacions bàsiques de jPBC per tal de treballar amb pairings.

– Utilitzar les funcions implementades en C quan sigui possible:

```
PairingFactory.getInstance().setUsePBCWhenPossible(true);
```

– Definir un pairing (corba de tipus A):

```
PairingParametersGenerator ppg =
    new TypeACurveGenerator(rBits, qBits);
Pairing e =
    PairingFactory.getPairing(ppg.generate(), null);
```

– Obtenir els camps (estructures algebràiques) que intervenen en un pairing:

```
Field Zr = e.getZr();
Field G1 = e.getG1();
Field G2 = e.getG2();
Field H = e.getGT();
```

– Obtenir un element aleatori d'un camp:

```
Element elt = G1.newRandomElement();
```

– Fer que un element sigui immutable, és a dir, que el seu valor sigui sempre el mateix:

```
elt = elt.getImmutable();
```

– Obtenir el pre-processament de l'operació d'exponenciació:

```
ElementPowPreProcessing eppp =
    elt.getElementPowPreProcecessing();
```

– Obtenir el pre-processament de l'operació de pairing:

```
PairingPreProcessing ppp =
    e.getPairingPreProcessingFromElement(elt);
```

4.2 Implementació de l'esquema de signatura CL

En aquesta secció expliquem breument les decisions que hem pres a l'hora d'implementar l'esquema de signatura CL.

Com hem vist a la secció 3.3, els esquemes de signatura utilitzen una clau pública, una clau privada i una signatura. És per aquest motiu que hem dissenyat una classe per cada una d'elles: `PublicKey`, `PrivateKey` i `Signature`. Com és obvi, els atributs d'aquestes classes son els elements que hem vist a l'apartat 3.3.1, és a dir, la classe `PublicKey`, per exemple, té atributs anomenats `X`, `Y`, `Zi`, etc. Cal comentar, també, que la classe `PrivateKey` té un atribut de la classe `PublicKey`, ja que com hem vist, la clase privada és defineix com la clau pública més una sèrie de paràmetres.

Per tal de generar alguns valors tenim la classe `CLUtils`. Els mètodes d'aquesta classe són els següents:

- `getNElementsFromField`: Aquest mètode genera la seqüència $[z_i]$.
- `getxyElements`: Aquest mètode genera els elements x i y.
- `getXYElements`: Aquest mètode genera els elements X i Y.
- `getZiElements`: A partir de la seqüència $[z_i]$, aquest mètode genera la seqüència $[Z_i]$.
- `getWiElements`: A partir de la seqüència $[z_i]$, aquest mètode genera la seqüència $[W_i]$.

La classe encarregada de generar les claus, signar un bloc i verificar la signatura és la classe `CLSignature`. Aquesta classe, per cadascuna de les tasques anteriors, té definit un mètode:

```
public PrivateKey CLKeyGen(int n, String type)

public Signature CLSign(PrivateKey privateKey,
            ArrayList<Element> mi, Field G)

public boolean CLVerifySign(PublicKey publicKey,
            ArrayList<Element> mi, Signature sign)
```

A més d'aquests tres mètodes, també en té un per tal de triar i preparar un pairing: `setUpPairing`. A l'apèndix A podeu veure la implementació completa dels mètodes `CLKeyGen`, `CLSign` i `CLVerifySign`.

Finalment, per tal de provar l'esquema de signatura CL, tenim una classe anomenada `CLSignatureApp`.

Modificacions

Cal recordar que en la secció 2.3 vam veure els tipus de corbes que hi ha implementades a la llibreria jPBC. En el cas de les corbes de tipus A tenim que $\mathbb{G}_1 = \mathbb{G}_2$. Aquesta peculiaritat ens permet fer una variació del codi que, com veurem en el anàlisi de rendiment (secció 5.2), optimitza la signatura i la verificació.

D'una banda, com només és necessari emmagatzemar un grup, l'objecte de la clau pública perd un element. Com la clau privada conté la clau pública, també es beneficia de la pèrdua de l'element.

D'altra banda hem tingut que modificar l'ordre en el que els elements criden al mètode de pairing per tal de que el codi funcioni correctament.

