UNIVERSITATEA „ALEXANDRU IOAN CUZA" din IAȘI

FACULTATEA de INFORMATICĂ

Curbe Eliptice

LUCRARE DE LICENȚĂ

AUTOR: Alberto-Alexandru CIERI

COORDONATOR: Conf. dr. Sorin IFTENE

DATA: Iulie 2017

DECLARAȚIE PRIVIND ORIGINALITATEA SI RESPECTAREA DREPTURILOR DE AUTOR

Prin prezenta declar că Lucrarea de licență cu titlul “Titlu lucrare” este scrisă de mine și nu a mai fost prezentată niciodată la o altă facultate sau o instituție de învățământ superior din țară sau străinătate. De asemenea, declar că toate sursele utilizate, inclusiv cele preluate de pe Internet, sunt indicate în lucrare, cu respectarea regulilor de evitare a plagiatului:

- toate fragmentele de text reproduse exact, chiar și în traducere proprie din altă limbă, sunt scrise între ghilimele și dețin referința precisă a sursei;
- reformularea în cuvinte proprii a textelor scrise de către alţi autori deţine referinţa precisă;
- codul sursă, imagini etc. preluate din proiecte open-source sau alte surse sunt utilizate cu respectarea drepturilor de autor şi deţin referinţe precise;
- rezumarea ideilor altor autori precizează referinţa precisă la textul original.

Introducere

În aceasta lucrare ne-am propus să facem o prezentare de ansamblu a curbelor eliptice. Această lucrare conține atât suport teoretic, cât și practic. Astfel, ne propunem să întelegem suportul matematic care stă la criptografiei pe curbe eliptice, fară a pierde din vedere aplicațiile practice.

Orice problemă poate fi rezolvată în mai multe feluri, acest lucru fiind reflectat în această lucrare printr-un Studiul Comparativ al unor algoritmi, care au rolul realizării unor operații speciale pe curbe eliptice. Optimizările la acest nivel impactează intr-un mod deosebit aplicații practice des întâlnite, precum protocolul ECDH sau ECDSA. Este foarte important, nu numai să știm care este cel mai rapid algoritm, prin citirea unor de exemplu a unor documentații de specialitate, dar să și înțelegem cum funcționează aceste operații la bază. Fiecare abordare are avantaje și dezavantaje, iar implementarea soluției potrivite necesită luarea unor decizii în cunoștință de cauză, care vine inclusiv de la înțelegerea de la nivel teoretic.

Primul capitol din această lucrare este rezervat algebrei, oferind definiții și teoreme matematice fundamentale curbelor eliptice și nu numai. De exemplu, conceptul de Corpuri Finite, discutate în Secțiunea 3 de la acest capitol, stă la baza întregii Criptografii moderne.

Al doilea capitol introduce conceptele de curbe eliptice, puncte de pe o curbă eliptică și operații cu aceste puncte. Topicul este unul foarte vast, existând foarte multă literatură de specialitate. Noi oferim o privire de ansamblu, concentrându-ne apoi pe un set de curbe și algoritmi care se folosesc în practică.

Capitolul trei prezintă două protocoale foarte folosite, ECDH și ECDSA. Prezentarea problemei Logaritmului Discret aplicată curbelor eliptice, justifică folosirea lor în domeniul securitatea informației. Acestă parte din lucrare este menită să facă tranziția către partea aplicată, oferind însă și câteva idei teoretice importante.

Ultimul capitol este rezervat implementării unui framework în care putem experimenta ideile discutate pâna la acel moment din lucrare. Scopul aplicației este realizarea unui Studiu Comparativ, unde vom discuta despre avantajele sau dezavantajele unor algoritmi, care implementează operații pe curbe eliptice. De asemenea, vom oferi ca funcționalitate în aplicație protocolul ECDSA, având metode pentru generarea de chei, semnare și verificarea semnăturii. Acest protocol este foarte relevant lucrării, înglobând foarte multe din conceptele prezentate.

Table of Contents

CAPITOLUL 1

STRUCTURI ALGEBRICE DE BAZĂ

În această secțiune vom pune bazele matematice pentru urmatoarele capitole, făcând astfel o introducere a unor concepte fundamentale din Teoria Grupurilor, a Corpurilor și în special despre Corpuri finite, care au o importanță deosebită pentru tema acestei licențe. Conceptele din acest capitol au fost preluate din [1] și [2].

1.1 Grupuri

Definiție 1.1. Un *grup* reprezintă o mulțime S împreună cu o operație de compoziție $\cdot$ astfel încât sunt respectate următoarele reguli:
- legea $\cdot$ este *asociativă* $\rightarrow \forall x,y,z \in S : (x \cdot y) \cdot z = x \cdot (y \cdot z)$
- existența unui element *neutru*, unic $\rightarrow \exists e \in S : x \cdot e = x, \forall x \in S$
- $\forall x \in S, \exists y \in S$, numit *invers* al elementului x, astfel încat $x \cdot y = y \cdot x = e$

Definiție 1.2. Se numește grup *abelian*, acel grup în care legea de compoziție $\cdot$ este de asemenea comutativă, adică pentru orice două elemente $x, y \in S$ avem $x \cdot y = y \cdot x$

Definiție 1.3. Fie G un grup și H o submulțime a lui G. H este un *subgrup* a lui G dacă sunt îndeplinite condițiile:
-închidere la operația de compoziție $\rightarrow \forall x,y \in H, x \cdot y \in H$
- dacă $x \in H \Rightarrow x^{-1} \in H, e \in H$

Definiție 1.4. Se numește *ordin* al grupului G, cardinalul mulțimii G, notat $|G|$. Pentru un element $g \in G$, se numește *ordin* al lui g, notat prin $ord_G(g)$, cel mai mic număr natural nenul n, astfel încât $g^n = e$, unde e este element neutru al lui G. Dacă nu există un astfel de număr, spunem că ordinul lui g este infinit.

Definiție 1.5. Fie G un grup și $g \in G$. Se numește subgrup generat de g mulțimea $\{g^n \mid n \in \mathbb{Z}\}$ și se notează $\langle g \rangle$. În cazul în care G este grup finit(conține un număr finit de elemente), $G = \langle g \rangle$ dacă și numai dacă $ord_G(g) = |G|$. Atunci G se numește grup *ciclic* iar g *generator* pentru G.

Definiție 1.6. Fie $n \geq 1$. Notăm cu $\phi(n) = |(\mathbb{Z}/n\mathbb{Z})^*|$. Funcția ϕ se numește *funcția lui Euler*. Avem $\phi(n) = |\{x \mid 1 \leq x \leq n, gcd(x,n) = 1\}|$

Teoremă 1.1. *Fie n,x două numere întregi astfel incât $gcd(n,x) = 1$. Atunci este adevarată relația:* $x^{\phi(n)} \equiv 1(mod n)$

1.2 Inele

Definiție 1.7. Un *inel* reprezintă un triplet $(R,+,\cdot)$, unde R este o mulțime iar $+$ și $\cdot$ reprezintă două legi de compoziție. Următoarele proprietăți trebuiesc simultan îndeplinite:
- $(R,+)$ reprezintă un grup abelian
- $x \cdot y = y \cdot x, \forall x, y \in R$ și de asemenea există element neutru la înmulțire, diferit de elementul neutru la adunare
- legea $\cdot$ este *distributivă* față de $+$, adică $\forall x, y, z \in R, x \cdot (y+z) = x \cdot y + x \cdot z$ și $(y+z) \cdot x = y \cdot x + z \cdot x$

Definiție 1.8. Fie R, R' două inele cu operațiile $+, \times$ respectiv $\oplus, \otimes$. Un *homomorfism* de inele este o funcție $\Psi : R \to R'$ care este definită pentru orice două elemente $x, y \in R$ astfel:
- $\Psi(x+y) = \Psi(x) \oplus \Psi(y)$
- $\Psi(x \times y) = \Psi(x) \otimes \Psi(y)$
- $\Psi(1_R) = 1_{R'}$

Definiție 1.9. Fie R un inel. I Se numește *ideal* (la stânga sau la dreapta) a lui R dacă $I \subseteq R, I \neq \emptyset$ și respectă:
- I este un subgrup pentru $(R,+)$
- $\forall x \in R, \forall y \in I \Rightarrow x \cdot y, y \cdot x \in I$

I se numește ideal *bilateral* dacă este ideal la stânga și la dreapta.
Idealul $I \subsetneq R$ este *prim* dacă $\forall x, y \in R$ cu $x \cdot y \in I$ avem $x \in I \vee y \in I$
Idealul $J \subsetneq R$ este *maximal* dacă pentru oricare alt ideal J, avem $J = I \vee J = R$

Observație 1.1. *Fie Ψ un homomorfism de la $\mathbb{Z}$ la un inel R definit astfel:*

$$\Psi(n) = \begin{cases} 1+\cdots+1 & \textit{de n ori daca } n \geq 0 \\ -(1+\cdots+1) & \textit{de -n ori altfel} \end{cases}$$

Nucleul lui Ψ este un ideal a lui $\mathbb{Z}$ și dacă toți multiplii de 1 sunt diferiți, atunci $ker\Psi = 0$. Altfel, dacă R este finit, de exemplu, câțiva multiplii vor fi 0. Altfel spus, nucleul lui Ψ este generat de un numar natural m

Definiție 1.10. Fie R un inel și Ψ definit ca mai sus. Nucleul lui Ψ are forma $m\mathbb{Z}, m \in \mathbb{N}$. Elementul poartă denumirea de *caracteristică* a inelului R și se notează $char(R)$

Definiție 1.11. Fie R un inel. Un element $x \in R$ este inversabil daca $\exists! y \in R, x \cdot y = y \cdot x = e$. y se numeste unitate. Multimea tuturor unitatilor se noteaza cu R^*

1.3 Corpuri

Definiție 1.12. Un *corp* K este un inel comutativ cu toate elementele diferite de 0 inversabile.

Exemplu 1.1. *Mulțimea numerelor raționale $\mathbb{Q}$ împreună cu operațiile obișnuite de adunare si înmulțire este un corp. Pentru orice număr prim p, Z_p, împreună cu operațiile modulo p, este corp.*

Propoziție 1.1. *Caracteristica unui corp este* 0 *sau p, un număr prim.*

Propoziție 1.2. *Fie R un inel și I un ideal. Inelul factor R/I este corp dacă și numai dacă I este maximal.*

Definiție 1.13. Fie K, K' două corpuri. Un homomorfism de corpuri este un homomorfism de inele între K si K'. Remarcăm faptul că o astfel de funcție este tot timpul injectivă, deoarece nucleul acesteia este $\{0\}$

Definiție 1.14. Fie L un corp. Un *subcorp* a lui L este o submulțime K a lui L care își păstrează proprietatea de corp, operațiile aditive respectiv multiplicative fiind moștenite de la L. În această situație, corpul mare, L, se numește o *extensie* a corpului K.

1.3.1 Corpuri Finite

În sistemele criptografice bazate pe curbe eliptice, este important să implementăm eficient operații pe corpuri finite. Trei tipuri de corpuri finite reprezintă candidați potriviți pentru implementarea acestor operații, respectiv corpurile prime, corpurile binare si corpurile de extensie optimale. În randurile care urmează vom introduce pe rând aceste concepte.

Definiție 1.15. Un corp finit este un corp, conform definiției 1.12, care are mulțimea elementelor finită. Pe lângă cele două operații aritmetice de bază, putem defini scăderea și împărțirea pe baza operațiilor de adunare si înmulțire. Astfel avem, pentru un corp K: $a, b \in K, a - b = a + (-b)$, unde $-b$ este inversul aditiv a lui b și respectiv $a, b \in K, a/b = a \times b^{-1}$, unde b^{-1} este inversul multiplicativ a lui b, garantat să existe într-un corp.

Teoremă 1.2. *Fie q ordinul unui corp finit $\mathbb{F}$. Există și este unic(până la un isomorfism) un astfel de corp, dacă și numai dacă $q = p^m$, unde p este caracteristica corpului $\mathbb{F}$. Dacă $m = 1$, atunci corpul este prim, iar daca $p \geq 2$ este corp de extensie.*

Definiție 1.16. Fie p un număr prim. Numerele naturale în mulțimea $\{0, 1, 2, ..., p-1\}$ împreună cu operațiile de înmulțire și adunare modulo p formează un corp finit cu ordinul p. Vom nota acest corp cu F_p. Pentru orice numar $a \in \mathbb{Z}$, $a \bmod p$ reprezintă restul împarțirii lui a la p, număr unic în intervalul $[0, p-1]$. Această operație mai poartă denumirea de reducție modulară.

