INSTITUTO SUPERIOR DE TECNOLOGIA EM CIÊNCIAS DA COMPUTAÇÃO DE PETRÓPOLIS – ISTCC-P
FUNDAÇÃO DE APOIO À ESCOLA TÉCNICA DO ESTADO DO RIO DE JANEIRO – FAETEC

Implementação de uma API para Criptografia Assimétrica Baseada em Curvas Elípticas

Pedro Carlos da Silva Lara

Petrópolis, RJ
2008

Pedro Carlos da Silva Lara

Implementação de uma API para Criptografia Assimétrica Baseada em Curvas Elípticas

Trabalho de conclusão de curso de graduação apresentado ao Instituto Superior de Tecnologia em Ciências da Computação de Petrópolis como parte dos requisitos para obtenção do título de "Tecnólogo da Informação e Comunicação".

Fábio Borges de Oliveira, M.Sc.
LNCC/MCT

Renato Portugal, D.Sc.
LNCC/MCT

Eduardo Lucio Mendes Garcia, D.Sc.
LNCC/MCT

Franklin de Lima Marquezino, M.Sc.
LNCC/MCT

Petrópolis, RJ
2008

Símbolos

$\mathbb{Z}_p$	Corpo finito formado pelos restos módulo p primo
$\mathbb{Z}_p^*$	Grupo multiplicativo formado pelos restos módulo p primo $-\{0\}$
$\mathbb{F}_q$	Corpo finito de ordem q
$\mathbb{N}$	Conjunto dos números naturais
$\mathbb{Z}$	Conjunto dos números inteiros
$\mathbb{Q}$	Conjunto dos números racionais
$\mathbb{R}$	Conjunto dos números reais
$b\|a$	Significa que b é divisor de a (a é múltiplo de b)
$\bar{a}$	Classe de equivalência do elemento a
$(\mathbb{Z}/m\mathbb{Z})$	Sistema completo de resíduos módulo m (ou só $\mathbb{Z}_m$)
$(\mathbb{Z}/m\mathbb{Z})^*$	Sistema reduzido de resíduos módulo m
$\varphi(m)$	Função φ de Euler
$(G, *)$	Grupo com a operação $*$ definida
$(A, +, \cdot)$	Anel com as operações de soma $+$ e produto $\cdot$ definidas
$\oplus$	Operação de soma entre classes de equivalência módulo m
$\odot$	Operação de produto entre classes de equivalência módulo m
$\#\Omega$	Ordem da curva elíptica
$K[x]$	Anel dos polinômios sobre o corpo K na variável x
$\left(\frac{a}{b}\right)$	Símbolo de Legendre
$GF(2^m)$	Corpo de Galois 2^m
$char(\mathbb{K})$	Característica do corpo $\mathbb{K}$
∞	Ponto no infinito
Δ	Discriminante de uma curva elíptica
$\gg i$	Deslocamento de i bits a direita
$\ll i$	Deslocamento de i bits a esquerda
$\&$	Operação lógica de E
$\|$	Operação lógica de OU
$\hat{}$	Operação lógica de OU exclusivo
$\mathrm{NAF}_w(K)$	Representação da forma não-adjacente de K com dimensão w
$O(\cdot)$	Notação assintótica O (big-O)

Resumo

Desde a introdução da criptografia assimétrica por W. Diffie e M. Hellman em 1976, o Problema do Logaritmo Discreto (PLD) tem sido usado sob várias formas. Uma variação do uso do PLD está em curvas elípticas, sobre a qual este trabalho se prontifica a descrever alguns tópicos relevantes. Uma vez que o PLD é mais difícil de ser resolvido em curvas elípticas, a chave criptográfica poderá ser substancialmente reduzida, sendo esta a grande vantagem de sistemas criptográficos baseados em curvas elípticas. Neste trabalho exporemos os detalhes de implementação de uma API (*Application Programming Interface*) para construção de protocolos que usam criptografia baseada em curvas elípticas bem como todo aparato teórico necessário para o entendimento.

Abstract

Since the introduction of asymmetric cryptography by W. Diffie and M. Hellman in 1976, the Discrete Logarithm Problem (DLP) has been used in various forms. A variation of the use of DLP is over elliptic curves, on which this work is to describe some relevant topics. Once the DLP is more difficult to be solved in elliptic curves, the cryptographic key can be substantially reduced, which is the great advantage of cryptographic systems based on elliptic curves. This work present the details of implementation of an API (Application Programming Interface) for construction of protocols that use cryptography based on elliptic curves and all apparatus necessary for the theoretical understanding.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

1 Introdução

Whitfield Diffie e Martin E. Hellman [7] propuseram uma interessante solução para o problema de dois usuários estabelecerem uma chave secreta em um canal de comunicação inseguro, que é considerada a primeira prática de criptografia de chave pública. Suponha que dois usuários, Alice e Bob, queiram estabelecer uma chave secreta compartilhada. Alice seleciona aleatoriamente um inteiro secreto a tal que $a \in \mathbb{Z}_p$ e envia para Bob $P_A = g^a \mod p$ ($a = 0,\ a = 1$ e $a = p - 1$ devem ser evitados [39]), sendo p um primo e g um gerador do grupo cíclico $\mathbb{Z}_p^*$. Analogamente, Bob escolhe um inteiro secreto $b \in \mathbb{Z}_p$ e envia à Alice $P_B = g^b$. Agora ambos usam sua chave secreta para obter uma chave secreta compartilhada $K = (P_A)^b = (P_B)^a = g^{ab}$, observe que a segurança deste protocolo está baseada na dificuldade computacional de calcular a dado g e p. Quando estas variáveis são relativamente grandes este problema se torna inviável, e é conhecido como Problema do Logaritmo Discreto (PLD). Taher ElGamal [8] propôs um algoritmo também baseado na dificuldade de resolver o PLD. Vamos supor que Bob queira enviar uma mensagem $m \in \mathbb{Z}_p^*$ para Alice. Inicialmente, Alice escolhe $s \in \mathbb{Z}_p$, e logo após computa $y = g^s \mod p$. Alice publica y junto com p e g, que será a sua chave pública, e s será a sua chave privada. Bob escolhe aleatoriamente $z \in \mathbb{Z}_p$ e calcula $c_1 = g^z \mod p$ e $c_2 = m \cdot y^z \mod p$, depois envia para Alice o criptograma (c_1, c_2). Para decifrar m, Alice computa $c_2 \cdot (c_1^s)^{-1} = m \cdot y^z \cdot (g^{sz})^{-1} = m \cdot (g^{sz}) \cdot (g^{sz})^{-1} = m \mod p$, recuperando assim a mensagem m. Desde então muitos algoritmos de criptografia assimétrica usam o PLD como base de sua segurança. Em 1985, V. Miller [23] e N. Koblitz [13], propuseram independentemente a aplicação de curvas elípticas em criptografia assimétrica. A segurança deste método está, mais uma vez, baseada no PLD. Este método criptográfico exige uma chave de comprimento consideravelmente menor que a chave usada em alguns clássicos da criptografia assimétrica, tais como o RSA, no entanto com segurança equivalente, ficando acessível a sistemas com restrições computacionais. Neste trabalho apresentaremos a experiência da implementação em *software* das operações básicas necessárias para a construção de protocolos de criptografia baseada em curvas elípticas. Mostraremos todo aparato algébrico para o entendimento da implementação. Começaremos na seção 2 introduzindo conceitos básicos, porém necessários, de teoria dos números. Na seção 3 discutiremos o conceito abstrato de curvas elípticas e mostraremos alguns resultados referente a este grupo aditivo. A seção 4 abordaremos a segurança de métodos

baseados em curvas elípticas. A próxima seção iremos introduzir alguns conceitos sobre criptografia: simétrica, assimétrica, funções de *hash* e assinatura. Já na seção 6 mostraremos a aplicação de curvas elípitcas em criptografia assimétrica. Finalmente, a seção 7 iremos explorar a implementação proposta neste trabalho. Nesta seção mostraremos o funcionamento dos principais algoritmos.

2 Conceitos Algébricos

Nesta seção iremos abordar os conceitos algébricos e tópicos básicos de teoria dos números que serão de suma importância para o entendimento da técnica que foi implementada. O leitor com maior inclinação matemática poderá omitir as subseções que se seguem até a seção 3 quando tocaremos pela primeira vez no principal tema desta monografia. Muitas demonstrações serão omitidas, no entanto, poderão ser encontradas em [34].

2.1 Relação de Equivalência

Sendo X um conjunto não-vazio, uma relação R só é uma relação de equivalência se satisfaz as seguintes propriedades abaixo:

i) **(reflexiva)** $(a, a) \in R$ para qualquer $a \in X$.

ii) **(simétrica)** Se $(a, b) \in R$ logo $(b, a) \in R$, para todo $a, b \in X$.

iii) **(transitiva)** Se $(a, b) \in R$ e $(b, c) \in R$, então $(a, c) \in R$ para quaisquer $a, b, c \in X$.