Capítol 5

Anàlisi de rendiment

En aquest capítol mostrarem els resultats obtinguts de les proves de rendiment per tal de veure com es comporta el nostre codi amb diferents tipus de corbes. També analitzarem en quines situacions és millor utilitzar un tipus de corba que un altre.

5.1 Entorn de test

Com ja hem comentat al capítol anterior, hem utilitzat l'entorn de programació NetBeans perquè aquest és de gran ajuda a l'hora d'obtenir resultats gràcies a les eines de *profiling* que té integrat. Aquest entorn utilitza la versió de Java 1.7 i l'hem fet córrer sobre el sistema operatiu Linux Ubuntu 12.04 de 64 bits. Pel que fa al hardware, hem utilitzat una màquina amb 4GB de RAM i un processador Intel core i5 vPro amb dos nuclis que treballen a 3.2GHz.

5.2 Resultats

Per fer les proves hem decidit escollir les corbes de tipus A, D i F (veure secció 2.3). Pel que fa a la mida q dels cossos finits hem seguit les propostes de seguretat donades per Ben Lynn [10]. Un resum d'aquestes juntament amb un recordatori dels tipus de corbes és el següent:

- **Tipus A:** Corbes supersingulars amb grau d'immersió $k = 2$ i q de 512 bits.
- **Tipus D:** Corbes ordinàries amb grau d'immersió $k = 6$ i q de 171 bits.

- **Tipus F:** Corbes ordinàries amb grau d'immersió $k = 12$ i q de 160 bits.

L'objectiu de les proves és fer dues comparacions: una primera on comparem els temps d'execució de la signatura i de la verificació per a cada tipus de corba i una segona on comparem els consums de memòria. Aquestes comparacions sorgeixen de la necessitat de trobar aquelles corbes que minimitzin l'ús de CPU, així com de memòria, per tal de fer els *Smart Meters* el menys costosos possible.

Com la signatura depèn del nombre de missatges per bloc, les proves consistiran en signar i verificar blocs de 25, 50, 75 i 100 missatges per tal de veure amb claredat les diferències entre utilitzar un tipus de corba o un altre.

A més dels tres tipus anteriors de corbes, representarem amb A* la modificació realitzada a la implementació general per a les corbes de tipus A.

5.2.1 Comparació envers el temps d'execució

A continuació mostrem les taules amb les dades recollides dels temps d'execució de les proves que hem explicat anteriorment.

Tipus Corba	Signatura (mseg)	Verificació (mseg)
A	1600	1000
D	3500	3800
F	1600	15800
A*	249.3	357.7

Taula 5.1: Temps d'execució per signar i verificar blocs de 25 missatges.

Tipus Corba	Signatura (mseg)	Verificació (mseg)
A	3100	2000
D	6300	7500
F	2400	30000
A*	479.5	702.4

Taula 5.2: Temps d'execució per signar i verificar blocs de 50 missatges.

Tipus Corba	Signatura (mseg)	Verificació (mseg)
A	4700	3000
D	9500	11600
F	3100	47200
A*	709.8	1039.5

Taula 5.3: Temps d'execució per signar i verificar blocs de 75 missatges.

Tipus Corba	Signatura (mseg)	Verificació (mseg)
A	6130	3970
D	12000	14880
F	3380	59440
A*	958	1406.2

Taula 5.4: Temps d'execució per signar i verificar blocs de 100 missatges.

A continuació mostrem les taules amb les dades recollides dels temps d'execució de les proves que hem explicat anteriorment.

Com podem veure a les taules 5.1, 5.2, 5.3 i 5.4, en termes generals, la implementació modificada per a les corbes de tipus A és la que menys triga en signar i verificar. Per tant, tenint en compte això, ara analitzarem amb més profunditat els altres tres tipus de corbes, és a dir, els de la implementació general. Això ens donarà una visió més acurada de quin tipus de corba és millor utilitzar.

Si ens fixem en els augments dels temps d'execució tant de la signatura com de la verificació, podem veure que per als tres tipus de corbes els augments són de forma gairebé lineal respecte al nombre de missatges per bloc.