Definiție 1.17. Corpurile binare au caracteristica 2 și deci ordinul 2^m. O metodă de a construi F_{2^m} este considerarea fiecărui element ca un polinom cu coeficienți 0 sau 1 și cu gradul cel mult $m-1$. Avem astfel : $F_{2^m} = \{a_{m-1}z^{m-1} + a_{m-2}z^{m-2} + ... + a_1 z + a_0 \mid a_i \in 0, 1\}$

Capitolul 2

Curbe Eliptice

În aceast capitol vom prezenta conceptul de curbă eliptică, pornind de la definițiile de bază, fiind incluse concepte precum structura de grup formată de punctele de pe o curbă eliptică, construcția curbelor eliptice, reprezentări ale punctelor de pe o curbă. Vom continua discuția spre aritmetica eficientă, unde vom prezenta algorimi eficienți pentru înmulțirea unui punct cu un scalar și pentru înmulțire multiplă, prezentând câte un exemplu la fiecare algoritm.

2.1 Introducere

Definiție 2.1. Conform [3], definim o *curbă eliptică* E peste un corp K prin ecuația:

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

unde $a_1, a_2, a_3, a_4, a_5, a_6 \in K$, iar discriminantul ecuației, $\Delta \neq 0$. Discriminantul ecuației este definit astfel:

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

Definiție 2.2. Fie L orice extensie a corpului K. Definim mulțimea de L-puncte raționale peste E astfel: $E(L) = \{(x,y) \in L \times L : y^2 + a_1xy + a_3y - x^3 - a_2x^2 - a_4x - a_6 = 0\} \cup \{\infty\}$

Observație 2.1. *În următoarele rânduri voi face o serie de observații asupra ecuației unei curbe eliptice:*
(i) Ecuația de la definiția 2.1 se numește Ecuație Weierstrass
(ii) Condiția ca discriminantul Δ să fie diferit de 0, asigură "netezimea" curbei eliptice, adică nu există puncte care să aibe 2 sau mai multe tangente diferite la curbă.
(iii) L-punctele raționale sunt acele puncte, (x,y), care satisfac ecuația Weierstrass, cu $x,y \in L$. Punctul de la infinit este considerat un punct L-rațional pentru toate extensiile L ale corpului K

Definiție 2.3. Fie E_1,E_2 două curbe eliptice, definite astfel:
$E_1 : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$
$E_2 : y^2 + \overline{a_1}xy + \overline{a_3}y = x^3 + \overline{a_2}x^2 + \overline{a_4}x + \overline{a_6}$
Spunem că cele două curbe sunt *izomorfe* dacă există $u,r,s,t \in K, u \neq 0$ astfel încât schimbarea de variabilă $(x,y) \to (u^2x + r, u^3y + u^2sx + t)$ transformă ecuația E_1 în ecuația E_2. Acest tip de transformare se numește schimbare "admisibilă" de variabilă.

Definiție 2.4. Ecuația Weierstrass a unei curbe eliptice poate fi simplificată în mod considerabil, aplicând schimbări admisibile de variabilă. Vom trata trei cazuri separate de schimbări de variabilă, în funcție de caracteristica corpului K, ajungând la o formă *simplificată a Ecuației Weierstrass.*. Vom aborda trei cazuri, primul fiind $char(K) \neq \{2,3\}$

1. Fie K un corp si E o curbă eliptică dată prin ecuația lui Weierstrass. O schimbare admisibilă de variabilă este: $(x,y) \to (\frac{x-3a_1^2-12a_2}{36}, \frac{y-3a_1x}{216} - \frac{a_1^3+4a_1a_2-12a_3}{24})$
Această schimbare transformă ecuația E în ecuația

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

Discriminantul acestei noi ecuații este $\Delta = -16(4a^3 + 27b^2)$

2. Dacă $char(K) = 2$ trebuie să considerăm două subcazuri. Dacă $a_1 \neq 0$, atunci o schimbare admisibilă de variabilă este: $(x,y) \to (a_1^2x + \frac{a_3}{a_1}, a_1^3y + \frac{a_1^2a_4+a_3^2}{a_1^3})$, care transformă E în:

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

O astfel de ecuație se numește *non-supersingulară* și are discriminantul $\Delta = b$
Dacă $a_1 = 0$ atunci o schimbare admisibilă ar fi $(x,y) \to (x + a_2, y)$ care transformă curba E in

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

O astfel de ecuație se numește *supersingulară* și are discriminantul $\Delta = c^4$

3. Dacă $char(K) = 3$ trebuie să considerăm din nou două subcazuri. Dacă $a_1^2 \neq -a_2$, atunci o schimbare admisibilă de variabilă este $(x,y) \to (x + \frac{\alpha}{\beta}, y + a_1 x + a_1 \frac{\alpha}{\beta} + a_3)$, unde $\alpha = a_4 - a_1 a_3$ și $\beta = a_1^2 - a_2$. Ecuația E devine:

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

O astfel de ecuație se numește *non-supersingulară* și are discriminantul $\Delta = -a^3 b$
Dacă $a_1^2 = -a_2$, atunci considerăm o schimbare admisibilă de variabilă $(x,y) \to (x, y + a_1 x + a_3)$, care transformă curba E în:

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

O astfel de curbă este *supersingulară* și are discriminantul $\Delta = -a^3$

Observație 2.2. *Vom lucra cu forma simplificată a ecuației Weierstrass pe tot parcursul următoarelor capitole.*

2.1.1 Grupul punctelor de pe o curbă eliptică

Fie E o curbă eliptică în formă Weierstrass peste un corp F_q. Punctele care aparțin acestei curbe formează o structură de grup abelian, acestea respectând regulile unei astfel de structuri.

- Definim elementul neutru în grup ca fiind punctul de la infinit, notat ∞. Astfel, pentru orice punct P de pe curbă, avem: $P + \infty = \infty + P$

- Fie $P(x,y) \in E(F_q)$. Atunci există $-P = (x,-y) \in E(F_q)$ astfel încât $P + (-P) = \infty$. Numim $-P$, inversul punctului P. De asemenea, avem $\infty = -\infty$

- Oricare ar fi două puncte, $P, Q \in E(F_q)$, avem $P + Q \in E(F_q)$. În continuare vom defini această operație de adunare mai în detaliu.

Definiție 2.5. Adunarea a două puncte de pe o curbă eliptică este foarte intuitivă din punct de vedere geometric. Fie P,Q două puncte și R suma lor. Rezultatul este obținut astfel. Mai întâi desenăm o linie între P,Q. Acestă linie intersectează curba într-un al treilea punct. Punctul R este reflecția la axa Ox a acestui punct(Figura 2.1a). Dublul unui punct $P(2P = R)$ este definit astfel. Desenăm o tangentă la curba eliptică în P, aceasta intersectând curba într-un punct secundar. Punctul R este din nou reflecția la axa Ox(Figura 2.1b).

Observație 2.3. *Formulele algebrice pentru adunarea a două puncte diferă în funcție de sistemul de coordonate folosit, sau tipul de corp algebric peste care este definită curba eliptică (corp prim, binar sau de extensie).*

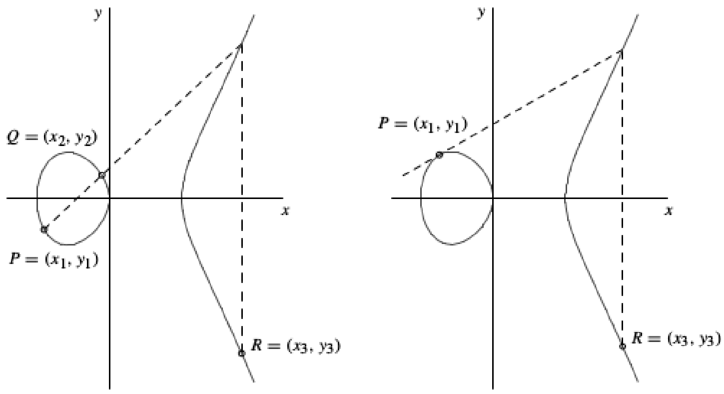

Figura 2.1: Adunarea si dublarea unui punct pe o curba eliptica

Definiție 2.6. Pentru punctele reprezentate prin coordonate afine, formulele de calcul sunt după cum urmează. Fie două puncte, $P(x_1, y_1), Q(x_2, y_2) \in E(F_p)$. Notăm cu $R(x_3, y_3) = P + Q$. Formulele pentru adunarea a două puncte pot fi demonstrate matematic destul de ușor, pornind de la ideea ca P,Q și simetricul rezultatului față de axa Ox se află pe aceași dreaptă, respectiv pe aceași curba eliptică.

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

cu

$$\lambda = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1}, P \neq Q \\ \frac{3x_1^2 + a}{2y_1}, P = Q \end{cases}$$

Demonstrație 2.1. *Fie $P(x_1, y_1), Q(x_2, y_2) \in E(F_p)$ Punctele se află pe aceași dreaptă. Scriind ecuația dreptei care trece prin cele două puncte și considerând că $-R \in E(F_p)$,* *rezolvăm sistemul de ecuații:* $\begin{cases} 0 = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x & y & 1 \end{vmatrix} \\ y^2 = x^3 + ax + b \end{cases}$ *Panta dreptei este* $\lambda = \begin{cases} \frac{y_2-y_1}{x_2-x_1}, P \neq Q \\ \frac{3x_1^2+a}{2y_1}, P = Q \end{cases}$

Înlocuind în formula curbei eliptice, obținem formulele din definiția precedentă.

2.1.2 Construcția curbelor eliptice

Fie E o curbă eliptică peste un corp $K = F_q$. Așa cum am văzut în secțiunea anterioară, mulțimea de puncte $E(F_q)$ împreună cu operația de adunare, formează o structură de grup abelian, cu punctul de la infinit fiind elementul neutru din grup. Acest grup este folosit în criptografia bazată pe curbe eliptice.

Definiție 2.7. Numim *ordin* al unei curbe eliptice, numărul de puncte care satisfac ecuația Weierstrass, cu alte cuvinte, cardinalul mulțimii $E(F_q)$ și il notăm cu $\#E(F_q)$. Teorema care urmează, datorată lui *Hasse*, oferă o aproximare pentru acest ordin

Teoremă 2.1. *Fie E o curbă eliptică definită peste F_q. Atunci este adevărată relația:*

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

Întrucât $2\sqrt{q}$ este relativ mic în comparație cu q, putem afirma că $\#E(F_q) \approx q$

Teoremă 2.2. *Fie $q = p^m$, unde p este caracteristica corpului F_q. Atunci există o curbă eliptică E definită peste acest corp, cu $\#E(F_q) = q + 1 - t$ (t se numește urma curbei eliptice E) dacă și numai dacă una dintre următoarele condiții este adevărată:*
(i) $t \not\equiv 0$ mod p si $t^2 \leq 4q$
(ii) m este impar și $t = 0$ sau $t^2 = 2q$ și $p = 2$ sau $t^2 = 3q$ și $p = 3$
(iii) m este par și $t^2 = 4q$ sau $t^2 = q$ și $p \not\equiv 1$ mod 3 sau $t = 0$ și $p \not\equiv 1$ mod 4

Definiție 2.8. Fie p caracteristica corpului F_q. Numim curbă eliptică *supersingulară*, dacă p divide t, unde t este urma curbei. Altfel, curba E este *non-supersingulară*

Teoremă 2.3. *Fie E o curbă eliptică peste corpul F_q și fie ordinul acesteia $\#E(F_q) = q + 1 - t$. Atunci, $\#E(F_q) = q + 1 - V_n$, unde definim șirul $\{V_n\}$ recursiv, prin formula $V_0 = 2, V_1 = t$ și $V_n = V_1 V_{n-1} - q V_{n-2}, \forall n \geq 2$*

Următoarea teoremă, descrie structura grupului pentru o curbă eliptică. Vom nota un grup ciclic de ordin n, cu Z_n.

Teoremă 2.4. *Fie E o curbă eliptică definită peste F_q. Atunci grupul $E(F_q)$ este izomorfic cu $Z_{n_1} \oplus Z_{n_2}$, unde $n_1, n_2 \in Z$, unici determinati, cu n_2 care divide atât n_1 cât și $q-1$. Dacă $n_2 = 1$, spunem ca $E(F_q)$ este grup ciclic.*

2.1.3 Reprezentări ale punctelor de pe o curbă eliptică

De multe ori, în efectuarea operațiilor pe curbe eliptice, poate fi avantajos să reprezentăm un punct, în alte coordonate decât cele afine. De exemplu, în calculul sumei a două puncte (operație care la rândul ei este folosită în algoritmii pentru înmulțirea cu un scalar și în înmulțirea multiplă) se observă necesitatea de a efectua operația de invers modular de mai multe ori. Acestă operație este costisitoare din punct de vedere computational, astfel vom folosi diferite tipuri de reprezentări.