Exemplo 2.1.1 *Vamos verificar se a relação R_m em $\mathbb{Z} \times \mathbb{Z}$ é uma relação de equivalência tais que:*

$$(a, b) \in R_m \Leftrightarrow m|(a - b) \text{ com } m \in \mathbb{N}.$$

i)$(a, a) \in R_m$, *para todo* $a \in \mathbb{Z}$ $m|(a - a) \Rightarrow m|0$.

ii) *Se* $(a, b) \in R_m$, *então* $m|(a - b)$ *logo existe* $k \in \mathbb{Z}$ *tais que* $a - b = mk$ *ou seja* $b - a = m(-k)$ *então* $m|(b - a)$, *logo* $(b, a) \in R_m$.

iii) Se $(a,b) \in R_m$ *e* $(b,c) \in R_m$*, então* $m|(a-b)$ *e* $m|(b-c)$ *logo existe* $k_1, k_2 \in \mathbb{Z}$*, tais que* $a-b = mk_1$ *e* $b-c = mk_2$ *logo* $a-c = m(k_1+k_2)$ *então* $m|(a-c)$ *logo* $(a,c) \in R_m$*.*

R_m *é uma relação de equivalência [34], a qual denotaremos por* $a \equiv b \pmod{m}$.

2.2 Classes de Equivalência

Seja A um conjunto não–vazio e seja R uma relação de equivalência no conjunto A. Para todo $x \in A$ tal que x se relacione com a, chamamos de classe de equivalência do elemento a, denotado por $\bar{a}$ o conjunto

$$\bar{a} = \{x \in A : (x,a) \in R\}.$$

2.3 Operações Módulo m

Antes de entrarmos no foco desta seção vamos a uma importante definição

Definição 2.3.1 (MDC e MMC) *Sejam* a, b *inteiros diferentes de zero. Definimos o máximo divisor comum (MDC) entre* a *e* b *(denotado por* (a,b) *ou* $mdc(a,b)$*) pelo maior inteiro que divide ambos* a *e* b*. Analogamente, definimos o mínimo múltiplo comum (MMC) entre* a *e* b *(denotado por* $< a,b >$ *ou* $mmc(a,b)$*) como sendo o menor inteiro múltiplo tanto de* a *quanto de* b*.*

O que segue são algumas propriedades de $a \equiv b \pmod{m}$ sendo $a, b \in \mathbb{Z}$ e $m \in \mathbb{N}$. Com base na congruência acima equivale dizer que existe um $k \in \mathbb{Z}$ tal que $a = b + mk$, visto que $m|(a-b)$.

i) $a \equiv b \pmod{m} \Leftrightarrow a - b \equiv 0 \pmod{m}$.

ii) Se $a \equiv b \pmod{m}$ e $c \equiv d \pmod{m}$ então $(a+c) \equiv (b+d) \pmod{m}$.

iii) Se $a \equiv b \pmod{m}$ e $c \equiv d \pmod{m}$ então $(a-c) \equiv (b-d) \pmod{m}$.

iv) Se $a \equiv b \pmod{m}$ e $c \equiv d \pmod{m}$ então $ac \equiv bd \pmod{m}$.

v) Se $a \equiv b \pmod{m}$ então para todo $x \in \mathbb{Z}$, $ax \equiv bx \pmod{m}$.

vi) Se $d \in \mathbb{Z}$ tal que $mdc(d, m) = 1$ se $ad \equiv bd \pmod{m}$ então $a \equiv b \pmod{m}$.

vii) Se $a \equiv b \pmod{m}$ e $c \equiv d \pmod{m}$ então para todo $x \in Z$ $ax \equiv c \pmod{m} \Leftrightarrow bx \equiv d \pmod{m}$.

Definição 2.3.2 *Um conjunto de inteiros* $\{a_1, \ldots, a_n\}$ *é dito ser um sistema completo de resíduos módulo m se:*

i) $a_i \not\equiv a_j \pmod{m}$ *para todos* $i, j \in \{1, \ldots, n\}$ *e* $i \neq j$.

ii) *Para todo* $n \in \mathbb{Z}$ *existe* $i \in \{1, \ldots, n\}$ *tal que* $n \equiv a_i \pmod{m}$.

Um *sistema completo de resíduos módulo m* não necessariamente deve apresentar os restos menores que m e maiores ou iguais a *zero*, nesse caso podem apresentar os múltiplos dos possíveis restos de m, porém se $\{a_1, \ldots, a_n\}$ e $\{b_1, \ldots, b_k\}$ são *sistemas completos de resíduos módulo m* então $n = k$, ou seja, eles obrigatoriamente devem ter o mesmo número de elementos.

Definição 2.3.3 *Um conjunto* $\{a_1, \ldots, a_r\}$ *é denominado um sistema reduzido de resíduos módulo m se:*

i) $(a_i, m) = 1$, *para todo* $i \in \{1 \ldots, r\}$.

ii) $a_i \not\equiv a_j \pmod{m}$ *se* $i, j \in \{1, \ldots, r\}$ *com* $i \neq j$.

iii) *Para todo* $n \in \mathbb{Z}$ *tal que* $(m, n) = 1$, *existe* $i \in \{1, \ldots, r\}$ *tal que* $n \equiv a_i \pmod{m}$.

Esse conjunto será denotado por $(\mathbb{Z}/m\mathbb{Z})^*$.

2.4 A Função φ de Euler

Em um *sistema completo de resíduos módulo m* temos sempre um número bem definido de elementos no conjunto, ou seja, temos sempre m elementos, entretanto, um *sistema reduzido de resíduos módulo m* exige um mecanismo

um pouco mais delicado para descobrir o número de elementos nele contido. O que a função φ de Euler faz é retornar a cardinalidade de um *sistema reduzido de resíduos módulo m.* Logo definimos:

$$\varphi : \mathbb{N} \to \mathbb{N}$$

$$\varphi(m) = \#\{x \in \mathbb{N} : x \leq m \text{ e } (x, m) = 1\}.$$

A seguir serão descritas algumas propriedades importantes da função de Euler, e logo após aplicaremos algumas propriedades através de exemplos.

i) $\varphi(1) = \varphi(2) = 1$.

ii) $\varphi(p) = p - 1$, isso se p for um primo.

iii) Se $m, n \in \mathbb{N}$, e $(m, n) = 1$ então $\varphi(m \cdot n) = \varphi(m) \cdot \varphi(n)$.

iv) Se p é primo e $\alpha \in \mathbb{N}$ logo $\varphi(p^\alpha) = p^\alpha(1 - \frac{1}{p})$.

v) Se P_n é um conjunto formado por fatores primos de $n \in \mathbb{N}$ temos: $\varphi(n) = n \prod_{p \in P_n}(1 - \frac{1}{p})$.

vi) Se $n \in \mathbb{N}$ e $D(n) = \{d \in \mathbb{N} : d|n\}$, então $\sum_{d \in D(n)} \varphi(d) = n$.

vii) Se $m \in \mathbb{N}$ e se $\varphi(m)|(m - 1)$ logo não existe nenhum primo p tal que $p^2|m$.

Exemplo 2.4.1 *Repare que* $(\mathbb{Z}/15\mathbb{Z})^* = \{1, 2, 4, 7, 8, 11, 13, 14\}$ *logo* $\varphi(15) = 8$*, se tomarmos um número primo ao invés de 15, como por exemplo 13, temos que* $(\mathbb{Z}/13\mathbb{Z})^* = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$ *logo* $\varphi(13) = 13 - 1 = 12$.

Exemplo 2.4.2 *Como* $\varphi(4) = 2$ *e* $\varphi(7) = 6$ *visto que* $(7, 4) = 1$ *temos que* $\varphi(4 \cdot 7) = \varphi(4) \cdot \varphi(7) = 2 \cdot 6 = 12$.

Exemplo 2.4.3 *Visto que* $\varphi(27) = \varphi(3^3)$ *como três é primo, temos que* $\varphi(3^3) = 27 \cdot (1 - \frac{1}{3}) = 18$.

Exemplo 2.4.4 *Note que* $120 = 2^3 \cdot 3 \cdot 5$ *logo temos que* $\varphi(120) = 120 \cdot (1 - \frac{1}{2}) \cdot (1 - \frac{1}{3}) \cdot (1 - \frac{1}{5}) = 32$.

Exemplo 2.4.5 *Como* $D(6) = \{1, 2, 3, 6\}$*(conjunto dos números que divide seis) temos que* $\varphi(1) + \varphi(2) + \varphi(3) + \varphi(6) = 1 + 1 + 2 + 2 = 6$.

Outra propriedade de suma importância em teoria dos números, é o Teorema de Euler, do qual trataremos agora. Omitiremos as demonstrações de alguns resultados que seguem. De qualquer modo todas as provas dos teoremas e lemas poderão ser encontradas em [34] e [11]:

Teorema 2.4.6 (Euler) *Se* $m \in \mathbb{N}$ *e* $a \in \mathbb{N}$ *satisfazem* $(m, a) = 1$ *temos que:*

$$a^{\varphi(m)} \equiv 1 \pmod{m}.$$

Demonstração: *Seja* $\{r_1, \ldots, r_{\varphi(m)}\}$ *uma sistema reduzido de resíduos módulo* m. *Se* $(a, m) = 1$ *logo* $\{ar_1, \ldots, ar_{\varphi(m)}\}$ *também é um sistema reduzido de resíduos módulo* m. *Logo para todo* $i \in \{1, 2, 3, \ldots, \varphi(m)\}$ *existe* $j \in \{1, 2, 3, \ldots, \varphi(m)\}$ *tal que* $ar_i \equiv r_j \pmod{m}$, *então*

$$ar_1 \cdot ar_2 \cdot \ldots \cdot ar_{\varphi(m)} \equiv r_1 \cdot r_2 \cdot \ldots \cdot r_{\varphi(m)} \pmod{m}$$

e

$$a^{\varphi(m)} \cdot r_1 \cdot \ldots \cdot r_{\varphi(m)} \equiv r_1 \cdot \ldots \cdot r_{\varphi(m)} \pmod{m}$$

portanto

$$a^{\varphi(m)} \equiv 1 \pmod{m}.$$

O próximo resultado é conhecido como *Pequeno Teorema de Fermat* o qual está diretamente ligado ao teorema anterior.