Respecte als temps d'execució de la signatura, contràriament al que podiem pensar en un principi, degut a que és el tipus amb el grau de seguretat més alt, el millor tipus de corbes és el F, ja que és el que té els temps d'execució més baixos, o dit d'una altra manera, l'increment de temps respecte la mida de bloc és el més petit. A l'altre extrem tenim les corbes de tipus D.

Respecte a la verificació, el millor tipus és el A. El pitjor amb diferència és el F ja que per a una mida de bloc de 100 missatges tarda quasi 1 minut (el tipus A només tarda 4 segons).

És curiòs que per al tipus A, contràriament al que succeix per als altres tipus, els temps d'execució de la verificació són més baixos que els de la signatura.

5.2.2 Comparació envers el consum de memòria

Per aquesta comparació ens hem servit de les gràfiques de consum de memòria que l'eina de *profiling* de NetBeans extreu de les execucions. En aquestes gràfiques podem veure la memòria que reserva la màquina virtual de Java (Heap Size) i la que està utilitzant (Used Heap).

A continuació mostrem els consums que s'han extret de l'execució seqüencial de la signatura i la verificació amb blocs de 100 missatges, ja que és on veiem amb més claredat els resultats obtinguts i que poden extrapolar-se a les execucions amb blocs de 25, 50 i 75 missatges.

En aquesta comparació no tindrem en compte la implementació modificada per a les corbes de tipus A, ja que, tal i com hem vist en la comparació envers el temps d'execució, aquesta implementació és la que menys recursos gasta i, per tant, no ens dóna una visió clara de la comparació d'aquest tipus de corba respecte als altres.