Definiție 2.9. Fie K un corp și $c, d \in \mathbb{N}$. Definim o relație de echivalență peste mulțimea $K^3 \setminus \{(0,0,0)\}$ ca fiind :

$$(X_1, Y_1, Z_1) \equiv (X_2, Y_2, Z_2) \text{ dacă } X_1 = \lambda^c X_2, Y_1 = \lambda^d Y_2, Z_1 = \lambda Z_2, \lambda \in K^*$$

Există o corespondență $1-1$ între mulțimea de puncte în coordonate proiective $P(K)^* = \{(X, Y, Z) : X, Y, Z \in K, Z \neq 0\}$ și mulțimea punctelor în coordonate afine, $A(K) = \{(x, y) : x, y \in K\}$

Definiție 2.10. Considerăm $c = 1, d = 1$ în definiția *coordonatelor proiective*. Acestea se numesc *coordonate proiective standard*. Punctul l în coordonate proiective $(X, Y, Z), Z \neq 0$ corespunde punctului în coordonate afine $(X/Z, Y/Z)$. Ecuația curbei eliptice devine:

$$Y^2 Z = X^3 + aXZ^2 + bZ^3$$

Punctul de la infinit este $(0,1,0)$ în timp ce inversul unui punct oarecare este $(X, -Y, Z)$

Definiție 2.11. Considerăm $c = 2, d = 3$. Acest sistem de coordonate se numesc coordonate *proiective Jacobi*. Punctul $(X, Y, Z), Z \neq 0$ corespunde punctului în coordonate afine $(X/Z^2, Y/Z^3)$. Ecuația curbei devine:

$$Y^2 = X^3 + aXZ^4 + bZ^6$$

Punctul de la infinit este $(1,1,0)$, iar inversul unui punct este $(X, -Y, Z)$

Definiție 2.12. *Coordonatele Chudnovsky* sunt obținute din coordonatele Jacobi, adăugând niște redundanțe. Astfel, un punct reprezentat în acest tip de coordonate, arată în acest fel: (X, Y, Z, Z^2, Z^3). Acest tip de reprezentare aduce îmbunătățiri de performanță când folosim anumiți algoritmi specializați pentru înmulțirea cu un scalar.

Observație 2.4. *Se poate folosi $Z = 1$ in implementări pentru a simplifica calculele.*

2.2 Aritmetică eficientă

În această secțiune vom face o prezentare a metodelor eficiente de înmulțire cu un scalar și de înmulțire multiplă. Printre metodele de înmulțire cu un scalar, se numără algoritmul binar, algoritmul care folosește o reprezentare cu semn a scalarului și diferite metode de înmulțire cu fereastră. La înmultirea multiplă, vom discuta comparativ metoda naivă și două metode de înmultire mult mai eficiente.

2.2.1 Inmulțirea cu un scalar

Definiție 2.13. Definim operația de înmulțire a unui punct P de pe o curbă eliptică, cu un scalar, $k \in \mathbb{Z}$, notată cu kP, ca fiind:
- $0P = \infty$
- $(k+1)P = kP + P, k \geq 0$
- $kP = -|k|P, k < 0$

Există multe metode de a face acest lucru, de la metoda *brute force*, care face k adunări repetate ($kP = P + P + \cdots + P$) până la metode mai rafinate, precum cea a ferestrei glisante, care îmbunătățesc considerabil performanța operației. Vom discuta în continuare despre diferite metode eficiente pentru această operație.

2.2.1.1 Metoda Binară

Metoda naivă de a înmulți un punct cu un scalar, cea prezentată in definiție, necesită $k-1$ adunări pentru a calcula kP. O primă optimizare este dată de această metoda binară, care necesită cel mult m adunări [4] și în medie $m/2$ adunări, unde m este numărul de biți din reprezentarea lui k. Algoritmul constă în procesarea de la dreapta la stânga, sau de la stânga la dreapta a scalarului. După fiecare bit parcurs, dublăm rezultatul și când întâlnim un bit de 1 adunăm punctul P la rezultat. Astfel, după parcurgerea a $\log_2 k$ iterații(numărul de biți din reprezentarea lui k) avem rezultatul.

Exemplu 2.1. *Fie curba eliptică $E : y^2 = x^3 + 3x + 2$, punctul $P(10,16) \in E$ și un scalar $d = 5$. Reprezentarea binară a scalarului este $d = (1,0,1)$. Așadar rezultatul va fi egal cu: $r = (10,16) + 4 * (10,16)$. Am adunat la rezultat P, deoarece cel mai nesemnificativ bit este 1, apoi am dublat P de 2 ori(2 biți parcurși) și l-am adunat la rezultat deoarece bitul cel mai semnificativ este 1. Astfel obținem $5P = (66,73)$*

2.2.1.2 Reprezentări cu semn

Un avantaj major al curbelor eliptice, este faptul că în grupul format de punctele de pe o astfel de curbă, calculul inversului se poate realiza foarte eficient. Astfel, dacă consideram 2 puncte P, Q într-un astfel de grup, putem calcula $P+Q$ si $P-Q$ cu aproximativ același cost computațional. Această observație este foarte importantă în eficientizarea algoritmilor de înmulțire cu un scalar. Calculul inversului, având un cost neglijabil din punct de vedere computațional, nu este necesar să ne limităm la $\{0,1\}$ în reprezentarea unui scalar. Introducem astfel, conceptul de reprezentare cu semn.

Definiție 2.14. Conform, [5], o reprezentare a lui $n \in \mathbb{N}$, cu semn, poate fi notată astfel: $n = < u_{l-1}u_{l-2}...u_1u_0 >$, unde $u_i \in \{0,-1,1\}$. Astfel $n = \sum_{i=0}^{l-1} u_i 2^i$. Există o infinitate de astfel de reprezentări.

Definiție 2.15. Reprezentarea cu semn optimă pentru operația de înmulțire cu un scalar este așa numita NAF, sau Non Adjacent Form. Aceasta are proprietatea ca nu există două elemente consecutive din reprezentarea cu semn diferite de 0.

Teoremă 2.5. *NAF-ul are urmatoarele proprietati:*

- *Orice număr natural are un NAF*
- *NAF-ul unui număr este unic*
- *Lungimea NAF-ului unui număr natural este cu cel mult o unitate mai mare decât expansiunea binară a numarului*
- *NAF-ul are distanța Hamming minimă dintre toate reprezentările cu semn*

Un algoritm care folosește această reprezentare a scalarului, funcționează asemănător cu cel de la metoda binară. Parcurgem reprezentarea cu semn, dublăm rezultatul după fiecare bit parcurs, adunăm P când întâlnim un bit de 1 și scădem P când întâlnim un bit de -1. Această metodă necesită în medie $m/3$ adunări și m dublări, unde m reprezintă numărul de biți din reprezentarea scalarului [6].

Exemplu 2.2. *Fie curba eliptică $E: y^2 = x^3 + 3x + 2$, punctul $P(10,16) \in E$ și un scalar $d = 5$. Reprezentarea cu semn a lui d este, $d = (1,0,0,-1)$. Parcurgem de la dreapta la stânga rezultatul și avem după prima iterație $rezultat = P$. După a doua iterație $resultat = 2P$. Dupa a treia iterație $rezultat = 4P$, iar la ultima iterație, $rezultat = 8P - P = 7P$. Așadar $7P = (14,13)$*

2.2.1.3 Metoda cu fereastră

Algoritmii care se folosesc de reprezentarea cu semn a scalarului pot fi îmbunătățiți dacă avem disponibilă mai multă memorie. Vom procesa w cifre din scalar la o iterație, unde w reprezintă lățimea ferestrei.

Definiție 2.16. Fie $w \geq 2$ un număr natural. Numim o reprezentare NAF de lățime w pentru un scalar $n \in \mathbb{N}$, notată cu $w-NAF$, un șir de numere $n =< u_{l-1}u_{l-2}...u_1u_0 >$ astfel încât $|u_i| < 2^{w-1}$ și $n = \sum_0^{l-1} u_i 2^i$ astfel încât cel mult una din w cifre consecutive din șir este diferită de 0.

Teoremă 2.6. *Următoarele proprietăți sunt adevarate pentru $w-NAF$:*

- $2-NAF = NAF$ *pentru orice număr* $k \in \mathbb{N}$
- $w-NAF$ *unui număr este unic*
- *Reprezentarea $w-NAF$ este cu cel mult un bit mai mare decât reprezentarea binară a unui număr*
- *Densitatea medie a cifrelor diferite de 0 din reprezentarea $w-NAF$ de lungime l a unui număr este* $\frac{1}{w+1}$

Algoritmul care calculează nP folosind această metodă, este asemănător cu algoritmii precedenți, dar folosim această reprezentare $w-NAF$ pentru scalar. Fie $n =< u_{l-1}u_{l-2}...u_1u_0 >$ reprezentarea $w-NAF$ a unui număr natural n. Dorim să calculam nP. Algoritmul nostru parcurge fiecare număr din reprezentarea $w-NAF$, dublând rezultatul după fiecare iterație, adunând sau scăzând u_iP la rezultat, $i \in \{0, l-1\}$. Valoarea lui u_iP este precalculată și stocată.

Conform teoremei 2.6, numărul mediu de adunări și dublări pentru algoritmul prezentat mai sus este $[1D + (2^{w-2} - 1)A] + [\frac{m}{w+1}A + mD]$, unde $m = \log_2 n$, A reprezintă adunare și D dublare. [7]

Exemplu 2.3. *Fie curba eliptica $E : y^2 = x^3 + 3x + 2$, punctul $P(10,16) \in E$, un scalar $d = 39$ și lățimea ferestrei $w = 3$. Reprezentarea $3-NAF$ pentru d este $(1,0,0,-3,0,0,-1)$. Vom precalcula și stoca valorile pentru $(P,3P)$, iar apoi parcurgem $3-NAF$ de la stânga la dreapta. Inițializăm variabila rezultat cu "punctul de la infinit", iar după fiecare iteratie, în rezultat vom avea: $P, 2P, 4P, 5P, 10P, 20P, 39P$. După fiecare bit parcurs, am dublat rezultatul, am scăzut $-3P$ la iterația 4 și $-P$ la ultima iterație. Ambele valori erau precalculate. La final, avem $rezultat = 39P = (60,39)$*

2.2.1.4 Metoda cu fereastră glisantă

Pentru a eficientiza metoda cu fereastră fixă prezentată mai sus, vom folosi o așa zisă "fereastră glisantă" asupra cifrelor din $w-NAF$. Ideea e să folosim o fereastră de lățime w pe care o "glisăm" (sărim peste cifrele consecutive de 0) peste reprezentarea scalarului. Menținem întotdeauna o valoare impară în fereastră pentru a micșora numărul de precalculari necasar. La fiecare iterație, când găsim o cifra diferită de 0 în $w-NAF$, căutăm cel mai mare $t \leq w$ astfel încât valoarea numărului dat de $w-NAF$ de lungime t este impară. Adunăm sau scădem la rezultat acea valoare apoi "glisăm" fereastra la dreapta cu t poziții.

Observație 2.5. *Conform [8], numărul mediu de zerouri între ferestre este egal cu:*

$$\nu(w) = \frac{4}{3} - \frac{(-1)^w}{3 \times 2^{w-2}}$$

Astfel, numărul mediu de adunări și dublări al algoritmului cu fereastră glisantă este:

$$[1D + (\frac{2^w - (-1)^w}{3} - 1)A] + \frac{m}{w + \nu(w)}A + mD$$

Exemplu 2.4. *Vom folosi același exemplu ca la Metoda cu fereastră fixă, aceași curbă, aceeași lățime a ferestrei, același punct și același scalar. Algoritmul îl aplicam de la stânga la dreapta pe $3-NAF$-ul punctului P. La fiecare iterație avem nevoie de o variabilă t care dă dimensiunea ferestrei, și u, valoarea în fereastră. După fiecare iterație, vom avea: $s = 1, u = 1, resultat = P; resultat = 2P; resultat = 4P; s = 4, u = -3, resultat = 5P; resultat = 10P; resultat = 20P; s = 6, u = -1, resultat = 39P$. La final, avem $39P = (60, 39)$*

2.2.2 Înmulțirea multiplă

Definiție 2.17. O operație asemănătoare celei de înmulțire cu un scalar, este cea de înmulțire multiplă cu scalari. Fie doua puncte, P, Q de pe o curbă eliptică și doi scalari, $k, l \in \mathbb{Z}$. Dorim să aflăm rezultatul $kP + lQ$. Evident, similar cu operația de înmulțire cu un scalar, putem aplica o metodă directă de a înmulți punctul P cu scalarul k respectiv punctul Q cu scalarul l și apoi facem o adunare de puncte. Acest lucru este însă ineficient, întrucât există și aici metode mai rapide de a calcula acest lucru. Această operație de înmulțire rapidă, este una extrem de folosită în criptografia pe curbe eliptice, facându-și apariția, de exemplu, în cadrul unor protocoale criptografice, precum ECDSA, iar implementarea ineficientă a acesteia duce la mari probleme de performanță.