Corolário 2.4.7 (Fermat) *Se* p *é um número primo. Logo, para qualquer* $n \in \mathbb{N}$. *Temos:*

$$n^p \equiv n \pmod{p}.$$

Agora falaremos de importantes estruturas algébricas.

2.5 Grupos, Anéis e Corpos

Seja $G \neq \emptyset$ um conjunto e $*$ uma operação qualquer de $G \times G$ em G, ou seja:

$$* : G \times G \to G$$
$$(g_1, g_2) \longmapsto g_1 * g_2.$$

Dizemos que $(G, *)$ é um grupo se G é fechado para operação $*$ e se as seguintes propriedades são satisfeitas:

G1 (Associativa) Para todo $g_1, g_2, g_3 \in G$, $g_1 * (g_2 * g_3) = (g_1 * g_2) * g_3$.

G2 (Elemento neutro) Para todo $g \in G$ existe $e \in G$ tal que $e * g = g * e = g$.

G3 (Elemento inverso) Para todo $g \in G$ existe $g^{-1} \in G$ tal que $g * g^{-1} = g^{-1} * g = e$.

Um grupo é dito um *grupo abeliano* se:

G4 (Comutativo) Para todo $g_1, g_2 \in G$, $g_1 * g_2 = g_2 * g_1$.

Seja $(G, *)$ um grupo. Dizemos que J é um subgrupo de G, se $(J, *)$ é um grupo, denotamos esse fato por: $(J, *) \leq (G, *)$. Definimos a *ordem de um grupo* G como sendo o número de elementos de G, e pelo *Resultado de Lagrange*, se $(J, *) \leq (G, *)$ então a ordem J divide a ordem de G.. Seja $A \neq \emptyset$ na qual as duas operações "+" e "$\cdot$" estão definidas. Dizemos que $(A, +, \cdot)$ é um anel quando as seguintes propriedades são satisfeitas:

A1 $(A, +)$ é um *grupo abeliano.*

Para quaisquer $x, y, z \in A$.

A2 $x \cdot (y + z) = x \cdot y + x \cdot z$, e $(x + y) \cdot z = x \cdot z + y \cdot z$.

A3 $(x \cdot y) \cdot z = x \cdot (y \cdot z)$.

Alguns anéis satisfazem outras propriedades além das anteriores, os mesmos recebem nomes diferenciados.

A4 (Anel comutativo) $x \cdot y = y \cdot x$, para quaisquer $x, y \in (A, +, \cdot)$.

A5 (Anel com unidade) Existe $1 \in A$ tal que $x \cdot 1 = 1 \cdot x = x$, para todo $x \in A$.

Se o anel é comutativo e com unidade e se ele satisfaz **A6** então é um

A6 (Domínio de integridade) Para todo $x, y \in A$, se $x \cdot y = 0$ logo $x = 0$ ou $y = 0$.

Se A é um domínio de integridade e se ele satisfaz **A7** então esse conjunto é um

C1 (Corpo) Existe $x^{-1} \in A$ para todo $x \in A$ se $x \neq 0$ tal que $x \cdot x^{-1} = x^{-1} \cdot x = 1$.

Agora vamos definir as seguintes operações no conjunto dos inteiros módulo m $\mathbb{Z}_m$:

$$\oplus : \mathbb{Z}_m \times \mathbb{Z}_m \to \mathbb{Z}_m$$
$$(a, b) \mapsto a \oplus b = a + b \pmod{m}$$

$$\odot : \mathbb{Z}_m \times \mathbb{Z}_m \to \mathbb{Z}_m$$
$$(a, b) \mapsto a \odot b = a \cdot b \pmod{m}$$

Repare que $(\mathbb{Z}_m, \oplus)$ é um *grupo abeliano,* e $(\mathbb{Z}_m, \oplus, \odot)$ é um *anel comutativo e com unidade.*

Um grupo $(G, *)$ é chamado de um grupo cíclico quando podemos fixar um elemento $a \in G$ e aplicar sucessivamente $*$ e obter qualquer $g \in G$, ou seja, para todo $g \in G$, temos que $g = a * \ldots * a$. Chamaremos o elemento a de *gerador de* $(G, *)$.

Exemplo 2.5.1 *Seja* $((\mathbb{Z}_{11})^*, \odot)$ *um grupo com operação* $\odot$. *Fixando* $\bar{2}$ *e aplicando sucessivamente* $\odot$, *temos que:*

$$\begin{array}{ll} \bar{2}^1 = \bar{2} & \bar{2}^6 = \bar{9} \\ \bar{2}^2 = \bar{4} & \bar{2}^7 = \bar{7} \\ \bar{2}^3 = \bar{8} & \bar{2}^8 = \bar{3} \\ \bar{2}^4 = \bar{5} & \bar{2}^9 = \bar{6} \\ \bar{2}^5 = \bar{10} & \bar{2}^{10} = \bar{1} \end{array}$$

Logo $((\mathbb{Z}/11\mathbb{Z})^*, \odot)$ *é um grupo cíclico e tem como gerador* $\bar{2}$.

2.6 Anel dos Polinômios

Sendo A um anel. Definimos um polinômio sobre A com variável x como sendo uma expressão do tipo:

$$p(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \ldots + a_nx^n + \ldots$$

Iremos denotar por $A[x]$ o conjunto dos polinômios sobre A com variável x, e definimos as seguintes operações em $A[x]$:

Seja $p(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \ldots + a_nx^n + \ldots$ e $q(x) = b_0 + b_1x + b_2x^2 + b_3x^3 + \ldots + b_nx^n + \ldots$ polinômios pertencentes a $A[x]$, temos que:

i) $p(x) + q(x) = (a_0 + b_0) + (a_1 + b_1)x + \ldots + (a_k + b_k)x^k + \ldots$ para todo $k \in \mathbb{Z}^+$.

ii) $p(x) \cdot q(x) = c_0 + c_1x + \ldots + c_kx^k + \ldots$

na qual,

$$c_0 = a_0 b_0$$
$$c_1 = a_0 b_1 + a_1 b_0$$
$$c_2 = a_0 b_2 + a_1 b_1 + a_2 b_0$$
$$c_k = a_0 b_k + a_1 b_{k-1} + \ldots + a_j b_{k-j} + \ldots + a_k b_0.$$

Como "+" e "·" estão bem definidas, logo $A[x]$ também é um anel. Agora iremos tratar de propriedades elementares do grau de um polinômio pertencente a $A[x]$:

i) Se $f(x), g(x) \neq 0$, então $gr(f \cdot g) = gr(f) + gr(g)$.

ii) Se $f(x) + g(x) \neq 0$, logo $gr(f + g) \leq max\{gr(f), gr(g)\}$.

iii) Se $f(x) \cdot g(x) = 1$, logo temos que $gr(f \cdot g) = gr(1) = 0$ e $gr(f) + gr(g) = 0$.

Definição 2.6.1 *Sendo K um corpo e $K[x]$ um anel de polinômios sobre K na variável x, se $f(x), g(x) \in K[x]$ e $g(x) \neq 0$ dizemos que f(x) divide g(x) em $K[x]$ se existir $g_1(x) \in K[x]$ tais que $f(x) = g_1(x)g(x)$.*

Repare que essa definição decorre do conceito de divisibilidade nos inteiros, só que agora com um enfoque polinomial.

Definição 2.6.2 (MDC no anel dos polinômios) *Sendo $f(x)$ e $g(x)$ dois polinômios não nulos. Dizemos que $d(x) \in K[x]$ é um MDC para $f(x), g(x) \in K[x]$ se, se somente se:*

i) $d(x)|f(x)$ e $d(x)|g(x)$.

ii) Se $d'(x) \in K[x]$ é tal que $d'(x)|f(x)$ e $d'(x)|g(x)$ logo $d'(x)|d(x)$.

Definição 2.6.3 *Seja $f(x) = a_0 + a_1 x + \ldots + a_n x^n$ um polinômio pertencente a $K[x]$ chamaremos o coeficiente a_n de coeficiente dominante de $f(x)$. Se o coeficiente dominante for igual a 1, definimos $f(x)$ como um polinômio mônico ou unitário.*

2.7 Índices

Definição 2.7.1 *Sejam $a \in Z$ e $m \in \mathbb{N}$ tais que sejam primos entre si (ou seja $(a, m) = 1$). Definiremos o expoente de a módulo m ao menor natural t tal que $a^t \equiv \pmod{m}$. Denotaremos $t = \exp_m(a)$.*

Definição 2.7.2 *Definiremos a raiz primitiva de $m \in \mathbb{N}$ como sendo um inteiro g tal que $(a, g) = 1$ e $\exp_m(g) = \varphi(m)$.*

Definição 2.7.3 *Se a e m são primos entre si, e g uma raiz primitiva módulo m. Dizemos que t é um índice de a módulo m na base g se:*

$$g^t \equiv a \pmod{m}.$$

Sendo $t \in \{1, \ldots, \varphi(m)\}$. Denotamos esse fato por $ind_g(a) = t$.