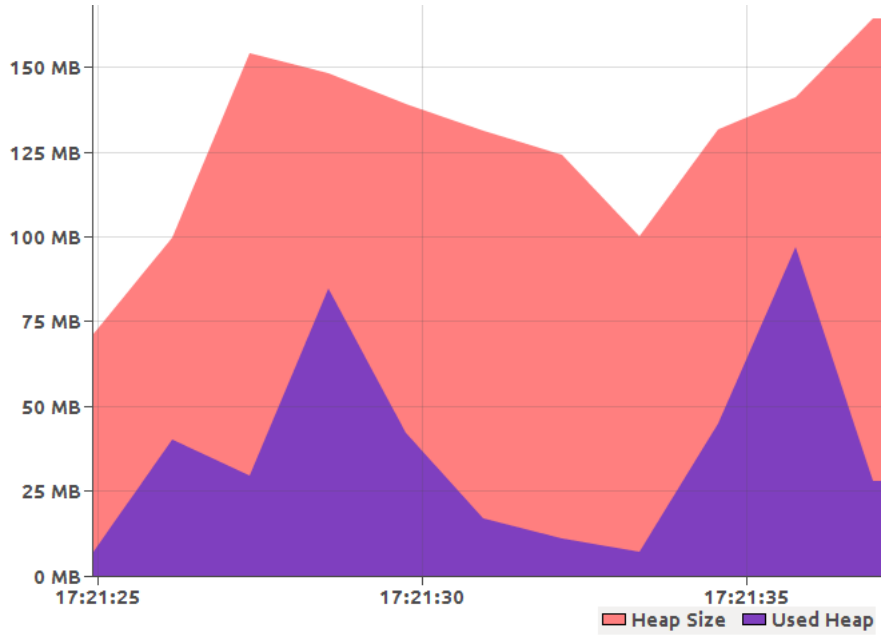

Figura 5.1: Gràfica del consum de memòria per a les corbes de tipus A

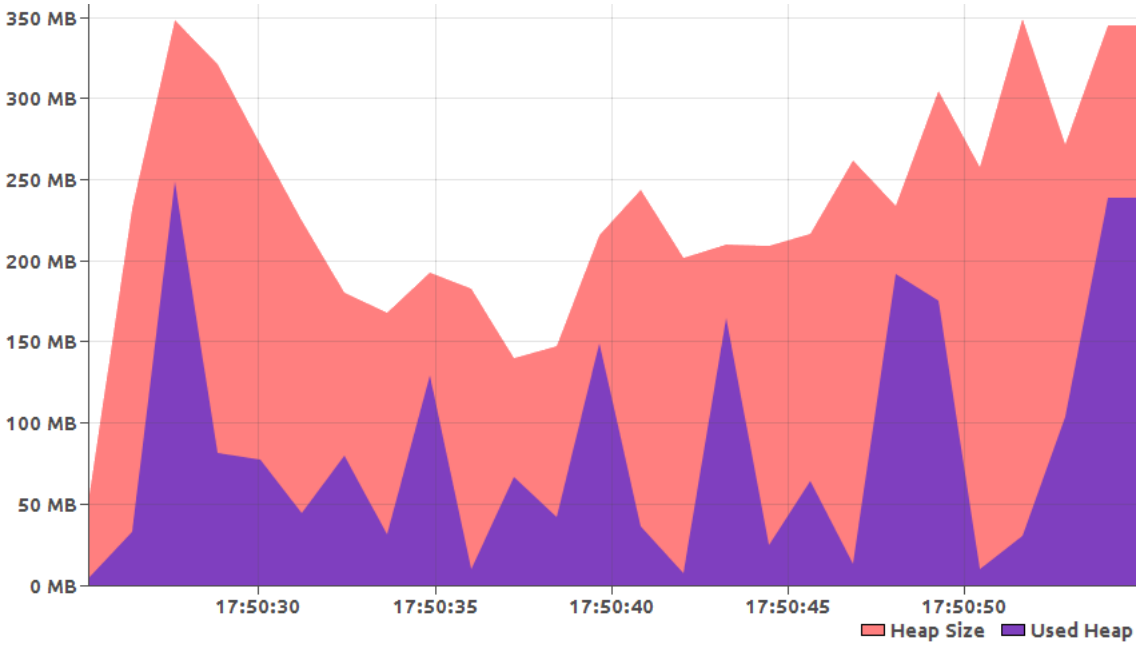

Figura 5.2: Gràfica del consum de memòria per a les corbes de tipus D

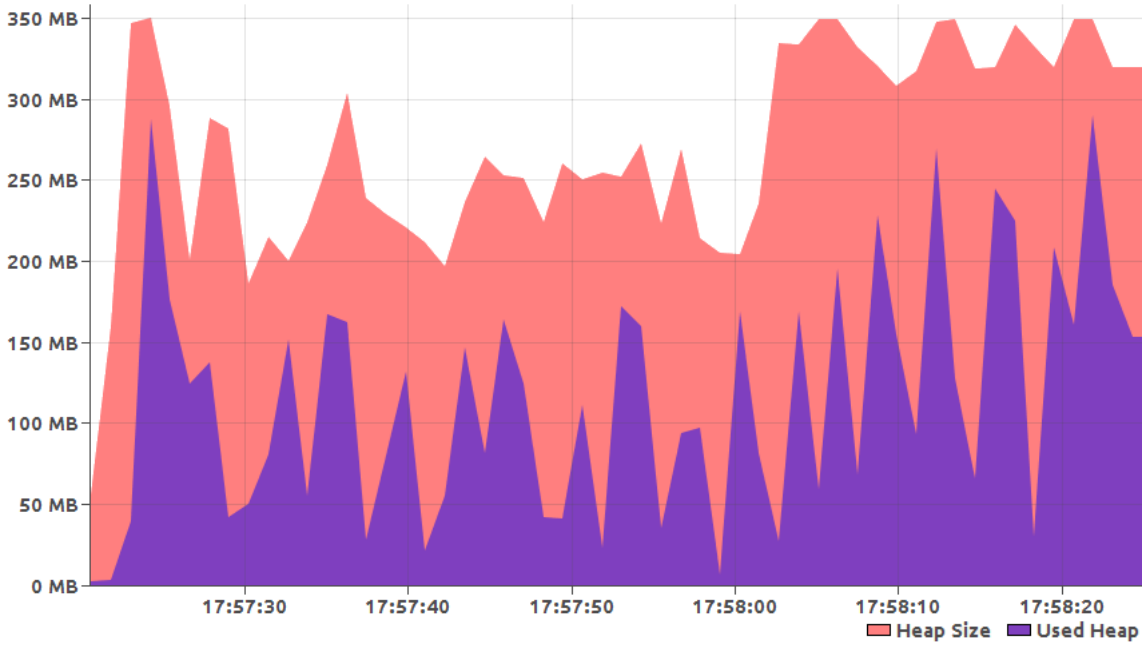

Figura 5.3: Gràfica del consum de memòria per a les corbes de tipus F

Pel que podem comprovar a la gràfica de les corbes de tipus A (gràfica 5.2.2), tenim dos pics molt diferenciats de consum que coincideixen amb la signatura i la verificació del bloc de missatges. Aquests pics no sobrepassen els 100MB, encara que la memòria que la màquina virtual de JAVA reserva, i que per tant està consumint, supera els 150MB en ambdós pics.

Pel que fa a les corbes de tipus D (gràfica 5.2.2), coneixent que aproximadament els primers 12 segons pertanyen a la signatura, podem veure una serie de pics de consum que coincideixen amb el càlcul dels paràmetres de la signatura A_i, B_i i $A_i^{xym_i}$ així com del càlcul de σ, que vam veure a la secció 3.3, sent el primer el més representatiu ja que arriba a prop els 250MB. Respecte a la verificació, podem veure que hi ha molts pics de memòria. Això es deu a que JAVA té implementat un sistema d'alliberament de memòria conegut com *Garbage Collector* que quan creu que es necessari allibera memòria i, al ser una execució llarga, actua en més ocasions, el que provoca que apareguin aquests pics.

Per a les corbes de tipus F (gràfica 5.2.2) succeeix una cosa similar. Tenim un pic molt alt de consum de memòria a l'hora de signar que en aquest cas supera els 275MB. Durant tota la verificació el *Garbage Collector* va actuant per estabilitzar la memòria entorn els 150-200MB. Cap al final de la verificació veiem com a l'hora de fer l'última validació en la que els pairings són molt pesats, la memòria torna a situar-se per sobre els 250MB tot i l'actuació del *Garbage Collector*.

5.3 Conclusions i futures línies de treball

Una vegada realitzada la implementació de l'esquema de signatura Camenisch-Lysyanskaya i havent extret els resultats de les proves amb diferents tipus de corbes, podem extreure les següents conclusions.