2.2.2.1 Metoda naivă

O primă metodă de a aborda calculul $kP + lQ$ se bazează pe metode deja discutate. Calculăm separat kP si lQ, folosind unii din algoritmii precizați la secțiunea anterioară, apoi adunăm rezultatele. Această metodă este ineficientă, făcând multe adunări și dublări redundante.

Exemplu 2.5. *Fie doua puncte,* $P(10,16), Q(14,13) \in E$*, unde* $E : y^2 = x^3 + 3x + 2$*. Căutăm rezultatul* $5P + 6Q$*. Folosim metoda binara pentru a gasi resultatele* $5P = (66,73)$ *și* $6Q = (89,57)$*. Apoi facem adunarea* $(66,73) + (89,57) = (36,20)$

2.2.2.2 Reprezentarea Joint Sparse Form

Definiție 2.18. Considerăm 2 reprezentări cu semn pentru scalarii care apar în calculul $kP + lQ$. Combinăm aceste 2 reprezentări într-o singură reprezentare, pe care o vom numi reprezentare reunită. Astfel, dacă considerăm 2 reprezentări cu semn pentru k, p, ca fiind $< u_{l-1}u_{l-2}...u_1u_0 >$ respectiv $< v_{l-1}v_{l-2}...v_1v_0 >$, reprezentarea lor reunită este $(< u_{l-1}u_{l-2}...u_1u_0 >, < v_{l-1}v_{l-2}...v_1v_0 >)$. Există multe astfel de reprezentări, dar cea mai eficientă este așa numita Joint Sparse Form, introdusă de [5], care are următoarele proprietăți:

- Pentru oricare 3 poziții consecutive, cel puțin una este $0,0$. Altfel spus, oricare ar fi $i, j \in \mathbb{N}$ avem $u_{i,j+k} = u_{1-i,j+k} = 0, k = 0, \pm 1$
- Termeni adiacenți nu pot avea semne diferite. Astfel, nu putem avea egalitatea $u_{i,j+1}u_{i,j} = -1$
- Daca $u_{i,j+1}u_{i,j} \neq 0 \Rightarrow u_{1-i,j+1} = \pm 1$ si $u_{1-i,j} = 0$

Exemplu 2.6. *Fie scalarii* $21, 26$*. Reprezentarea lor Joint Sparse Form este dată de:*

$$< 1, 0, -1, 0, -1, -1 >$$

$$< 1, 0, -1, 0, 1, 0 >$$

Algoritmul care folosește această reprezentare se numește *"Shamir's Trick"* și calculează $kP + lQ$ astfel. Precalculează $P, Q, -P, -Q, P + Q, P - Q, -P - Q$ și în funcție de cifrele care apar în parcurgerea reprezentării Joint Sparse, adaugă la rezultat una din valori. De exemplu dacă la o iterație avem combinația $1,1$ adăugăm $P + Q$, dacă avem $-1, -1$ adăugăm $-P - Q$, etc. În continuare vom prezenta un exemplu de calcul cu acest algoritm.

Exemplu 2.7. *Fie curba eliptică și punctele de la exemplul precedent, dar de această dată vom alege scalarii* $d = 21, l = 26$. *Vrem așadar să calculăm rezultatul* $dP + lQ$ *folosind metoda Joint Sparse Form. Mai întâi trebuie sa calculăm JSF pentru cei doi scalari, rezultatul fiind* $(1,0,-1,0,-1,-1),(1,0,-1,0,1,0)$. *Vom crea o variabilă* $rezultat$ *în care, la fiecare iterație vom avea următoarele:* $rezultat = P+Q; rezultat = 2P+2Q; rezultat = 3P+3Q; rezultat = 6P+6Q; rezultat = 11P+13Q; rezultat = 21P+26P = (48,35)$.

2.2.2.3 Metoda cu fereastră intercalată, w-NAF

Spre deosibire de metada precedentă de calcul, ideea aici este să facem pentru fiecare punct în parte precalculul separat, dar pasul de dublare trebuie făcut simultan. Putem folosi ferestre de dimensiuni diferite pentru fiecare scalar în parte, calculând $w-NAF$ lor. În faza de precalcul, stocăm pentru fiecare punct $iP, i < 2^{w-1}$, i impar. Reprezentările $w-NAF$ ale punctelor sunt procesate simultan de la stânga la dreapta, cu o singură variabilă Q pe post de rezultat, care se dublează la fiecare iterație. Putem vizualiza ce se întâmplă cu această variabilă la fiecare iterație în figura 2.2.

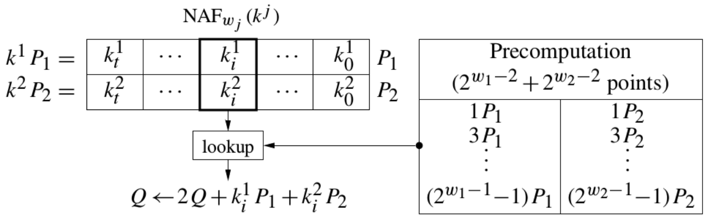

Figura 2.2: Variabila rezultat la o iterație oarecare

Observație 2.6. *Conform [9] Numărul mediu de adunari si dublări este:*

$$[|\{j : w_j > 2\}|D + \sum_{j=1}^{2}(2^{w_j-2} - 1)A] + [\max_{1\le j\le 2} l_j D + \sum_{j=1}^{2}\frac{l_j}{w_j+1}A]$$

Exemplu 2.8. *Păstrând curba și punctele de la exemplele precedente, dorim să calculăm* $10P + 41Q$, *cu ferestrele* $4,4$. *Trebuie așadar să calculam* $4-NAF$ *pentru cei doi scalari. Obținem* $(5,0),(3,0,0,0,-7)$. *Padăm cu 0 astfel încât cele două reprezentări să aibă lungimi egale, obținând astfel* $(0,0,0,5,0),(3,0,0,0,-7)$. *După fiecare iterație, în variabila rezultat avem:* $rezultat = 3Q; rezultat = 6Q, rezultat = 12Q, rezultat = 5P+24Q; rezultat = 10P+41Q = (8,21)$.

CAPITOLUL 3

APLICAȚII ÎN CRIPTOGRAFIE

În acest capitol vom prezenta o imagine de ansamblu asupra aplicațiilor curbelor eliptice, motivând în același timp suportul matematic pe care acestea se bazează, prin descrierea problemei logaritmului discret. Printre aplicațiile curbelor eliptice se numară construcția criptosistemelor cu chei publice, construirea de generatoare pseudoaleatoare de biți, semnaturi digitale, sisteme bazate pe identitate. Ne vom concentra pe protocolul de schimbul de chei, Diffie-Hellman, bazat pe curbe eliptice(*ECDH*) și protocolul de semnătură digitală, *ECDSA*.

3.1 Problema Logaritmului Discret Pentru Curbe Eliptice

Definiție 3.1. [10] Fie E o curbă eliptică peste un corp finit F_p, dată in formă Weierstrass simplificată, $E: y^2 = x^3 + ax + b(mod p)$ si 2 puncte $S, T \in E(F_p)$. Problema logarimului discret constă în aflarea unui număr, $k = \log_T S \in \mathbb{Z}$, sau $k \equiv \log_T S(mod p)$ astfel încât $S = kT \in E(F_p)$ sau $S \equiv kT(mod p)$.

Problema logaritmului discret pentru curbe eliptice(*ECDLP*) alese bine, este mai dificilă decât problema clasică a logaritmului discret(*DLP*) pentru grupuri multiplicative peste un corp finit. Dificultatea acestei probleme stă la baza criptografiei pe curbe eliptice. Parametrii curbei eliptice trebuie aleși astfel încat să evităm orice atac cunoscut asupra ECDLP. Un algoritm naiv este căutarea exaustivă (calculăm $T, 2T, 3T, \ldots$ până obținem

S). În cazul cel mai nefavorabil algoritmul parcurge p pași, iar în cazul mediu $p/2$. Așadar putem evita acest atac alegând un p suficient de mare, de exemplu $p \geq 2^{80}$. În cazul atacurilor Pohlig-Hellman si Pollard's rho, care au complexitatea $\mathcal{O}(\sqrt{\alpha})$, unde α este cel mai mare divizor prim a lui p. Pentru a contracara acest atac avem de asemenea nevoie să alegem p mare, de exemplu $p \geq 2^{160}$. Dacă restul parametrilor sunt aleși pentru a evita si atacuri de tip izomorfic(Weil, Tate), ECDLP este o problemă intractabilă. [11]

3.2 ECDH

Diffie-Hellman pe curbe eliptice este un protocol care dă posibilitatea stabilirii unui secret comun între doua entități. Acest secret poate fi o cheie sau poate fi folosit pentru a deriva o cheie, care ulterior poate fi folosită într-un protocol simetric de criptografie.
Vom numi cele două entități care participă la comunicare Alice și Bob. Conform [12], *ECDH* se desfășoară în următoarele etape:

- Se stabilesc parametrii curbei eliptice, punctul generator. De obicei aici, curbele alese pentru protocol, sunt cele care au fost verificate de către o autoritate de încredere, precum *NIST*.
- Alice alege un număr natural, random, în intervalul $[1, n-1]$, unde n este ordinul subgrupului generat de punctul generator, G. Apoi calculeaza $Q_A = aG$ si trimite Q_A lui Bob. Similar, Bob alege $b \in [1, n-1]$ si trimite lui Alice $Q_B = bG$.
- Atât Alice cât si Bob, calculează abG, care reprezintă secretul între cele două entități.

Observație 3.1. *Pentru ca un atacator să afle* abG*, trebuie sa afle* a*, din* aG *sau* b*, din* bG*. Din asta observăm că siguranța acestui protocol este garantată de dificultatea problemei logaritmului discret.*

Exemplu 3.1. *1 Fie curba eliptică NIST P192. Alice alege un număr random*

$$a = 4114691071888516598872686863459422089156924236587110051027$$

Aceasta calculează $Q_A = (3576689912069306634996719528847333570212949190268988897341,$ $2577620781095527148389100426144080789286031064305720917544)$ *și trimite la Bob*

rezultatul. Analog, Bob calculează $Q_B = bG$, unde este ales random,
$b = 3350281580565627922550490942568402436195033088753006169393$. *Alice primește:*

$$Q_B = (5237452004119114225824580697958296588006171898236471778302,$$

$$5239066786042179057430024496995536348905285581132510721784)$$

Ambele entități dețin acum secretul comun:

$$abG = (3889091514766761083889527264369850820381968816940879440305,$$

$$4004201504544591016017764551744695759122710025389314034700)$$

3.3 ECDSA

Aproape toate criptosistemele cu cheie publică, clasice, au un analog pe curbe eliptice. Astfel avem *ECDSA*, analogul pentru *DSA* (*Digital Signatura Algorithm*), propus in anul 1992 de către Scott Vanstone [13]. *ECDSA* a fost acceptat ca standard ISO în anul 1998, ANSI în anul 1999, ca standard IEEE și NIST în anul 2000.
Acest protocol asigură următoarele servicii criptografice:

- integritatea datelor: asigură faptul că datele nu pot fi modificate neautorizat
- autentificarea originii datelor: asigură faptul că sursa datelor este cea cunoscută
- non-repudierea: asigură faptul că o entitate nu poate nega acțiunile pe care le-a făcut în cadrul protocolului

Există trei etape in cadrul *ECDSA*: generarea cheilor, semnarea mesajului, verificarea mesajului. Urmează în continuare trei definiții, fiecare detaliind câte o etapă din acest protocol. Vom numi Alice si Bob enitățile care participa la comunicare. Alice dorește să trimită un mesaj semnat la Bob. Acesta va verifica semnatura. Fie E o curbă eliptică peste corpul F_p si un punct $G \in E$ de ordin n.

Definiție 3.2. Alice generează perechea cheie publică, cheie privată, astfel:

- Alege un număr natural $x \in [1, n-1]$
- Calculeaza $Q = xG$
- Q devine public, x rămâne secret. Perechea cheie publică, cheie privată este (Q, x)

Definiție 3.3. Alice generează o semnătură pentru mesajul m, astfel:

- Alege un număr natural $k \in [1, n-1]$
- Calculează $kP = (x1, y1)$ si $r \equiv x1 \bmod n$. Dacă $r = 0$, atunci ne întoarcem la primul pas
- Calculează $k^{-1} \bmod n$
- Calculează $s \equiv k^{-1}(H(m) + xr) \bmod n$. $H(m)$ reprezintă hash-ul mesajului m. Dacă $s = 0$ ne întoarcem la primul pas.

Perechea (r, s) reprezintă semnătura mesajului m

Definiție 3.4. Bob dorește sa verifice semnătura (r, s) pentru mesajul m. Următorii pași sunt necesari verificării:

- Verifică apartenența punctului Q, cheia publică, la curba E
- Verificarea faptului că r, s apartin intervalului $[1, n-1]$
- Calculează $H(m)$, unde H este aceași funcție hash folosită la semnare
- Calculează $w = s^{-1} \bmod n$
- Calculează $u_1 = H(m)w \bmod n$ si $u_2 = rw \bmod n$
- Calculează punctul $(x_1, y_1) = u_1 G + u_2 Q$
- Verifică dacă $r \equiv x_1$

Dacă oricare dintre verificări returnează fals, atunci semnătura este invalidă.