Lema 2.7.4 *Sendo g uma raiz primitiva módulo m e $m \in \mathbb{N}$, se:*

$$g^s \equiv g^t \pmod{m},$$

então

$$s \equiv t \pmod{\varphi(m)},$$

com $s, t \in \mathbb{N}$.

Iremos listar agora três importantes propriedades dos *índices módulo m*.

i) Se $a \equiv b \pmod{m}$, logo $ind_g(a) \equiv ind_g(b) \pmod{\varphi(m)}$.

ii) $ind_g(ab) \equiv ind_g(a) + ind_g(b) \pmod{\varphi(m)}$.

iii) $ind_g(a^n) \equiv n(ind_g(a)) \pmod{\varphi(m)}$.

Teorema 2.7.5 *Se $m \in \mathbb{N}$ possui ao menos uma raiz primitiva, sendo, $a \in \mathbb{Z}$, $k \in \mathbb{Z}$ e $d = (k, \varphi(m))$ logo a equação:*

$$x^k \equiv a \pmod{m}.$$

Tem uma solução se, e somente se:

$$a^{\frac{\varphi(m)}{d}} \equiv 1 \pmod{m}.$$

Definição 2.7.6 *Seja p um primo, se a equação $x^2 \equiv a \pmod{p}$ tem solução com $x \in \{1, \ldots, p-1\}$ indicamos a como resíduo quadrático módulo p. Definimos x como a raiz quadrada de $a \pmod{p}$.*

Teorema 2.7.7 (Teste de Euler) *Seja $p > 2$ um número primo, e a um inteiro, se p não divide a, logo a é um resíduo quadrático módulo p se, e somente se:*

$$a^{\frac{p-1}{2}} \equiv 1 \pmod{p}.$$

Definição 2.7.8 (Símbolo de Legendre) *Seja $p > 2$ um primo e $a \in \mathbb{N}$, tais que, $(a, p) = 1$. Definimos o Símbolo de Legendre por:*

$$\begin{cases} (\frac{a}{p}) = 1, \textit{se } a \textit{ é resíduo quadrático módulo } p \\ (\frac{a}{p}) = -1, \textit{se } a \textit{ não é resíduo quadrático módulo } p \end{cases}$$

Mostraremos agora algumas propriedades do *Símbolo de Legendre*.

i) Se $a \equiv b \pmod{p}$, logo, $(\frac{a}{p}) = (\frac{b}{p})$.

ii) $a^{\frac{p-1}{2}} \equiv (\frac{a}{p}) \pmod{p}$.

iii) $(\frac{ab}{p}) = (\frac{a}{p})(\frac{b}{p})$.

Definição 2.7.9 (Resto Principal) *Definimos o resto principal módulo m de um inteiro a qualquer como sendo o único $r_m(a) \in (-\frac{m}{2}, \frac{m}{2}]$ tal que $a \equiv r_m(a) \pmod{m}$.*

Lema 2.7.10 (Gauss) *Seja $p > 2$ um primo e $a \in \mathbb{N}$ talque $p \nmid a$. Seja λ o número de elementos cujos restos principais módulo p são negativos, então $(\frac{a}{p}) = (-1)^{\lambda}$.*

Lema 2.7.11 *Seja p um número primo maior que 2. Então 2 é resíduo quadrático módulo p se, e somente se, p é da forma $8k + 1$ ou $8k - 1$, se considerarmos o número 3, p deve ser, restritamente, da forma $12s + 1$ ou $12s - 1$.*

Definição 2.7.12 *Seja $s(a)$ o valor absoluto do resto principal de $a \cdot s$ módulo p. Sendo p um primo e $s \in \{1, \ldots, \frac{p-1}{2}\}$ e $a \in \{1, \ldots, p-1\}$, logo temos*

$$e_s(a) = \begin{cases} 1, \textit{se o resto principal módulo } p \textit{ de } a \cdot s \textit{ for maior ou igual a zero} \\ -1, \textit{se o resto principal módulo } p \textit{ de } a \cdot s \textit{ for menor que zero} \end{cases}$$

Observe que $a \cdot s \equiv e_s(a) \cdot s(a) \pmod{p}$.

Lema 2.7.13 *De posse das informações anteriores temos que:*

$$(\frac{a}{p}) = \prod_s e_s(a).$$

Enfim chegamos ao principal resultado desta seção, o *Teorema da Reciprocidade Quadrática de Gauss.*

Teorema 2.7.14 (Gauss) *Sejam p e q primos, ambos maiores que dois. Logo temos:*

$$(\frac{p}{q})(\frac{q}{p}) = (-1)^{\frac{(p-1)(q-1)}{4}}.$$

2.8 O Corpo de Galois $GF(2^m)$

O Corpo de Galois tem se mostrado muito útil em teoria dos códigos devido a suas propriedades. Tal corpo contém um número finito de elementos, portanto é viável seu uso em criptografia através de estruturas como o grupo formado pelos pontos de uma curva elíptica. Esta seção se prontifica a introduzir brevemente os principais tópicos pertinentes ao conteúdo necessário para a compreensão dos conceitos que se seguem. Inicialmente podemos destacar o seguinte resultado:

Teorema 2.8.1 *Se q é um inteiro positivo, então existe um corpo $\mathbb{K}$ tal que $\#\mathbb{K} = q$ ($\#\mathbb{K}$ denota a cardinalidade do corpo $\mathbb{K}$) se, e somente se, $q = p^m$ para algum inteiro $m \geq 1$ e sendo p um primo.*

O teorema acima diz, claramente, que a ordem de um corpo finito é sempre um primo ou uma potência de um primo. O corpo $GF(p)$ também escrito como $\mathbb{F}_p$, onde $m = 1$ recebe um nome diferenciado, este será denotado por corpo primo e sua ordem é $p - 1$. Se $a = b$ em $GF(p)$ isto implica que $a \equiv b \pmod{p}$. Sendo assim, $GF(p) = \{0, \ldots, p-1\}$. Quando $m > 1$

podemos representar os elementos de $GF(p^m)$ como polinômios de grau menor que m, onde os coeficientes pertencem a $GF(p)$, mais a frente iremos entrar neste mérito. Neste trabalho iremos nos concentrar em construir o corpo $GF(2^m)$, ou seja, com $p = 2$. Este corpo é o mais usado em criptografia com curvas elípticas. Para garantir uma segurança razoável estima-se o valor de m em torno de 160. Na maioria dos casos iremos trabalhar com a representação polinomial dos elementos de $GF(2^m)$. Portanto, os coeficientes destes polinômios pertencem ao conjunto $\mathbb{Z}_2 = \{0, 1\}$. Os elementos de $GF(2^m)$ podem ser vistos como polinômios do tipo

$$f(X) = a_{m-1}X^{m-1} + a_{m-2}X^{m-2} + \ \ldots \ + a_1X + a_0 = \sum_{i=0}^{m-1} a_iX^i,$$

onde $a_i \in \{0, 1\}$.
Abaixo iremos listar os elementos de $GF(2^3)$:

$$GF(2^3) = \{0,\ 1,\ X,\ X+1,\ X^2,\ X^2+1,\ X^2+X,\ X^2+X+1\}.$$

Podemos também representar os elementos de $GF(2^m)$ pelos seus respectivos coeficientes da forma $(a_{m-1}\ a_{m-2} \ldots a_1\ a_0)$, ou seja, *strings* de m bits. No caso de $GF(2^3)$ podemos escrevê-lo da forma

$$GF(2^3) = \{(000), (001), (010), (011), (100), (101), (110), (111)\}.$$

Como reduzimos os coeficientes módulo 2 podemos definir a soma e a subtração como um XOR bit a bit entre os elementos de $GF(2^m)$.

Exemplo 2.8.2 *Considere os dois polinômios* $f(X) = X^5 + X^4 + X^3 + 1$ *e* $g(X) = X^7 + X^6 + X^4 + X^3 + X^2$ *temos que:*

$$(X^5 + X^4 + X^3 + 1) \oplus (X^7 + X^6 + X^4 + X^3 + X^2 + X) =$$

$$= (X^7 + X^6 + X^5 + (1+1)X^4 + (1+1)X^3 + X^2 + X + 1) =$$

fazendo os coeficientes módulo 2 *temos*

$$= (X^7 + X^6 + X^5 + 0 \cdot X^4 + 0 \cdot X^3 + X^2 + X + 1) =$$

$$= (X^7 + X^6 + X^5 + X^2 + X + 1).$$

a representação acima constitui uma representação polinomial. Abaixo vemos outras duas representações, observe que a representação hexadecimal foi obtida através da representação binária.

$$\begin{array}{lll} (00111001) \oplus (11011110) & = (11100111) & \textit{representação binária.} \\ \{39\} \oplus \{de\} & = \{e7\} & \textit{representação hexadecimal.} \end{array}$$

As operações de adição e subtração são de grande importância pois delas derivamos as operações de multiplicação e divisão respectivamente. No entanto a multiplicação não é tão simples quanto a adição e a subtração em $GF(2^m)$. A multiplicação em $GF(2^m)$ segue inicialmente como a multiplicação usual de polinômios, após obtido o produto primeiro reduziremos seus coeficientes módulo p (neste caso $p = 2$). O próximo passo é verificar o grau do polinômio resultante. Para isso vamos definir o grau de um polinômio em $GF(2^m)$.