Com podem veure, els consums de memòria no són molt alts: entorn als 200MB amb pics de uns 300MB en els pitjors dels casos. Si tenim en compte que és senzill trobar dispositius amb uns 512MB de memòria i que la tendència és a que el preu d'aquesta baixi, és mes raonable fixar-se amb que el consum de CPU sigui el més baix possible.

Tenint en compte les fortaleses i febleses de les corbes analitzades, podem dir que utilitzar un tipus o un altre de corba depèn del protocol de comunicació que implementem entre el *Smart Meter* i l'entitat que recull les dades. Basant-nos en els més coneguts protocols per la transmissió de dades a través de Internet, podem fer les reflexions que a continuació expliquem.

Creiem que un protocol basat en la connexió, com és el cas del protocol TCP (*Transmission Control Protocol*), és el mes adient. Aquest protocol

tracta de suplir la pèrdua de paquets que provoca la congestió de la xarxa mitjançant notificacions de rebuda (*Acknowledgements*). Com que es reben i s'envien moltes dades, creiem que, a hores d'ara, les corbes de tipus A són les que millor s'adapten a aquesta situació, ja que són les que tenen una millor relació entre els temps de signatura i de verificació.

Com futures línies de treball relacionades amb aquest treball de final de màster podem dir que s'hauria de seguir investigant sobre aquests mètodes de signatura basats en pairings ja que en poc temps poden arribar a ser un substitut de la criptografia basada en el RSA. També s'hauria d'investigar sobre els tipus de corbes més adients per tal de crear un protocol que sigui capaç d'adaptar-se a qualsevol tipus extraient el màxim de rendiment possible de cadascun d'ells.

Bibliografia

[1] T. Bigordà. Atac al problema del Logaritme Discret mitjançant la Rho de Pollard. Universitat de Lleida, 2005.

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

[3] I. F. Blake, G. Seroussi, and N. P. Smart. *Advances in Elliptic Curve Cryptography*. Cambridge University Press, 2005.