Exemplu 3.2. *Fie curba eliptică NIST P192. Alice dorește să semneze mesajul "ECDSA Test" și Bob dorește să verifice semnătura generată de Alice pentru acest mesaj. Vom folosi algoritmul de hash, SHA-256.*
Fie $k = 4625097095239057140588402855395245031027973496939430959487$ *ales random în intervalul* $[1, 6277101735386680763835789423176059013767194773182842284081]$.
Calculăm $k^{-1} = 4710413267373252408099540974110494037363888547813696222366$ *și obținem semnătura*

$$(2699032564945752962859925026972916556791993705928932713 10,$$

$$699408792794960665825042281503387585867271893408733500400)$$

Acum Bob dorește sa verice această semnătură folosind cheia publică:

$$(269903256494575296285992502697291655679199370592893271310,$$

$$2643207341070101961263344757054732948306561800541827620664)$$

Primul doi pași sunt îndepliniți apoi calculează:

$$w = 714843452363798735796354036598666200526458427179804068702$$

,

$$u_1 = 3633317631989915989067429594252245985031715088503959053275$$

$$u_2 = 2970936840042406432399184033378025194469637021946843353374$$

$$(x_1, x_2) = (269903256494575296285992502697291655679199370592893271310,$$

$$2643207341070101961263344757054732948306561800541827620664)$$

Verifică apoi $x_1 \equiv r$ mod n si semnătura este intr-adevăr autentică.

CAPITOLUL 4

IMPLEMENTARE, TESTE

În aceast capitol vom prezenta aplicația, începând de la arhitectură, continuând apoi spre funcționalitatea oferită. Idea este să punem in aplicare conceptele teoretice de la capitolele anterioare, punând accent pe algoritmii de la secțiunea Aritmetica Eficienta, capitolul 2. Implementarea eficientă si corectă a acestor operații reprezintă un prim pas foarte important spre dezvoltarea de aplicații criptografice folosite în lumea reală, precum ECDSA.
Pentru implementare am ales limbajul de programare Python, versiunea 3.5.2, iar hardware-ul folosit in tabelele de test este: Quad Core CPU, i7-4700HQ, 2.4 GHz, 64 bit OS, 16 GB RAM.

4.1 Arhitectura Aplicației

În această aplicație s-a urmărit scrierea unui cod cât mai flexibil și concis, care să permită manipularea și aprofundarea conceptelor abstracte, matematice, discutate în secțiunile predente. Scopul final al aplicației a fost implementarea protocolul *ECDSA* și un studiu comparativ al performanței algoritmilor discutati in Capitolul 2, Secțiunea Aritmetică Eficientă.
Deși limbajul folosit suportă atât paradigma de programare orientată pe obiecte, cât si cea procedurală, în această aplicație am ales structurarea aproape completă a codului în clase, fiecare cu roluri bine definite, pe cât posibil în concordanță cu principiile *SOLID*[14], fară însă a face compromisuri în ceea ce privește flexibilitatea si simplitatea

codului. Diagrama *UML*[15] din figura 4.1 surprinde toate clasele din aplicație si relațiile dintre acestea. Se observă folosirea unor clase abstracte, deși nu există suport nativ pentru ele în Python. Acestea au fost folosite pentru a reduce multe din reduntanțele care apăreau în cod și pentru a oferi un sablon pentru anumite funcționalități care pot fi introduse în aplicație. Abstractizarea claselor în Python se face cu ajutorul pachetului *abcMeta*[16].

Funcționalitatea oferită de protocolul ECDSA, adică generarea cheilor, semnarea mesajului si verificarea semnăturii este încapsulată în 3 clase: *GenerateKeyPair*, *GenerateSignature*, *VerifySignature*. Algoritmii care vor apărea în Studiul Comparativ sunt implementați în clasele *ScalarMultiplication* și *JointMultiplication*. În continuare, voi prezenta structurile de date care stau la baza aplicației.

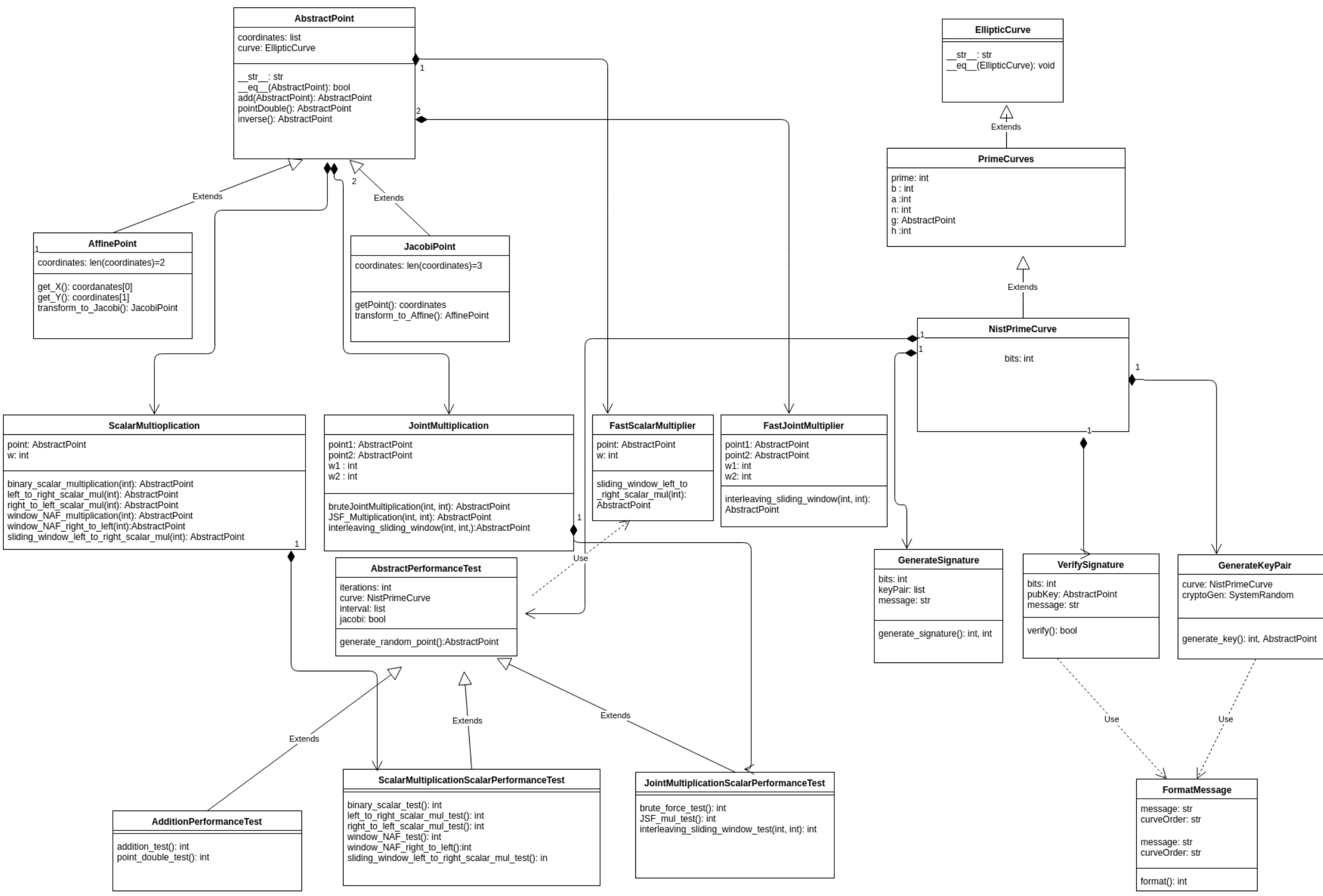

Figura 4.1: Arhitectura Aplicației

4.1.1 Structuri de date

În aplicație a fost necesară modelarea a două entități fundamentale, curba eliptică și punctul de pe o curbă eliptică. Pentru ambele concepte am creat clase abstracte, întrucât există multe tipuri de curbe eliptice, iar punctele pot avea și ele mai multe reprezentări.

4.1.1.1 Curba Eliptică

Am ales să modelăm conceptele de curbă eliptică peste un corp prim(clasa concretă *PrimeCurves*) și extensia acesteia care modelează curbele NIST peste corpuri prime, *NistPrimeCurve*. Clasa abstractă pentru o curbă eliptică este special modelată vag, din cauza multitudinii de tipuri de curbe existente, cu proprietăți specifice foarte speciale. Clasa *PrimeCurves* modelează orice curbă definită peste un corp prim si în afara parametrilor care definesc curba si a supraîncărcării operatorilor de egalitate si de reprezentare string a obiectului, am decis să nu adaug metode speciale. Curbele Nist, definite peste corpuri prime si modelate în clasa *NistPrimeCurve* au parametrii cunoscuți, definiți în [17].

4.1.1.2 Punctele de pe o curbă eliptică

Am modelat punctele reprezentate în coordonate afine (*AffinePoint*) și punctele în coordonate jacobiene (*JacobiPoint*). Design-ul punctelor a fost conceput în așa fel încât fiecare punct să conțină metode pentru operațiile de grup pentru curbe eliptice: adunare, dublare, invers. De asemenea, fiecare punct, are coordonate și aparține unei curbe, lucru reflectat în constructorul clasei *AbstractPoint*. Am luat decizia să nu adaug în clasele pentru puncte metode de înmulțire cu un scalar sau de înmulțire multiplă deoarece cei mai eficienți algoritmi necesită precalcularea unor valori, acest lucru adaugând un overhead clasei punct. Astfel, cel mai eficient algoritm de înmultire a fost separat într-o clasă, care se ocupă special de această operație La fel am procedat și cu operația de înmulțire multiplă.

Abstractizarea acestor concepte ajută la eventuala extindere a aplicației și la evitarea codului duplicat. Prin extinderea claselor abstracte, putem adauga suport pentru alte tipuri de curbe eliptice, precum cele definite peste corpuri binare, curbe Edwards, sau suport pentru diferite tipuri de coordonate și operațiile speciale aferente.

4.2 Aritmetică

În această secțiune vom aborda operațiile de grup pentru curbe eliptice, adunare, dublare, invers și operațiile speciale, înmulțirea cu un scalar și înmulțirea multiplă. Cele din urmă au fost explicate în capitolul doi, secțiunea Aritmetică Eficientă. Aici ne vom concentra pe implementarea acestora și rolul acestora în Studiul Comparativ, respectiv în protocolul ECDSA.

Justifici clasele ScalarMultiplier, zici ca prezinti atat aritmetica de grup cat si cea speciala...

4.2.1 Operații de Grup

În aplicatie am implementat operațiile care țin de grupul punctelor de pe o curbă eliptică în cele doua clase concrete pentru puncte, cele pentru coordonate afine si pentru coordonate jacobiene. Algoritmii de adunare constau în aplicarea unor formule, cele pentru coordonate afine fiind explicitate în Capitolul 2, Secțiunea Grupul Punctelor de pe o curba eliptică. Formulele de adunare și dublare a unui punct pentru coordonate Jacobiene sunt conform acestei surse [18].