Definição 2.8.3 *O grau de um polinômio $f \in GF(2^m)$ é o maior grau entre o seus monômios cujo seu coeficiente é diferente de zero. Denotaremos o grau de f como $gr(f)$.*

Exemplo 2.8.4 *Dado $f(X) = X^6 + X^3 + X + 1$ e $h(X) = 1$ temos que $gr(f) = 6$ e $gr(h) = 0$.*
É fácil perceber que se $f(X)$, $h(X) \in GF(2^m)$ logo $gr(f(X) \cdot h(X)) = gr(f(X)) + gr(h(X)) = 6$.

Na multiplicação, deveremos analisar o grau do polinômio resultante após a redução de seus coeficientes módulo 2. Caso o grau deste polinômio for maior ou igual a m deveremos reduzi-lo este grau através de um polinômio irredutível. Em outras palavras, extraímos o resto da divisão entre o polinômio resultante e um polinômio irredutível. Um polinômio irredutível nada mais é que um polinômio de grau m que não pode ser fatorado por outros polinômios de grau menor ou igual a m. Em $GF(2^8)$ usaremos o polinômio irredutível $m(X) = X^8 + X^4 + X^3 + X + 1$. De maneira geral, o número de polinômios irredutíveis de grau m sobre o corpo $GF(q)$ onde $q = p^m$ é dado por:

$$L_q(m) = \frac{1}{m} \sum_{d|m} \mu(\frac{m}{d}) q^d,$$

onde $\mu(m)$ denota a função de Möbius (veja o apêndice C). Assim como em $GF(p)$, onde todas as operações estão fechadas módulo p, em $GF(2^m)$

m	polinômios irredutíveis
1	$X+1,\ X$
2	X^2+X+1
3	$X^3+X+1,\ X^3+X^2+1$
4	$X^4+X+1,\ X^4+X^3+X^2+X+1,\ X^4+X^3+1$

Tabela 1: Polinômios irredutíveis até o grau 4.

estas operações estão fechadas módulo um polinômio irredutível de grau m. A tabela 1 mostra os polinômios irredutíveis até o grau 4.

Na verdade, se usarmos um polinômio que não seja irredutível este conjunto não forma um corpo, pois nem todo elemento diferente de zero possuirá um inverso multiplicativo, ficando este conjunto restrito a um anel. Logo, o conjunto destes polinômios possui uma inversa se, e somente se o polinômio no qual calculamos o resto é irredutível. Para calcular o inverso multiplicativo para polinômio usa-se o Algoritmo de Euclides Estendido para polinômios (Veja o apêndice D). Abaixo mostraremos como se obtém um produto entre dois polinômios em $GF(2^8)$.

Exemplo 2.8.5 *Vamos calcular o produto entre* $f(X) = X^5+X^3+X^2+X$ *e* $h(X) = X^4+X^2+X+1$*, onde* $g(X) = f(X)\cdot h(X)$*, logo*

$$f(X)\cdot g(X) = (X^5+X^3+X^2+X)\cdot(X^4+X^2+X+1) =$$

$$= X^9+2X^7+2X^6+3X^5+2X^4+3X^3+2X^2+X =$$

$$g(X) = X^9+X^5+X^3+X.$$

Conforme visto usaremos o polinômio irredutível $m(X) = X^8+X^4+X^3+X+1$. Como o grau de $g(X)$ é maior ou igual a 8 precisamos reduzi-lo módulo $m(X)$. Para isso calculamos um $r(X)$ tal que $g(X) = q(X)\cdot m(X)+r(X)$ onde $0 \leq r(X) < g(X)$. Em outras palavras temos que $g(X) \equiv r(X) \mod m(X)$. Abaixo segue um esquema para este cálculo

$$\begin{array}{llll} X^9 & +X^5 \quad +X^3 \quad +X & | & \underline{X^8+X^4+X^3+X+1} \\ \underline{X^9} & \underline{+X^5+X^4} \quad \underline{+X^2+X} & & X \\ & X^4+X^3+X^2 & & \end{array}$$

Onde obtemos que $r(X) = X^4 + X^3 + X^2$. Uma propriedade que é muito apreciada em criptografia é o fato dos elementos de $GF(2^m)$ diferentes de zero formarem um grupo cíclico chamado de grupo multiplicativo de $GF(2^m)$ de ordem $2^m - 1$. As tabelas 2 e 3 mostram, em $GF(2^2)$, a adição e a multiplicação entre seus respectivos elementos. Na multiplicação usamos o polinômio irredutível $m(X) = X^2 + X + 1$.

$\oplus$	0	1	X	$X+1$
0	0	1	X	$X+1$
1	1	0	$X+1$	X
X	X	$X+1$	0	1
$X+1$	$X+1$	X	1	0

Tabela 2: Soma em $GF(2^2)$.

$\otimes$	0	1	X	$X+1$
0	0	0	0	0
1	0	1	X	$X+1$
X	0	X	$X+1$	1
$X+1$	0	$X+1$	1	X

Tabela 3: Multiplicação em $GF(2^2)$.

O algoritmo 1 mostra uma técnica de multiplicação em $GF(2^m)$.

Como os elementos de $GF(2^m)$ formam um grupo cíclico é natural que haja um elemento gerador deste grupo. Chamaremos este gerador de polinômio primitivo, que é também um polinômio irredutível. Se g é um gerador de $GF(2^m)$ logo obtemos todos elementos de $GF(2^m) - \{0\}$ a partir das potências de g. Sendo assim

$$GF(2^m) = \{0, g^1, \ldots, g^{2^m-2}, g^{2^m-1}\}.$$

Agora iremos gerar o conjunto $GF(2^4)-\{0\}$ a partir das potências em $GF(2^4)$ do polinômio primitivo (gerador) $g(X) = X^2 + 1$ usaremos como polinômio irredutível $m(X) = X^4 + X + 1$, ou seja, toda a aritmética seguinte será calculada em função do resto da divisão pelo polinômio $m(X) = X^4 + X + 1$.

Algoritmo 1: Shift-and-add em $GF(2^m)$ (versão direita para esquerda).

Entrada: $a(X), b(X) \in GF(2^m)$
Saída: $c(X) = a(X) \cdot b(X) \mod f(X)$

```
1  início
2      se a_0 = 1 então
3          c(X) ← b(X);
4      senão
5          c(X) ← 0;
6      para i = 1 até m − 1 faça
7          b(X) ← b(X) · X mod f(X);
8          se a_i = 1 então
9              c(X) = c(X) + b(X);
10     retorna c(X);
11 fim
```

Os elementos de $GF(2^4)$ gerados por $g(X) = X^2 + 1$ em representação polinomial e binária são

$$\begin{aligned}
(g(X))^1 &= X^2 + 1 && = (0101)\\
(g(X))^2 &= X && = (0010)\\
(g(X))^3 &= X^3 + X && = (1010)\\
(g(X))^4 &= X^2 && = (0100)\\
(g(X))^5 &= X^2 + X + 1 && = (0111)\\
(g(X))^6 &= X^3 && = (1000)\\
(g(X))^7 &= X^3 + X^2 + X && = (1110)\\
(g(X))^8 &= X + 1 && = (0011)\\
(g(X))^9 &= X^3 + X^2 + X + 1 && = (1111)\\
(g(X))^{10} &= X^2 + X && = (0110)\\
(g(X))^{11} &= X^3 + X^2 + 1 && = (1101)\\
(g(X))^{12} &= X^3 + X^2 && = (1100)\\
(g(X))^{13} &= X^3 + 1 && = (1001)\\
(g(X))^{14} &= X^3 + X + 1 && = (1011)\\
(g(X))^{15} &= 1 && = (0001).
\end{aligned}$$

Podemos, neste momento, destacar dois fatos importantes relacionados ao elemento gerador de $GF(2^m)$.
Para $i, j \in \mathbb{N}$, temos:

1. $g^i g^j = g^{i+j} = g^{i+j \mod 2^m-1}$ em $GF(2^m)$.
2. $g^{ij} = g^{ij \mod 2^m-1}$ em $GF(2^m)$.

Sempre calculamos o resto entre o expoente de um elemento de $GF(2^m)$ e 2^m-1, sendo assim no exemplo acima temos que $(g(X))^{15} = (g(X))^{15 \mod 2^4-1} = ((g(X))^0 = 1$, ou seja, para todo $f \in GF(2^m)$ com $f \neq 0$ temos que $f^{2^m-1} = f^0 = 1$, sendo f um polinômio primitivo ou não. De modo geral, o número de polinômios primitivos (geradores) de grau m sobre o corpo $GF(q)$ é dado por:

$$a_q(m) = \frac{\varphi(q^m - 1)}{m},$$

onde $\varphi(m)$ denota a função de Euler. Vamos resumir algumas definições importantes em $GF(p^m)$:

1. $g(X)|f(X)$ se existe $q(X)$ tal que $f(X) = q(X)g(X)$.
2. $h(X) = g(X) \mod f(X)$ se $f(X)|[h(X) - g(X)]$.
3. $grau(f(X))$ é o maior expoente de $f(X)$.

Abaixo algumas propriedades de $GF(p^m)$:

1. Para cada $g(X)$ e $f(X)$ pertencentes a $GF(p^m)$ existem únicos $r(X)$ e $q(X)$ em $GF(p^m)$ tais que $g(X) = q(X)f(X) + r(X)$.
2. Se $g(X),\ f(X) \in GF(p^m)$ logo $grau(f(X)g(X)) = grau(f(X)) + grau(g(X))$.
3. Para todo $g(X),\ f(X) \in GF(p^m)$ temos que $(g(X)+f(X))^p = g(X)^p + f(X)^p$.