[4] J. Camenisch and A. Lysyanskaya. Signature Schemes and Anonymous Credentials from Bilinear Maps. In *Advances in Cryptology – CRYPTO 2004*, volume 3152 of *LNCS*, pages 56–72, 2004.

[5] A. De Caro and V. Iovino. jPBC: Java Pairing Based Cryptography. In *ISCC 2011*, pages 850–855. IEEE, 2011.

[6] D. Freeman. Constructing Pairing-Friendly Elliptic Curves with Embedding Degree 10. In *ANTS-VII*, volume 4076 of *LNCS*, pages 452–465, 2006.

[7] D. Freeman, M. Scott, and E. Teske. A Taxonomy of Pairing-Friendly Elliptic Curves. *Journal of Cryptology*, 23:224–280, 2010.

[8] E. Fujisaki and T. Okamoto. Statistical Zero Knowledge Protocols to Prove Modular Polynomial Relations. In *Advances in Cryptology – CRYPTO '97*, volume 1294 of *LNCS*, pages 16–30, 1997.

[9] N. Koblitz. Elliptic Curve Cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.

[10] B. Lynn. *On the implementation of pairing-based cryptosystems*. PhD thesis, Stanford University, 2007.

[11] A. Lysyanskaya, R. L. Rivest, A. Sahai, and S. Wolf. Pseudonym Systems. In *SAC '99*, volume 1758 of *LNCS*, pages 184–199, 2000.

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

[13] V. S. Miller. Short Programs for functions on Curves. Unpublished manuscript, 1986.

[14] V. S. Miller. Use of Elliptic Curves in Cryptography. In *Advances in Cryptology – CRYPTO '85*, volume 218 of *LNCS*, pages 417–426, 1986.

[15] V. S. Miller. The Weil Pairing, and Its Efficient Calculation. *Journal of Cryptology*, 17:235–261, 2004.

[16] A. Miyaji, M. Nakabayashi, and S. Takano. New Explicit Conditions of Elliptic Curve Traces for FR-Reduction. *IEICE transactions on fundamentals of electronics, communications and computer*, 84(5):1234–1243, 2001.

[17] A. Molina-Markham, G. Danezis, K. Fu, P. Shenoy, and D. Irwin. Designing Privacy-Preserving Smart Meters with Low-Cost Microcontrollers. In *FC 2012*, volume 7397 of *LNCS*, pages 239–253, 2012.

[18] T. P. Pedersen. Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In *Advances in Cryptology – CRYPTO '91*, volume 576 of *LNCS*, pages 129–140, 1992.

[19] M. Scott and P. S. L. M. Barreto. Generating more MNT elliptic curves. Cryptology ePrint Archive: Report 2004/058, 2004.

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

Apèndix A

Codi de la implementació

Implementació de la generació de claus

```
public PrivateKey CLKeyGen(int n, String type) {

    System.out.println("Key Generation");
    Pairing pairing = setUpPairinig(type);

    PairingFactory.getInstance().setUsePBCWhenPossible(true);

    Field Zr = pairing.getZr();
    Field G1 = pairing.getG1();
    Field G2 = pairing.getG2();
    Field H = pairing.getGT();

    // System Prams.
    Element g = G1.newRandomElement().getImmutable();
    Element h = H.newRandomElement().getImmutable();

    BigInteger p = Zr.getOrder();

    CLUtils clUtils = new CLUtils();

    ArrayList<Element> zi;
    ArrayList<Element> xy;
    ArrayList<Element> XYg;

    ArrayList<Element> Zi;
```

```
    ArrayList<Element> Wi;

    zi = clUtils.getNElementsFromField(n, Zr);
    xy = clUtils.getxyElements(Zr);
    XYg = clUtils.getXYElements(xy.get(0), xy.get(1), g);
    Zi = clUtils.getZiElements(g, zi);
    Wi = clUtils.getWiElements(XYg.get(1), zi);

    PublicKey pk = new PublicKey(p, G1, G2, H, g, h, pairing,
            XYg.get(0), XYg.get(1), Zi, Wi);

    PrivateKey privk = new PrivateKey(pk, zi, xy.get(0),
            xy.get(1));

    return privk;
}
```