Operațiile care țin de structura de grup au o importanță deosebită, întrucât sunt folosite în fiecare algoritm de înmulțire cu scalar și implicit de înmulțire multiplă. Din acest motiv, ne dorim o implementare cât mai eficientă, orice înbunătățire la acest nivel aducând sporuri de performanță peste tot în aplicație. În coordonate afine, apare algoritmul lui Euclid Extins în calculul unor inverși modulari, astfel justificând folosirea coordonatelor jacobiene, care prin introducerea unui set de redundanțe în reprezentarea unui punct, elimină nevoia de a calcula inverși modulari. În continuare vom prezenta implementarea adunării și dublării unui punct în coordonate jacobiene.

```
def add(self, other):
    if self is None:
        return other
    if other is None:
        return self

    u1 = (self.x * other.z ** 2) % self.curve.prime
    u2 = (other.x * self.z ** 2) % self.curve.prime
    s1 = (self.y * other.z ** 3) % self.curve.prime
```

```
        s2 = (other.y * self.z ** 3) % self.curve.prime
        if u1 == u2:
            if s1 != s2:
                return None
            else:
                return self.point_double()

        h = (u2 - u1) % self.curve.prime
        r = (s2 - s1) % self.curve.prime
        x3 = (r ** 2 - h ** 3 - 2 * u1 * h ** 2) % self.curve.prime
        y3 = (r * (u1 * h ** 2 - x3) - s1 * h ** 3) % self.curve.prime
        z3 = (h * self.z * other.z) % self.curve.prime
        return Jacobi_Point([x3, y3, z3, pow(z3, 2, self.curve.prime),
                pow(z3, 3, self.curve.prime)], self.curve)

    def point_double(self):
        if self is None:
            return None
        if self.y == 0:
            return None
        s = (4 * self.x * self.y ** 2) % self.curve.prime
        m = (3 * self.x ** 2 + self.curve.a * self.z ** 4) % self.curve.prime
        _x = (m ** 2 - 2 * s) % self.curve.prime
        _y = (m * (s - _x) - 8 * self.y ** 4) % self.curve.prime
        _z = (2 * self.y * self.z) % self.curve.prime
        return Jacobi_Point([_x, _y, _z, pow(_z, 2, self.curve.prime),
                pow(_z, 3, self.curve.prime)], self.curve)
```

Atât în coordonate afine, cât și în cele jacobiene, inversul unui punct este foarte simplu de calculat, prin negarea componentei a doua din reprezentarea punctului. Singurul dezavantaj al folosirii coordonatelor jacobiene îl constituie operația de trecere înapoi în coordonate afine, operație care necesită calcului a doi inverși modulari.

```
def transform_to_affine(self):[language=Python]
        return AffinePoint([self.coordinates[0] * inv(self.coordinates[2] ** 2,
self.curve.prime) % self.curve.prime, (self.coordinates[1] * inv(self.coordinate
3, self.curve.prime)) % self.curve.prime], self.curve)
```

4.2.2 Înmulțirea cu un scalar

Putem aborda problema înmulțirii cu un scalar în diferite feluri, de la cele mai ineficiente metode(apelararea functiei de adunare de cate ori este nevoie) până la metode sofisticate și eficiente, cum ar fi înmulțirea cu fereastră glisantă.

Cel mai eficient algoritm, rezultat în urma Studiului Comparativ, a fost încapsulat în clasa *FastScalarMultiplier*.

4.2.2.1 Metoda Binară

Prima metodă pe care am implementat-o este cea binară. Algoritmul constă în procesarea reprezentării binare a scalarului de la cel mai nesimnificativ bit la cel mai semnificativ.

```
def binary_scalar_multiplication(self, d):
    P = self.point
    if d == 0:
        return None
    if d == 1:
        return P
    result = None
    while d > 0:
        if d % 2:
            result = P.add(result)
        d //= 2
        P = P.add(P)
    return result
```

În variabila *result*, ținem rezultatul după fiecare iterație, iar în variabila P ținem o copie punctului care dorim să îl înmulțim. Astfel, după fiecare bit parcurs dublăm P și când întâlnim un bit de 1 adunăm la rezultat P. La final returnăm variabila *result*.

4.2.2.2 Reprezentări cu semn

Un avantaj major al grupului punctelor de pe o curbă eliptică îl reprezintă calculul foarte ușor din punct de vedere computațional al inversului. Astfel, putem să apelăm la reprezentările cu semn pentru scalar, acestea optimizând calculul de înmulțire cu un scalar. Cea mai eficientă astfel de reprezentare este NAF.

În continuare voi prezenta un algoritm pentru calculul reprezentării NAF.

```python
def naf(d):
    i = 0
    res = []
    while d >= 1:
        if d % 2 == 0:
            res.append(0)
        else:
            res.append(2 - d % 4)
            d -= res[i]
        d //= 2
        i += 1
    return res[::-1]
```

Cifrele care formează reprezentarea NAF sunt generate de resturile împărțirii repetate a scalarului la 2. Această împărțire este una cu semn, resturile putând fii $\{0,1,-1\}$. Când, în bucla *while*, scalarul d este impar și trebuie sa alegem între $r \in \{-1,1\}$, alegem în așa fel încât $\frac{d-r}{2}$ să fie par, astfel făcând ca următoarea cifră din NAF să fie 0.

Având la dispoziție metoda pentru aflarea reprezentării cu semn a unui numar natural, putem eficientiza metoda binară. Algoritmul de la stânga la dreapta pentru calculul lui dP, unde d este scalarul si P este un punct de pe o curba eliptica, apelează funcția pentru reprezentare cu semn. Parcurgem aceasta, dublând cantitatea din variabila *result* după fiecare bit parcurs, iar la bitul de 1 adunăm punctul P, la bitul -1 scadem P din rezultat. Urmează implementarea acestui algoritm.

```python
def left_to_right_scalar_mul(self, d):
    signed_d = naf(d)
    result = None
    for i in signed_d:
        if result is None:
            result = None
        else:
            result = result.point_double()
        if i == 1:
            result = self.point.add(result)
```

```
        if i == -1:
            result = self.point.inverse().add(result)
    return result
```

Algoritmul de la dreapta la stânga funcționează în același mod, singura diferență fiind faptul că NAF-ul este calculat în aceași buclă while în care calculăm rezultatul final. În acea buclă descompunem scalarul, ținem bit-ul curent într-o variabilă, u, iar apoi în funcție de valoarea acestuia modificăm rezultatul. Cei doi algoritmi au aceași complexitate asimtotică.

```
def right_to_left_scalar_mul(self, d):
    result = None
    R = self.point
    while d >= 1:
        if d % 2 == 1:
            u = 2 - (d % 4)
            d -= u
            if u == 1:
                result = R.add(result)
            else:
                result = R.inverse().add(result)
        d //= 2
        R = R.point_double()
    return result
```

4.2.2.3 Metoda cu fereastră

Metoda cu fereastră poate fi privită ca o generalizare a metodei cu semn, aceasta aducând plusuri de performanță în schimbul folosirii unui plus de memorie. Această metodă se folosește de reprezentarea $w-NAF$ a scalarului. Un algoritm eficient pentru calculul $w-NAF$ va fi prezentat în continuare. Funcția mods este modulo cu semn, adica pentru un scalar d, avem d mods $2^w = u$, unde u este un numar intreg care satisface $u \equiv d$ mod 2^w si $-2^{w-1} \leq u \leq 2^{w-1}$.

```
def w_NAF(d, w):
    i = 0
    res = []
    while d >= 1:
```

```
        if d % 2 == 0:
            res.append(0)
        else:
            res.append(mods(d, w))
            d -= res[i]
        d //= 2
        i += 1
    return res[::-1]
```

Se observă că cifrele din $w-NAF$ sunt de fapt restul împărțirii cu semn a scalarului la 2, rest care aparține intervalului $[-2^{w-1}, 2^{w-1}-1]$. Astfel, daca scalarul d este impar și $r = d$ mods 2^w, atunci $\frac{d-r}{2}$ este divizibil cu 2^{w-1}, asigurând astfel ca următoarele $w-1$ cifre din reprezentare sunt 0.

Metoda cu fereastră aduce un plus de performanță prin reducerea numărului de adunări necesare în calculul multiplului unui punct oarecare P de pe o curbă eliptică. Aici vom folosi reprezentarea $w-NAF$ in loc de NAF. Vom precalcula și stoca valorile pentru iP si $i \in \{1, 2^{w-1}\}$. Astfel, când parcurgem $w-NAF$ -ul numarului, în funcție de cifra gasită vom aduna sau scădea din rezultat valorile potrivite din multimea de valori precalculate, dublând variabila acumulator la fiecare iterație.

```
def window_NAF_multiplication(self, d, w):
    d = w_NAF(d, w)
    Q = None
    for i in range(0, len(d)):
        if Q is None:
            Q = None
        else:
            Q = Q.point_double()
        if d[i] != 0:
            if d[i] > 0:
                Q = P[d[i]].add(Q)
            else:
                Q = P[-d[i]].inverse().add(Q)
    return Q
```

Valorile precalculate sunt stocate într-un dicționar P. Dicționarul în Python este o structură de date de tip *Hash Map*. Am ales să stocăm astfel, din cauză că această structură de date oferă lookup-uri foarte rapide, în $\mathcal{O}(1)$ amortizat.

O versiune ușor modificată a algoritmului precedent este propusă în [19]. Acesta este un algoritm de la dreapta la stânga, care calculează $w - NAF$ pentru scalar în aceași buclă în care este calculat și rezultatul. Valorile diferite de zero din reprezentare sunt stocate intr-un Hash Map si rezultatul este calculat la final, prin adunarea valorilor stocate și returnat.

```
def window_NAF_on_the_fly_scalar_mul(self, k, w):
    R = self.point
    m = 2**(w-1) - 1
    Q = {}
    for i in range(1, m + 1, 2):
        Q[i] = None
    while k >= 1:
        if k % 2 == 1:
            t = mods(k, w)
            if t > 0:
                Q[t] = R.add(Q[t])
            if t < 0:
                Q[-t] = R.inverse().add(Q[-t])
            k -= t
        R = R.point_double()
        k //= 2
    for i in range(3, m + 1, 2):
        if Q[i] is not None:
            Q[1] = Q[i].right_to_left_scalar_mul(i).add(Q[1])
    return Q[1]
```

.

4.2.2.4 Metoda cu fereastră glisantă

Diferența dintre metoda prezentată anterior și aceasta constă în faptul că aceasta din urmă ne permite să sărim peste zerourile din reprezentarea $w - NAF$ a scalarului. Pentru această metodă vom implementa o versiune ușor modificată față de cea originală, propusă de către [20].

Algoritmul apelează funcția pentru $w - NAF$ si parcurge în bucla while această reprezentare. Când găsim prima cifră diferită de zero, o procesăm, adăugam la rezultat valoarea corespunzătoare din dicționarul de valori precalculate, apoi "glisăm" fereastra peste cifrele care sunt zero.

```
def sliding_window_left_to_right_scalar_mul(self, d, w):
    d = w_NAF(d, w)
    Q = None
    i = 0
    while i < len(d):
        if d[i] == 0:
            if Q is None:
                Q = None
            else:
                Q = Q.point_double()
            i += 1
        else:
            s = max(len(d) - i - w + 1, 0)
            s = len(d) - 1 - s
            while d[s] == 0:
                s -= 1
            u = NAF(d[i:s + 1])
            for j in range(1, i - s + 2):
                if Q is not None:
                    Q = Q.point_double()
                else:
                    Q = None
            if u > 0:
                Q = _P[u].add(Q)
            if u < 0:
                Q = Q.add(_P[-u].inverse())
            i = 1 + s
    return Q
```

4.2.3 Înmulțirea multiplă

A doua operație specială implementată în aplicație este cea a înmulțirii multiple, care constă în înmulțirea unor puncte cu scalari și apoi adunarea lor. O abordare naivă ar fi să folosim metodele de la secțiunea precedentă pentru înmultirea fiecărui punct, separat apoi adunarea rezultatelor. Aceasta este explicitată în rândurile care urmează.

4.2.3.1 Metoda Naivă

Această metodă, folosește cel mai rapid algoritm de înmulțire, cel cu fereastră glisantă, pentru a înmulții punctele cu doi scalari, k, l. Apoi adunăm rezultatele și găsim soluția.

```
def brute_joint_multiplication(self, k, l):
    result1 = self.multiplier1.sliding_window_left_to_right_scalar_mul(k)
    result2 = self.multiplier2.sliding_window_left_to_right_scalar_mul(l)
    return result1.add(result2)
```

4.2.3.2 Metoda cu Joint Sparse Form

Un dezavantaj major al metodei naive este calculul separat al înmulțirii cu un scalar. Metoda Joint Sparse Form, folosește reprezentarea combinată a scalarilor, definită în Capitolul 2, Secțiunea Aritmetică Eficientă. În continuare, vom prezenta implementarea metodei care folosește această reprezentare, așa numitul "Shamir's Trick".

```
def JSF_Multiplication(self, k, l):
    """Add using Shamir Trick, variation of algorithm 3.48, Menezez Book"""
    jsf = self.JSF(k, l)

    R = None

    for i, j in zip(jsf[0], jsf[1]):
        if R is None:
            R = None
        else:
            R = R.point_double()
        if i or j:
            R = precom[i][j].add(R)
        return R
```

De data aceasta, ținem într-o structura de tip static valorile precalculate, de tip matrice, $3x3$, pe care am construit-o, în așa fel încât, să acoperim toate combinațiile posibile de cifre, care pot apărea într-o coloană din reprezentare.