3 Álgebra das Curvas Elípticas

Nesta seção iremos apresentar uma breve introdução à teoria das curvas elípticas de modo mais geral, mais a frente iremos discutir estas curvas sobre

estruturas mais específicas. Antes de definirmos uma curva elíptica iremos definir o conceito de característica de um corpo $\mathbb{K}$ qualquer. Este conceito é, de certa maneira, um pré-requisito para tratarmos de curvas elípticas.

Definição 3.0.6 *A característica de um corpo* $\mathbb{K}$*,* $char(\mathbb{K})$*, é o número de vezes que a identidade multiplicativa de* $\mathbb{K}$ *deve ser adicionada a si mesma para que o resultado seja igual à identidade aditiva. Se o resultado nunca for a identidade aditiva, define-se* $char(\mathbb{K}) = 0$ *[5].*

Exemplo 3.0.7 *A característica de* $\mathbb{Z}_p$ *é* p*, já que* $1 + 1 + \ldots + 1$ *(*p *vezes) é* p*, e obviamente* $p \equiv 0 \mod p$*. A característica do corpo* $\mathbb{R}$ *dos números reais é zero, já que* $1 + 1 + \ldots + 1$ *nunca terá zero como resultado.*

Definição 3.0.8 *Uma curva elíptica sobre um corpo* $\mathbb{F}$ *(assumiremos sempre que* $\mathbb{F}$ *é um corpo de característica maior que* 3*) é o lugar geométrico dos pontos* $(x, y) \in \mathbb{F} \times \mathbb{F}$ *que satisfazem a equação*

$$y^2 + axy + by = x^3 + cx^2 + dx + e \tag{1}$$

mais um ponto, chamado de ponto no infinito, que será denotado por ∞*.*

Podemos simplificar a equação (1) deixando na forma

$$y^2 = x^3 + ax + b \tag{2}$$

com $a, b \in \mathbb{F}$. Esta curva deve ser uma curva não-singular, ou seja, não possui raízes múltiplas, para tanto precisamos ter:

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

A figura 1 e a figura 2 mostram duas curvas elípticas sobre o corpo $\mathbb{R}$. É importante ressaltar quão diferentes ficam as curvas após uma pequena variação em seus parâmetros.

Uma das propriedades mais interessantes das curvas elípticas, é o fato de seus pontos formarem uma estrutura de grupo, para isso, vamos definir a "soma" entre seus pontos. Como foi visto, nós contamos com um ponto chamado de ponto no infinito ∞; este ponto desempenhará um importante papel, pois será o elemento neutro da soma. Logo, se Ω é uma curva elíptica sobre um corpo $\mathbb{F}$, e temos o ponto $P \in \Omega$, então:

$$P + \infty = P = \infty + P.$$

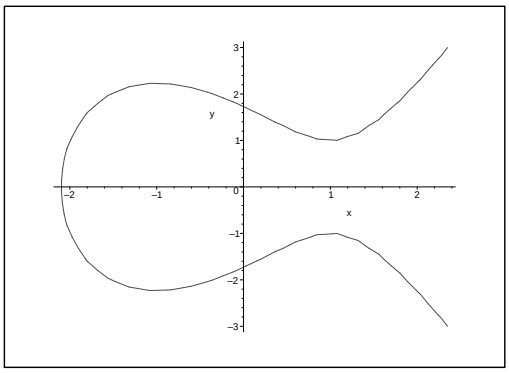

Figura 1: $y^2 = x^3 - 3x + 3$.

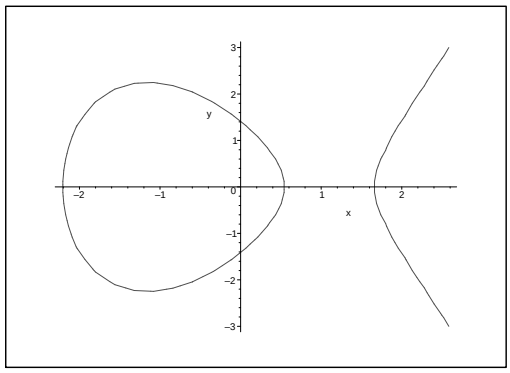

Figura 2: $y^2 = x^3 - 4x + 2$.

o simétrico de um ponto $P = (x, y)$ é o ponto $-P = (x, -y)$. Se somarmos P com o seu simétrico $-P$ iremos, naturalmente, obter o ponto no infinito, ou seja

$$P + (-P) = \infty = (-P) + P.$$

A figura 3 mostra como se obtém geometricamente o simétrico de um ponto $P \in \Omega$. Sejam P e Q dois pontos de uma curva elíptica sobre o corpo $\mathbb{R}$ dos

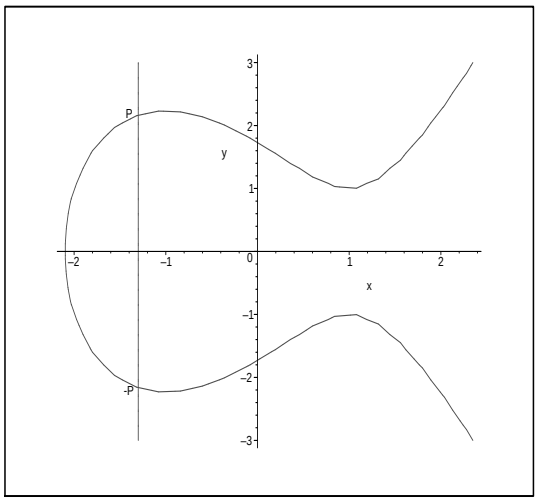

Figura 3: O simétrico do ponto P.

números reais, com $P \neq Q$, considere a reta determinada pelo segmento $\overline{PQ}$. Esta reta interceptará a curva em um ponto $-R$. O simétrico de $-R$, que é

dado por R, será a soma de P e Q. Logo

$$R = P + Q.$$

A figura 4 representa graficamente a soma entre dois pontos distintos $P, Q \in \Omega$ em uma curva elíptica sobre o corpo $\mathbb{R}$. Em criptografia trabalhamos com curvas elípticas sobre corpos finitos, neste caso $\mathbb{F}_p$ e $\mathbb{F}_{2^m}$, em outras palavras, trabalhamos com pontos discretos. O gráfico abaixo é apenas um apelo geométrico para a operação de soma entre os pontos formados por uma curva elíptica. Se quisermos dobrar um ponto $P \in \Omega$ o procedimento

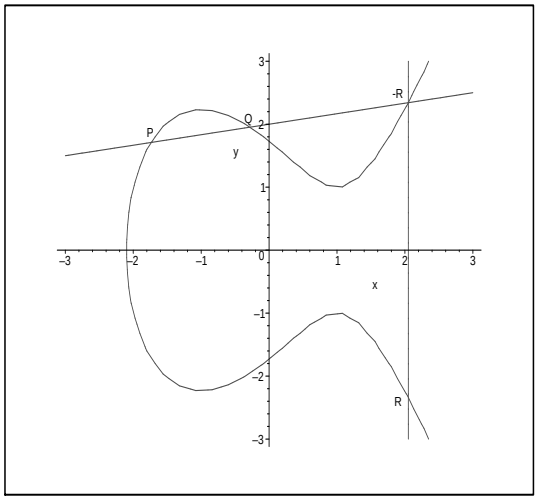

Figura 4: $P + Q = R$.

segue da seguinte maneira: passamos uma reta tangente ao ponto P; esta reta interceptará a curva Ω em outro ponto, que denotaremos por $-R$. O simétrico de $-R$ será a soma de P e P, ou seja

$$R = P + P = 2P.$$

Este fato está representado na figura 5. Um caso específico é configurado se tivermos P um ponto da forma $P = (x, 0)$. Neste caso a reta tangente à curva no ponto P será vertical, contudo, não interceptará a curva em um outro ponto. Neste caso temos que

$$P + P = 2P = \infty.$$

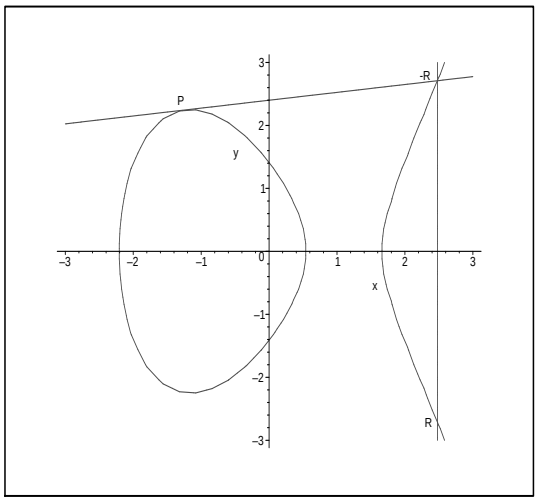

Figura 5: $P + P = R$.