Implementació de la signatura

```
public Signature CLSign(ArrayList<Element> mi,
        PrivateKey privateKey, Field G) {

    Element x = privateKey.getX().getImmutable();
    Element y = privateKey.getY().getImmutable();

    ArrayList<Element> zi = privateKey.getZi();

    Element a = G.newRandomElement().getImmutable();
    ElementPowPreProcessing aPre =
            a.getElementPowPreProcessing();

    Element b = aPre.powZn(y).getImmutable();

    System.out.println("CL Signature");

    ArrayList<Element> Ai = new ArrayList<>();
    ArrayList<Element> Airo = new ArrayList<>();
    ArrayList<Element> Bi = new ArrayList<>();

    Element aElev;
```

```
    Element aiElmnt;
    ElementPowPreProcessing aiElmntPre;

    for (int i = 1; i < mi.size(); i++) {
        aiElmnt = aPre.powZn(zi.get(i)).getImmutable();
        Ai.add(aiElmnt);
        aiElmntPre =
                aiElmnt.getElementPowPreProcessing();
        Bi.add(aiElmntPre.powZn(y));

        aElev = x.mul(y.mul(mi.get(i)));
        Airo.add(aiElmntPre.powZn(aElev));
    }

    aElev = x.add(x.mul(y.mul(mi.get(0))));
    Element ro = aPre.powZn(aElev);

    for (int i = 0; i < Airo.size(); i++) {
        ro = ro.mul(Airo.get(i));
    }

    Signature signature = new Signature(a, b, ro, Ai, Bi);
    return signature;
}
```

Implementació de la verificació

```
public boolean CLVerifySign(PublicKey publicKey,
        ArrayList<Element> mi, Signature signature) {

    System.out.println("CL Verify Signature");
    Pairing e = publicKey.getE();
    PairingPreProcessing ePreR;
    PairingPreProcessing ePreL;

    Element right;
    Element left;

    // Primera validación
    Element a = signature.getA();
```

```
Element b = signature.getB();
Element ro = signature.getSigma();

Element g = publicKey.getgEle();
Element Y = publicKey.getY();
Element X = publicKey.getX();

ePreL = e.getPairingPreProcessingFromElement(Y);
left = ePreL.pairing(a);

ePreR = e.getPairingPreProcessingFromElement(g);
right = ePreR.pairing(b);

if (left.isEqual(right) == false) {
    return false;
}

// Segunda / tercera validación
ArrayList<Element> Ai = signature.getAi();
ArrayList<Element> Bi = signature.getBi();
ArrayList<Element> Zi = publicKey.getZi();

Element ai;
PairingPreProcessing ZiPairPre;

for (int i = 0; i < Ai.size(); i++) {

    ai = Ai.get(i);
    ZiPairPre =
            e.getPairingPreProcessingFromElement(
                    Zi.get(i + 1));
    left = ZiPairPre.pairing(a); //e(Z[i], a)
    right = ePreR.pairing(ai); //e(g, A[i])

    if (left.isEqual(right) == false) {
        return false;
    }

    left = ePreL.pairing(ai); //e(Y, A[i])
    right = ePreR.pairing(Bi.get(i)); //e(g, B[i])
```

```
            if (left.isEqual(right) == false) {
                return false;
            }

        }

        // Cuarta validación
        right = ePreR.pairing(ro);

        ePreL = e.getPairingPreProcessingFromElement(X);
        Element eXb = ePreL.pairing(b).getImmutable();
        ElementPowPreProcessing eXbmi =
                eXb.getElementPowPreProcessing();
        left = ePreL.pairing(a);
        left = left.mul(eXbmi.powZn(mi.get(0)));

        Element mltpl;
        ElementPowPreProcessing mltplPre;

        for (int i = 1; i < mi.size(); i++) {
            mltpl = ePreL.pairing(Bi.get(i - 1)).getImmutable();
            mltplPre = mltpl.getElementPowPreProcessing();

            left = left.mul(mltplPre.powZn(mi.get(i)));
        }

        return left.isEqual(right) != false;
    }
```