4.2.3.3 Metoda cu fereastră intercalată, w-NAF

Pentru această metoda, vom considera reprezentările $w-NAF$ ale scalarilor. Vom pada cu 0, reprezentarea mai scurtă, pentru a avea ambele $w-NAF$ de aceași lungime. Vom ține valorile precalculate în Hash Maps, pe care le accesăm când facem calculul propriuzis.

```
def interleaving_sliding_window(self, k, l, w1, w2):
    """Algorithm 3.5.1 menezez, 'Interleaving with NAF' """

    _P = {}
    _Q = {}
    R = None
    naf = [w_NAF(k, w1), w_NAF(l, w2)]
    _l = max([len(naf[0]), len(naf[1])])

    #padding stage
    for i in range(_l - len(naf[0])):
        naf[0].insert(i, 0)
    for i in range(_l - len(naf[1])):
        naf[1].insert(i, 0)

    for i in range(_l):
        if R is None:
            R = None
        else:
            R = R.point_double()
        for j in range(2):
            if naf[j][i] != 0:
                if naf[j][i] > 0:
                    if j == 0:
                        R = _P[naf[j][i]].add(R)
                    else:
```

```
                    R = _Q[naf[j][i]].add(R)
            else:
                if j == 0:
                    R = _P[-naf[j][i]].inverse().add(R)
                else:
                    R = _Q[-naf[j][i]].inverse().add(R)
    return R
```

Am creat o matrice de $2 \times l$, unde l este lungimea $w - NAF$. Astfel, în matrice avem reprezentările celor doi scalari. Parcurgem cu două bucle *for* matricea, și în funcție de valorile coloanelor adunăm sau scădem din variabila acumulator R. După fiecare coloană parcursă, dublăm valoarea din acumulator.

4.3 ECDSA

ECDSA este un protocol de semnare digitală, folosit cu succes pentru securizarea monedei virtuale *Bitcoin*, sau în protocololul *SSL*/*TLS*. Se asigură integritatea datelor, autentificarea originii și non-repudierea. Protocolul se desfășoară în trei etape, Generarea cheilor, semnarea mesajului și verificarea semnăturii. Funcționalitatea necesară desfășurării celor trei etape este încapsulată în 3 clase, *GenerateKeyPair*, *GenerateSignature* și *VerifySignature*. Curbele eliptice folosite în protocol sunt curbele Nist, astfel parametrii sunt cunoscuți.

4.3.1 Generarea cheilor

Perechea cheie privată, cheie publică în acest protocol se generează astfel. Cheia privată reprezintă un număr, k, ales random intre $[1, n-1]$, unde n este ordinul punctului generator, G. Cheia publică este kG. Operația de înmulțire cu un scalar se realizează cu ajutorul algoritmului cu fereastră glisantă, implementat în clasa *FastScalarMultiplier*, prezentat în secțiunea precedentă. Metoda care generează cheile, din clasa *GenerateKeyPair*, este implementată astfel:

```
def generate_key(self):
    multiplier = FastScalarMultiplier(self.curve.g)
    k = self.cryptogen.randrange(1, self.curve.n - 1)
    point = multiplier.sliding_window_left_to_right_scalar_mul(k)
    if point.get_X() == 0:
```

```
        raise ValueError("Please␣Generate␣Key␣pair␣Again")
    return k, point
```

Astfel, după crearea unei instanțe din această clasa și apelul funcției de mai sus, avem cheile necesare desfășurării protocolului.

4.3.2 Generarea semnăturii

Semnătura este derivată din perechea cheie privată, cheie publică și este formată din numerele întregi, r și s. Metoda din clasa *GenerateKeyPair* care generează semnătura, este implementată astfel:

```
def generate_signature(self):
    r = self.pubKey.get_X()
    s = (inv(self.prvKey, self.curve.n) *
    (self.z + r * self.prvKey)) % self.curve.n
    if s == 0:
        raise ValueError("Generate␣key␣pair␣again")
    return r, s
```

Pentru calculului inversului modulo n a cheii private se folosește algoritmul lui Euclid Extins. Clasa *GenerateKeyPair* primește în constructor, atât perechea de chei generată, cât și mesajul care se dorește a fi semnat, în format *string*.

4.3.3 Verificarea semnăturii

Avem nevoie de trei lucruri pentru a verifica semnătura, acestea fiind semnătura, cheia publică și mesajul. Astfel, aceste trei lucruri se găsesc în constructorul clasei *VerifySignature*. Algoritmul de verificare a semnăturii este implementat astfel:

```
def verify(self):
    # pasul 1 --> Check to see signature numbers are in range
    if 1 > self.sig[0] or self.sig[0] >= self.curve.n
    or 1 > self.sig[0] or self.sig[0] >= self.curve.n:
        return False

    # pasul 4 --> calculam w = s^-1
    w = inv(self.sig[1], self.curve.n)
    #print("w is " + str(w))
```

```
# pasul 5 --> calculam u1 = zw mod n si u2 = rw mod n
u1 = (self.z * w) % self.curve.n
u2 = (self.sig[0] * w) % self.curve.n
#print("u_1 is " + str(u1))
#print("u_2 is " + str(u2))

pasul 6 --> calculam punctul de pe curba eliptica
muliplier = FastJointMultiplier(self.curve.g, self.pubKey)
p = muliplier.interleaving_sliding_window(u1, u2)
#print("(x1, y1) is " + str(p))

# pasul 7 --> check signature
if self.sig[0] == p.get_X():
    return True
return False
```

Pașii din acest algoritm sunt cei descriși în Capitolul 3, Secțiunea ECDSA. Se observă folosirea algoritmului eficient de înmulțire multiplă la pasul 6, încapsulat în clasa *FastJointMultiplier*. Astfel, în acest protocol, am aplicat algoritmii eficienți, explicitați în secțiunea precedentă.

4.4 Studiu Comparativ

Nu uita sa faci un tabel cu ECDSA, eficient/ineficient... Poate numele sa ramana doar studiu comparativ.

În acestă secțiune vom face o comparație între algoritmii prezentați la secțiunea Aritmetică. Vom testa impactul folosirii coordonatelor jacobiene și vom vedea de asemenea cum diferite implementări afectează performanța unui protocol foarte folosit, precum este ECDSA. Hardware-ul folosit în tabelele de test este: Quad Core CPU, i7-4700HQ, 2.4 GHz, 64 bit OS, 16 GB RAM.

Primele operații testate sunt adunarea și dublarea. Vom face o comparație între cele două metode tipuri de reprezentări, cea cu coordonate afine si cea în coordonate jacobiene. Pentru testare am generat random două numere pe o curba eliptică si le-am adunat. Am cronometrat 1000 de adunări, de fiecare dată cu puncte diferite. Nu am cronometrat generarea random a punctelor.

Adunarea punctelor de pe curbe eliptice			
Curba NIST	Coordonate	Metoda	Timp de execu-tie(secunde)
P192	Afine	Adunare	0.052
P192	Afine	Dublare	0.0513
P192	Jacobiene	Adunare	0.0114
P192	Jacobiene	Dublare	0.0083
P224	Afine	Adunare	0.061
P224	Jacobiene	Adunare	0.0123
P256	Jacobiene	Adunare	0.01474
P256	Jacobiene	Dublare	0.0103
P384	Afine	Adunare	0.11
P384	Jacobiene	Adunare	0.0212
P384	Jacobiene	Dublare	0.0139

Tabela 4.1

Observăm ca implementarea în coordonate jacobiene aduce un spor serios de performanță, operațiile de adunare și dublare fiind de aproximativ cinci ori mai rapide cu această reprezentare.

Pentru testarea operației de înmulțire cu un scalar am decis să aleg diferite intervale pentru mărimea scalarului, aceastea depinzând de curba selectată. Pentru fiecare test sunt rulate 1000 de iterații cu scalari aleși random, în intervalul $5-32$ biti, respectiv $128-198$ si $330-384$ pentru curbele $P192$ și respectiv $P384$. Ne vom concentra în special pe coordonate jacobiene în testele care urmează.

Curba P192				
Algoritm	Coordonate	Intervalul	Fereastra	Timp
Alg Binar	Afine	$[2^5, 2^{32}]$	-	2.38
Alg Binar	Jacobiene	$[2^5, 2^{32}]$	-	0.58
Alg Binar	Jacobiene	$[2^{128}, 2^{192}]$	-	3.44
Inmultire de st. la dr.	Jacobian	$[2^5, 2^{32}]$	-	0.4
Inmultire de st. la dr.	Afine	$[2^{128}, 2^{192}]$	-	13.8
Inmultire de st. la dr.	Jacobian	$[2^{128}, 2^{192}]$	-	2.55
Inmultire de dr. la st.	Afine	$[2^{128}, 2^{192}]$	-	13.69
Inmultire de dr. la st.	Jacobiene	$[2^5, 2^{32}]$	-	0.406
Inmultire de dr. la st.	Jacobiene	$[2^{128}, 2^{192}]$	-	2.43
Metoda cu fereastra	Jacobiene	$[2^5, 2^{32}]$	3	0.365
Metoda cu fereastra	Jacobiene	$[2^{128}, 2^{192}]$	3	2.31
Metoda cu fereastra	Jacobiene	$[2^{128}, 2^{192}]$	4	2.14
Metoda cu fereastra	Jacobiene	$[2^{128}, 2^{192}]$	5	2.07
Metoda cu fereastra dr la st	Jacobiene	$[2^5, 2^{32}]$	3	0.43
Metoda cu fereastra dr la st	Jacobiene	$[2^{128}, 2^{192}]$	3	2.35
Metoda cu fereastra dr la st	Jacobiene	$[2^{128}, 2^{192}]$	3	2.29
Metoda cu fereastra dr la st	Jacobiene	$[2^{128}, 2^{192}]$	5	2.47
Fereastra glisanta st. la dr.	Jacobiene	$[2^5, 2^{32}]$	3	0.37
Fereastra glisanta st. la dr.	Jacobiene	$[2^{128}, 2^{192}]$	4	2.11
Fereastra glisanta st. la dr.	Jacobiene	$[2^{128}, 2^{192}]$	5	1.98
Fereastra glisanta st. la dr.	Afine	$[2^{128}, 2^{192}]$	5	10.93

Tabela 4.2

Se observă eficiența metodelor cu fereastră pentru numere mari, algoritmul de înmulțire care folosește reprezentarea cu semn fiind foarte eficient pentru numere mici. Algoritmii cu fereastra sunt însă cu aproximativ 15% mai eficienți decât metodele care folosesc reprezentarea cu semn a scalarului. De asemenea nu se observă o înbunătățire a eficienței algoritmilor dacă procesam de la stânga la dreapta sau de la dreapta la stânga în nici o metodă testată.

Pentru scalarii mari, cele mai bune rezultate au fost obținute cu o fereastră glisantă de lățime 5. Algoritmii devin din ce în ce mai buni, odată cu mărirea ferestrelor, dar costul din punct de vedere al memoriei devine unul ridicat.

De asemenea putem observa beneficiul impresionant adus de folosirea coordonatelor Jacobiene, acestea aducând îmbunătățiri consistente de performanta, algoritmii fiind de aproximativ $5-6$ ori mai rapizi in aceste coordonate.

Curba P384				
Algoritm	Coordonate	Intervalul	Fereastra	Timp
Alg Binar	Afine	$[2^5, 2^{32}]$	-	5.25
Alg Binar	Jacobiene	$[2^5, 2^{32}]$	-	0.99
Alg Binar	Jacobiene	$[2^{128}, 2^{192}]$	-	12.36
Inmultire de st. la dr.	Jacobian	$[2^5, 2^{32}]$	-	0.68
Inmultire de st. la dr.	Afine	$[2^{330}, 2^{384}]$	-	58.25
Inmultire de st. la dr.	Jacobian	$[2^{330}, 2^{384}]$	-	8.25
Inmultire de dr. la st.	Afine	$[2^{330}, 2^{384}]$	-	58
Inmultire de dr. la st.	Jacobiene	$[2^5, 2^{32}]$	-	0.65
Inmultire de dr. la st.	Jacobiene	$[2^{330}, 2^{384}]$	-	8.02
Metoda cu fereastra	Jacobiene	$[2^5, 2^{32}]$	3	0.58
Metoda cu fereastra	Jacobiene	$[2^{330}, 2^{384}]$	3	7.45
Metoda cu fereastra	Jacobiene	$[2^{330}, 2^{384}]$	4	7.05
Metoda cu fereastra	Jacobiene	$[2^{330}, 2^{384}]$	5	6.80
Metoda cu fereastra dr la st	Jacobiene	$[2^5, 2^{32}]$	3	0.7
Metoda cu fereastra dr la st	Jacobiene	$[2^{330}, 2^{384}]$	3	7.7
Metoda cu fereastra dr la st	Jacobiene	$[2^{330}, 2^{384}]$	4	7.33
Metoda cu fereastra dr la st	Jacobiene	$[2^{330}, 2^{384}]$	5	7.55
Fereastra glisanta st. la dr.	Jacobiene	$[2^5, 2^{32}]$	3	0.6
Fereastra glisanta st. la dr.	Jacobiene	$[2^{330}, 2^{384}]$	4	7.08
Fereastra glisanta st. la dr.	Jacobiene	$[2^{330}, 2^{384}]$	5	6.55
Fereastra glisanta st. la dr.	Afine	$[2^{330}, 2^{384}]$	5	47.2

Tabela 4.3

Pentru testarea operației de înmulțire multiplă am considerat de asemenea 1000 de iterații, împreună cu două ordine de mărime pentru scalari. Se observă ca metoda cu Joint Sparse Form aduce o înbunătățire cu aproximativ 50% fată de abordarea naivă. Metoda cu fereastră intercalată aduce înbunătățiri de aproximativ 20% cu ferestrele 6,6 față de metoda JSF. Consistent cu observațiile de la Adunare și Înmulțirea cu scalar, coordonatele jacobiene aduc înbunătățiri de aproximativ 500%. Așadar este de aproximativ $10-12$ ori mai eficientă operația de înmulțire multiplă în coordonate jacobiene cu metoda ferestrei intercalate, decât aceași operație în coordonate afine, cu metoda naivă.