3.1 Curvas Elípticas Sobre $\mathbb{Z}_p$

Seja o corpo $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$, sendo p um primo qualquer. Todas as operações admitidas a seguir devem ser calculadas em função do resto da divisão por p, ou seja, todos os parâmetros e variáveis pertencem ao conjunto $\mathbb{Z}_p$ ou $\mathbb{F}_p$. Uma curva elíptica definida sobre o corpo finito primo $\mathbb{F}_p$ tem equação do tipo (2). Todos os pares $(x, y) \in \mathbb{F}_p$ que satisfazem a equação (2) pertencem à curva elíptica. Como vimos anteriormente, o ponto no infinito ∞ é essencial para que os pontos de uma curva elíptica formem um grupo. Também devemos garantir que esta curva não possua pontos repetidos, para tanto devemos ter $\Delta \neq 0$. Iremos definir uma fórmula para soma entre dois pontos pertencentes a uma curva elíptica [14], que será muito útil mais a frente. Se quisermos somar $P = (x_1, y_1)$ com $Q = (x_2, y_2)$ e seja $R = (x_3, y_3)$ o resultado desta soma, e se $P \neq -Q$, temos

$$x_3 = \lambda^2 - x_1 - x_2 \mod p$$

$$y_3 = \lambda(x_1 - x_3) - y_1 \mod p$$

onde

$$\lambda = \begin{cases} \frac{(y_2 - y_1)}{(x_2 - x_1)} & \mod p, \text{se } x_1 \neq x_2 \\ \frac{(3x_1^2 + a)}{(2y_1)} & \mod p, \text{se } x_1 = x_2 \text{ e } y_1 \neq 0. \end{cases} \tag{3}$$

Se quiséssemos multiplicar um ponto P de uma curva elíptica Ω por um número inteiro n, o procedimento seria somá-lo n vezes. Para ilustrar esta situação, seja $P \in \Omega$, queremos obter $7P \in \Omega$. Bastaria computar:

$$7P = P + 2(P + 2P).$$

A soma do número total de pontos que satisfaz uma curva elíptica Ω mais o ponto no infinito é denominada ordem, que denotaremos por $\#\Omega$. Um resultado importante que diz respeito à ordem de uma curva elíptica se deve a Hasse:

Teorema 3.1.1 (Hasse) *Se Ω uma curva elíptica sobre $\mathbb{F}_q$, temos*

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

Existe um algoritmo de R. Scoof para obter $\#\Omega$ (ver [31] e [30]) que possui tempo de execução $O((\log p)^8)$.

Exemplo 3.1.2 *Considere a curva elíptica sobre $\mathbb{F}_{23}$ com a equação $\Omega : y^2 = x^3 + x$, repare que o ponto $(9, 5)$ satisfaz a equação.*

$$y^2 \mod p = x^3 + x \mod p$$

$$5^2 \mod 23 = 9^3 + 9 \mod 23$$

$$25 \mod 23 = (729 + 9) \mod 23$$

$$25 \mod 23 = 738 \mod 23$$

$2 = 2$. *Logo* $(9, 5) \in E(\mathbb{F}_{23})$.

Os outros pontos que pertencem à curva acima são:

$$(0,0)(1,5)(1,18)(9,18)(11,10)(11,13)(13,5)(13,18)$$
$$(15,3)(15,20)(16,8)(16,15)(17,10)(17,13)(18,10)$$
$$(18,13)(19,1)(19,22)(20,4)(20,19)(21,6)(21,17)$$

mais o ponto ∞. Logo a ordem desta curva é $\#\Omega = 24$.

A figura 6 abaixo mostra estes pontos no plano coordenado. É importante destacar a simetria entre a reta $y = \frac{23}{2}$.

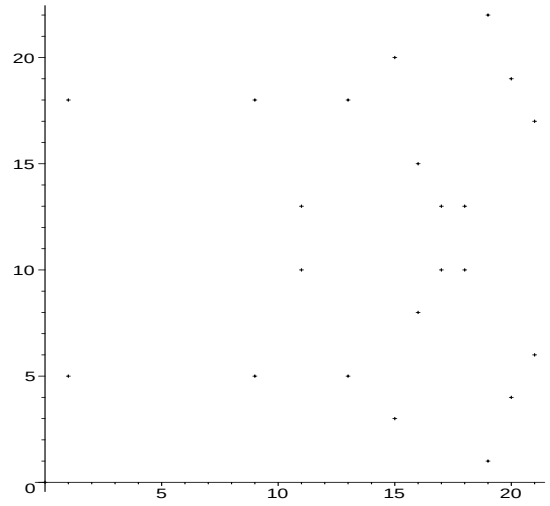

Figura 6: Os pontos da curva $\Omega : y^2 = x^3 + x$ sobre $\mathbb{F}_{23}$.

3.2 Curvas Elípticas Sobre $\mathbb{F}_{2^m}$

A partir de agora iremos usar $\mathbb{F}_{2^m}$ ao invés de $GF(2^m)$. Nesta seção iremos definir, brevemente, curvas elípticas sobre o corpo de Galois $\mathbb{F}_{2^m}$ (ou $GF(2^m)$). A equação padrão de uma curva elíptica sobre $\mathbb{F}_{2^m}$ é ligeiramente diferente das curvas sobre $\mathbb{F}_p$:

$$\Omega : y^2 + xy = x^3 + ax^2 + b \text{ em } \mathbb{F}_{2^m}.$$

Seja $P = (x, y)$ um ponto de uma curva elíptica qualquer sobre $\mathbb{F}_{2^m}$. O inverso deste ponto é dado por $-P = (x, y + x)$. Observe a diferença: o inverso de $P = (x, y)$, em $\mathbb{F}_p$, é $-P = (x, -y)$. As equações que definem a soma também são um pouco diferentes das equações em $\mathbb{F}_p$. Se quisermos somar $P = (x_1, y_1)$ com $Q = (x_2, y_2)$ e seja $R = (x_3, y_3)$ o resultado desta soma, e se $P \neq -Q$, então

$$x_3 = \begin{cases} \left(\frac{y_2+y_1}{x_2+x_1}\right)^2 + \frac{y_2+y_1}{x_2+x_1} + x_1 + x_2 + a, \text{ se } P \neq Q. \\ x_1^2 + \frac{b}{x_1^2}, \text{ se } P = Q. \end{cases}$$

e

$$y_3 = \begin{cases} \left(\frac{y_2+y_1}{x_2+x_1}\right)(x_1 + x_3) + x_3 + y_1, \text{ se } P \neq Q. \\ x_1^2 + \left(x_1 + \frac{y_1}{x_1}\right) x_3 + x_3, \text{ se } P = Q. \end{cases}$$

4 Segurança

A redução do tamanho da chave é resultado de não existir um algoritmo de tempo sub-exponencial para resolver o PLD sobre curvas elípticas. O algoritmo mais rápido que se conhece possui tempo de execução puramente exponencial $O(\sqrt{n})$ [9], em relação ao número de *bits* da ordem da curva, representada por n (ver [40], [16]), no entanto, para resolver o Problema do Logaritmo Discreto (PLD) no grupo multiplicativo $\mathbb{Z}_p^*$ e o Problema da Fatoração de Inteiros (PFI) (ver [32], [4]), que possuem complexidades equivalentes, existe um algoritmo de tempo sub-exponencial: $O(e^{(c+O(1))(\ln n)^{1/3}(\ln \ln n)^{2/3})})$, sendo c uma constante e $n = pq$. Por estes motivos, um sistema criptográfico baseado em curvas elípticas necessita de uma chave de aproximadamente 160 *bits*, e o RSA, por exemplo, utiliza uma chave de 1024 *bits*, mantendo segurança equivalente em ambos. Na tabela 4 podemos ver o tamanho das chaves em *bits*, com o mesmo grau de segurança. Como era de se esperar, os algo-

Modelo de Criptografia			
Simétrico	ECC	RSA	Razão ECC:RSA
80	163	1024	1:6
128	256	3072	1:12
192	384	7680	1:20
256	512	15360	1:30

Tabela 4: Tamanho das chaves em *bits*.

ritmos simétricos necessitam de chaves muito menores que os assimétricos. O mais relevante na tabela 4 é a diferença considerável entre os algoritmos assimétricos. Nesta seção ressaltaremos a presença de dois tipos de curvas que apresentam certa "fraqueza" contra algoritmos específicos (algoritmos que trabalham com certos aspectos bem definidos para poder resolver o problema em questão), são elas: supersingulares (supersingular elliptic curve) e anômalas (anomalous elliptic curve). Pelo teorema de Hasse, sabemos que $\#\Omega(\mathbb{F}_q) = q + 1 - t$, onde $|t| \leq 2\sqrt{q}$, chamaremos t de traço de Frobenius. Uma curva elíptica é chamada de supersingular se, e somente se, $t \equiv 0 \mod p$. Seja $\Omega(\mathbb{F}_p)$ uma curva elíptica, definimos esta curva como anômala, se a ordem desta curva for p, ou seja, $\#\Omega = p$. Estas duas curvas devem ser evitadas, pois existem algoritmos eficientes contra ambas. Inicialmente,