Curba P192				
Algoritm	Coordonate	Intervalul	Ferestre	Timp
Alg Brut	Afine	$[2^5, 2^{32}]$	-	3.7
Alg Brut	Afine	$[2^{128}, 2^{198}]$	-	24
Alg Brut	Jacobiene	$[2^5, 2^{32}]$	-	0.73
Alg Brut	Jacobiene	$[2^{128}, 2^{198}]$	-	4.47
Inmultire cu JSF	Afine	$[2^5, 2^{32}]$	-	2.45
Inmultire cu JSF	Afine	$[2^{128}, 2^{198}]$	-	15.54
Inmultire cu JSF	Jacobiene	$[2^5, 2^{32}]$	-	0.49
Inmultire cu JSF	Jacobiene	$[2^{128}, 2^{198}]$	-	3.08
Interleaving	Jacobiene	$[2^5, 2^{32}]$	3, 3	0.52
Interleaving	Jacobiene	$[2^{128}, 2^{198}]$	3, 3	3.17
Interleaving	Jacobiene	$[2^{128}, 2^{198}]$	3, 4	2.95
Interleaving	Jacobiene	$[2^{128}, 2^{198}]$	4, 4	2.83
Interleaving	Jacobiene	$[2^{128}, 2^{198}]$	4, 5	2.77
Interleaving	Jacobiene	$[2^{128}, 2^{198}]$	5, 5	2.7
Interleaving	Afine	$[2^{128}, 2^{198}]$	6, 6	12.8
Interleaving	Jacobiene	$[2^{128}, 2^{198}]$	6, 6	2.53

Tabela 4.4

Rulăm aceleași teste și pentru curba $P384$, obținând aproximativ aceleași diferențe între algoritmi. Se observă că, algoritmii în coordonate afine, sunt de aproximativ 4.3 ori mai ineficienți pentru curba aceasta, față de curba $P192$. În coordonate jacobiene lucrurile stau mai bine, diferențele de timp fiind mai mici, de aproximativ 3,2 ori mai lenți pentru $P384$, față de $P192$. Așadar, pentru curba $P384$, folosirea coordonatelor afine aduce un dezavantaj foarte pronunțat de performanță.

Curba P384				
Algoritm	Coordonate	Intervalul	Ferestre	Timp
Alg Brut	Afine	$[2^5, 2^{32}]$	-	8.18
Alg Brut	Afine	$[2^{330}, 2^{384}]$	-	104.27
Alg Brut	Jacobiene	$[2^5, 2^{32}]$	-	1.12
Alg Brut	Jacobiene	$[2^{330}, 2^{384}]$	-	14.39
Inmultire cu JSF	Afine	$[2^5, 2^{32}]$	-	5.5
Inmultire cu JSF	Afine	$[2^{330}, 2^{384}]$	-	65.4
Inmultire cu JSF	Jacobiene	$[2^5, 2^{32}]$	-	0.79
Inmultire cu JSF	Jacobiene	$[2^{330}, 2^{384}]$	-	9.78
Interleaving	Jacobiene	$[2^5, 2^{32}]$	3, 3	0.79
Interleaving	Jacobiene	$[2^{330}, 2^{384}]$	3, 3	9.86
Interleaving	Jacobiene	$[2^{330}, 2^{384}]$	3, 4	9.47
Interleaving	Jacobiene	$[2^{330}, 2^{384}]$	4, 4	8.99
Interleaving	Jacobiene	$[2^{330}, 2^{384}]$	4, 5	8.69
Interleaving	Jacobiene	$[2^{330}, 2^{384}]$	5, 5	8.52
Interleaving	Afine	$[2^{330}, 2^{384}]$	6, 6	56.15
Interleaving	Jacobiene	$[2^{330}, 2^{384}]$	6, 6	8.19

Tabela 4.5

4.5 Testare

Pentru a ne asigura de funcționarea corectă a aplicației, am decis să implementăm o suită de *unit teste*, cu ajutorul framework-ului [21]. Au fost testate toate operațiile de grup, aritmetica specială și protocolul ECDSA.

4.5.1 Testarea Operațiilor de grup

Pentru a testa aceste operații, am creat doua clase de test,*AffinePointTest*, *JacobiPointTest*, pentru cele două tipuri de puncte. Am considerat o curbă eliptică de dimensiuni foarte mici, $E : y^2 = x^3 + 2x + 3 \bmod 97$. Implementarea clasei de test arată în felul următor:

```
class AffinePointTest(TestCase):

    def test_add(self):
        curve, point1, point2 = self.createSUT()
        result = point1.add(point2)
```

```
        expected_result = AffinePoint([1, 54], curve)
        self.assertEqual(result, expected_result)

    def test_point_double(self):
        curve, point1, point2 = self.createSUT()
        result = point1.point_double()
        expected_result = AffinePoint([32, 90], curve)
        self.assertEqual(result, expected_result)

    def test_inverse(self):
        curve, point1, point2 = self.createSUT()
        result = point1.inverse()
        expected_result = AffinePoint([17, 87], curve)
        self.assertEqual(result, expected_result)

    def test_transform_to_Jacobi(self):
        curve, point1, point2 = self.createSUT()
        result = point1.transform_to_Jacobi()
        expected_result = JacobiPoint([17, 10, 1], curve)
        self.assertEqual(result, expected_result)

    @staticmethod
    def createSUT():
        curve = PrimeCurves(97, 3, 2)
        point1 = AffinePoint([17, 10], curve)
        point2 = AffinePoint([95, 31], curve)
        return curve, point1, point2
```

Avem o metodă statică care creează datele necesare testelor. Fiecare test este independent. O clasă construită analog testează funcționalitatea operațiilor în coordonate jacobiene.

4.5.2 Testarea aritmeticii speciale

Pentru testarea înmulțirii cu un scalar, am păstrat curba $E : y^2 = x^3 + 2x + 3 \bmod 97$. Am ales punctul $point1 = (73, 14) \in E$ și scalarul 6. Am creat o instanță a clasei ScalarMultiplication și am testat pe rând fiecare algoritm. Un exemplu de unit test arată în felul următor:

```
def test_sliding_window_left_to_right_scalar_mul(self):
    curve, instance, scalar, expected_result = self.createSUT()
    result = instance.binary_scalar_multiplication(scalar)
    self.assertEqual(result, expected_result)
```

Metoda statică care crează datele pentru teste, este:

```
@staticmethod
def createSUT():
    curve = PrimeCurves(97, 3, 2)
    point1 = AffinePoint([73, 14], curve)
    instance = ScalarMultiplication(point1, 5)
    scalar = 6
    expected_result = AffinePoint([3, 91], curve)
    return curve, instance, scalar, expected_result
```

Testarea înmulțirii multiple are loc în condiții similare. De data aceasta, alegem punctele (73,14),(55,6), împreună cu scalarii 7,8. Metoda care crează instanța de test este:

```
@staticmethod
def createSUT():
    curve = PrimeCurves(97, 3, 2)
    point1 = AffinePoint([73, 14], curve)
    point2 = AffinePoint([55, 6], curve)
    instance = JointMultiplication(point1, point2, 5, 4)
    scalar1, scalar2 = 7, 8
    expected_result = AffinePoint([28, 34], curve)
    return instance, expected_result, scalar1, scalar2
```

Testăm fiecare algoritm cu aceste date, un exemplu ar fi:

```
def test_nterleaving_sliding_window(self):
    instance, expected_result, scalar1, scalar2 = self.createSUT()
    result = instance.interleaving_sliding_window(scalar1, scalar2)
    self.assertEqual(result, expected_result)
```

4.5.3 Testarea ECDSA

Pentru testarea protocolului ECDSA, am ales să ne concentrem pe verificarea semnăturii. Mai întâi am testat condițiile normale de funcționare, adică semnătura este verificată pentru mesajul original și pentru semnătura corecta, apoi am modificat acești doi parametrii, verificând faptul ca verificarea returnează fals. Astfel, am creat clasa de test pentru ECDSA.

```
class EcdsaTest(TestCase):
    def test_sig_with_correct_parameters(self):
        bits, sig, message, pub = self.createSut()
        verifier = VerifySignature(bits, pub, sig, message)
        result = verifier.verify()
        self.assertEqual(result, True)

    def test_wrong_message_should_not_verify(self):
        bits, sig, message, pub = self.createSut()
        altered_mes = "Wrong␣message"
        verifier = VerifySignature(bits, pub, sig, altered_mes)
        result = verifier.verify()
        self.assertEqual(result, False)

    def test_wrong_sig_shoud_not_verify(self):
        bits, sig, message, pub = self.createSut()
        altered_sig = [131223132131, 132131231231]
        verifier = VerifySignature(bits, pub, altered_sig, message)
        result = verifier.verify()
        self.assertEqual(result, False)

    @staticmethod
    def createSut():
        bits = 192
        priv, pub = GenerateKeyPair(bits).generate_key()
        message = "ECDSA␣Test"
        sig = GenerateSignature(bits, [priv, pub], "ECDSA␣Test").generate_si
        return bits, sig, message, pub
```

Concluzii

În

BIBLIOGRAFIE

[1] Henry Cohen and Gerhard Frey.
Handbook of Elliptic and Hyperelliptic Curve Cryptography.
2006.
Chap. 2, Algebraic Background, pp. 19–35.

[2] Ion D. Ion and Nicolae Radu.
Algebra.
Editura Didactică și Pedagogică, 1975,
Pp. 31–123.

[3] Darrel Hankerson, Alfred Menezes, and Scott Vanstone.
Guide To Elliptic Curve Cryptography.
2014.
Chap. 3, Elliptic Curve Arithmetic, pp. 97–100.

[4] Joseph H. Silverman.
The Arithmetic of Elliptic Curves.
Springer, 2008.
Chap. 11, Algorithmic Aspects of Elliptic Curves.

[5] Jerome A. Solinas.
"Low-Weight Binary Representations for Pairs of Integers".
In: (2001).

[6] Darrel Hankerson, Alfred Menezes, and Scott Vanstone.
Guide To Elliptic Curve Cryptography.
2014.
Chap. 3, Elliptic Curve Arithmetic, p. 98.

[7] Darrel Hankerson, Alfred Menezes, and Scott Vanstone.
Guide To Elliptic Curve Cryptography.
2014.

Chap. 3, Elliptic Curve Arithmetic, p. 100.

[8] Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide To Elliptic Curve Cryptography*. 2014. Chap. 3, Elliptic Curve Arithmetic, p. 101.

[9] Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide To Elliptic Curve Cryptography*. 2014. Chap. 3, Elliptic Curve Arithmetic, pp. 111–112.

[10] Song Y. Yan. *Quantum Computational Number Theory*. Springer, 2015. Chap. 5, Quantum Computing for Elliptic Curve Discrete Logarithms, pp. 173–174.

[11] Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide To Elliptic Curve Cryptography*. 2014. Chap. 4, Cryptographic Protocols, pp. 153–155.

[12] Song Y. Yan. *Quantum Computational Number Theory*. Springer, 2015. Chap. 5, Quantum Computing for Elliptic Curve Discrete Logarithms, pp. 193–194.

[13] Scott Vanstone. "Responses to NIST's Proposal". In: (1992).

[14] *Principles Of Ood*. URL: `http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod`.

[15] *The Unified Modeling Language*. URL: `http://www.uml-diagrams.org/`.

[16] *Python Abstract Classes*. URL: `https://docs.python.org/3/library/abc.html`.

[17] *Recommended Elliptic Curves For Federal Government Use*. 1999.

[18] *Cryptography / Prime Curve / Jacobian Coordinates*.
URL: https://en.wikibooks.org/wiki/Cryptography/Prime_Curve/Jacobian_Coordinates.

[19] Marco Castrillón López, Luis Hernández Encinas, and Pedro Martínez Gadea.
Geometry, Algebra and Applications: From Mechanics to Cryptography.
2015.
Chap. 11, Implementation of Cryptographic Algorithms for Elliptic Curves, pp. 127–129.

[20] Darrel Hankerson, Alfred Menezes, and Scott Vanstone.
Guide To Elliptic Curve Cryptography.
2014.
Chap. 3, Elliptic Curve Arithmetic, pp. 100–101.

[21] *Python Testing Framework*.
URL: https://docs.python.org/3/library/unittest.html.