as curvas supersingulares foram consideradas promissoras para criptografia assimétrica, uma vez que permite uma velocidade maior na encriptação. Quando Ω está definida sobre o corpo $\mathbb{Z}_p$, com $p > 3$, pode-se demonstrar que Ω é supersingular se, e somente se, $t^2 \in \{0, p, 2p, 3p, 4p\}$, onde $\#\Omega = p+1-t$. Para o cálculo da ordem da curva, $\#\Omega$, existe um algoritmo bem eficiente que é devido a R. Schoof [31]; este algoritmo possui complexidade $O(\log^8 n)$. Em 1991, Menezes, Okamoto e Vanstone (MOV) [19] descobriram um algoritmo em tempo sub-exponencial para o cálculo de logaritmos discretos em curvas elípticas supersingulares. Este novo algoritmo possui complexidade equivalente ao PLD sobre grupos multiplicativos $\mathbb{Z}_p^*$, porém, como a chave criptográfica de algoritmos baseados em curvas elípticas é substancialmente reduzida, este algoritmo se torna bem eficiente. Na verdade, MOV usaram o emparelhamento Weil para reduzir o PLD sobre curvas elípticas para o PLD sobre grupos multiplicativos de corpos finitos $\mathbb{F}_q$ onde $q = p^m$. A eficiência do algoritmo de MOV está no fato que para o PLD sobre $\mathbb{F}_q$ existem algoritmos de tempo sub-exponencial. Um exemplo de curvas elípticas supersingulares sobre corpos da forma $\mathbb{F}_{2^m}$ é $y^2 + y = x^3 + ax + b$. Em 1991 A. Miyaji propôs o uso de curvas anômalas em criptografia, devido a algumas propriedades. Em 1999, Smart [36] descreveu métodos de ataques polinomiais em curvas elípticas anômalas (traço de Frobenius $t = 1$), dentre outros conceitos, este ataque usa o corpo dos números p-ádicos. Na verdade Smart mostrou que o isomorfismo de grupos

$$\psi : \Omega(\mathbb{F}_p) \to \mathbb{F}_p^+$$

pode ser eficazmente computado, sendo $\Omega(\mathbb{F}_p)$ uma curva anômala; logo calcular k tal que $P = kQ$ é equivalente a calcular k tal que $\psi(P) = k\psi(Q)$ onde $P, Q \in \Omega(\mathbb{F}_p)$. E conseqüentemente reduzindo o PLD em $\Omega(\mathbb{F}_p)$ para o PLD sobre grupos aditivos $\mathbb{F}_p^+$. Dados $a, b \in \mathbb{F}_p^+$ com $a \neq 0$ e p primo, o PLD em $\mathbb{F}_p^+$ (grupo aditivo dos restos módulo p) consiste em encontrar $l \in \{0, \ldots, p-1\}$ tal que $la \equiv b \pmod{p}$. Isto implica que $l \equiv ba^{-1} \pmod{p}$. Observe que isto pode ser facilmente resolvido através do algoritmo estendido de Euclides para encontrar a^{-1}. Com estas comprovações estas classes de curvas foram descontinuadas em protocolos de seguranças. Estes tipos de ataque não são viáveis nem estendíveis a outros tipos de curvas. É importante ressaltar que o número total de curvas supersingulares e curvas anômalas sobre um determinado corpo finito é demasiadamente menor do que o número total de curvas. No entanto, é de extrema relevância que se leve em consideração as seguintes restrições: Seja $G = (x, y)$ um ponto base

de uma curva elíptica $\Omega(\mathbb{F}_p) : y^2 = x^3 + ax + b$ e n a ordem do subgrupo $\langle G\rangle \subseteq \Omega(\mathbb{F}_p)$, ($\langle G\rangle = \{\alpha G : \alpha \in \mathbb{Z}\}$) considere $h = \frac{\#\Omega}{n}$.

i) $4a^3 + 27b^2 \not\equiv 0 \pmod{p}$.
ii) $\#\Omega \neq p$.
iii) $t \not\equiv 0 \pmod{p}$.
iv) $h \leq 4$.

Sendo t o traço de Frobenius. A condição i) deve ser satisfeita para que o conjunto dos pontos da curva Ω venha a ser um grupo. A condição ii) exclui a presença de curvas anômalas. A restrição iii) é importante para detectarmos curvas elípticas supersingulares e finalmente a condição iv) determina se o ponto base G tem uma ordem segura, ou seja, dado um ponto $Q \in \Omega$, é intratável calcular k tal que $Q = kG$. Nos últimos tempos pesquisadores fizeram algumas contribuições em algoritmos já existentes para o cálculo de logaritmos discretos sobre curvas elípticas, porém, se as restrições para a segurança forem alcançadas não existe algoritmo eficiente para o PLD sobre curvas elípticas. De qualquer forma, o NIST (National Institute of Standards and Technology) recomenda em seu site curvas e parâmetros de aplicação segura (veja apêndice E)

4.1 O PLD Sobre Curvas Elípticas

Antes de entrar no mérito do PLD sobre curvas elípticas, vamos falar de forma mais geral (sobre qualquer grupo): Seja $(G, *)$ um grupo munido com a operação $*$, e dado $\alpha \in G$ um gerador de um subgrupo J de G, ou seja, $J = \{\alpha^i : i \geq 0\}$. Sendo $\beta \in J$, o problema consiste em calcular s tal que $\alpha^s = \beta$ onde $\alpha^s = \alpha * \alpha * \ldots * \alpha$ s vezes [39]. No caso de uma curva elíptica, o conjunto G é formado pelos pontos desta curva mais o ponto no infinito ∞, e a operação binária $*$ é representada pela soma de dois pontos na curva, ou seja, a dificuldade que era na exponenciação agora está na multiplicação de um inteiro por um ponto de uma curva elíptica. Isso significa que o problema consiste em encontrar um inteiro que foi multiplicado por um ponto da curva elíptica, isto é, seja Ω uma curva elíptica, dados os pontos $P, Q \in \Omega$ encontrar s tal que $P = sQ$.

Exemplo 4.1.1 (PLD sobre curvas elípticas) *Considere a curva elíptica sobre* $\mathbb{F}_{11}$ $\Omega : y^2 = x^3 + 3x + 2$*, calcular* k *tal que* $k \cdot (4,1) = (6,7)$*, neste caso iremos apresentar uma tabela (tabela 5) com todos os múltiplos do ponto* $(4,1)$*.*

Logo, o inteiro k *vale* 7*.*

$i \cdot (4,1)$	$\lambda = \frac{y_2 - y_1}{x_2 - x_1}$	$\lambda^2 - x_1 - x_2 = x_3$	$\lambda(x_1 - x_3) - y_1 = y_3$	(x_3, y_3)
$2 \cdot (4,1)$	9	7	5	$(7,5)$
$3 \cdot (4,1)$	$\frac{(5-1)}{(7-4)} = 9$	$(9^2 - 4 - 7) = 3$	$(9 \cdot (4-3) - 1) = 4$	$(3,4)$
$4 \cdot (4,1)$	$\frac{(4-1)}{(3-4)} = 5$	$(5^2 - 4 - 3) = 2$	$(5 \cdot (4-2) - 1) = 4$	$(2,4)$
$5 \cdot (4,1)$	$\frac{(4-1)}{(2-4)} = 8$	$(8^2 - 4 - 2) = 10$	$(8 \cdot (4-10) - 1) = 8$	$(10,8)$
$6 \cdot (4,1)$	$\frac{(8-1)}{(10-4)} = 4$	$(4^2 - 4 - 10) = 6$	$(4 \cdot (4-6) - 1) = 4$	$(6,4)$
$7 \cdot (4,1)$	$\frac{(4-1)}{(6-4)} = 3$	$(3^2 - 4 - 6) = 6$	$(3 \cdot (4-6) - 1) = 7$	$(6,7)$
$8 \cdot (4,1)$	$\frac{(7-1)}{(6-4)} = 7$	$(7^2 - 4 - 6) = 10$	$(7 \cdot (4-10) - 1) = 3$	$(10,3)$
$9 \cdot (4,1)$	$\frac{(3-1)}{(10-4)} = 3$	$(3^2 - 4 - 10) = 2$	$(3 \cdot (4-2) - 1) = 7$	$(2,7)$
$10 \cdot (4,1)$	$\frac{(7-1)}{(2-4)} = 4$	$(4^2 - 4 - 2) = 3$	$(4 \cdot (4-3) - 1) = 7$	$(3,7)$
$11 \cdot (4,1)$	$\frac{(7-1)}{(3-4)} = 8$	$(8^2 - 4 - 3) = 7$	$(8 \cdot (4-7) - 1) = 6$	$(7,6)$
$12 \cdot (4,1)$	$\frac{(6-1)}{(7-4)} = 5$	$(5^2 - 4 - 7) = 4$	$(5 \cdot (4-4) - 1) = 10$	$(4,10)$

Tabela 5: Os pontos de uma curva elíptica gerado por $(4,1)$.

Quando as variáveis são consideravelmente grandes, não existe algoritmo eficiente para resolver este problema.

4.2 Comparação com RSA e PLD convencional

A tabela 4 mostra os tamanhos das chaves para que os algoritmos simétricos e assimétricos sejam quebrados usando o mesmo esforço computacional. A eficiência computacional é comparada na tabela 6 [41]. Os tempos são medidos em milissegundos, o tamanho dos corpos finitos nos sistemas de curvas elípticas é aproximadamente 191 bits, e as chaves RSA tem tamanho 1024 bits. Observamos que essa comparação não é inteiramente adequada, pois a

segurança estimada de curvas sobre corpos de 191 bits é consideravelmente maior que a do algoritmo RSA com chaves de 1024 bits. Se o mesmo nível de segurança fosse estabelecido, curvas sobre corpos de aproximadamente 160 bits deveriam ser usadas, aumentando a velocidade em cerca de 50% [5].

operação	$\mathbb{Z}_p$	$GF(2^m)$	RSA
geração de chaves	5,5	11,7	$\approx 10^3$
assinatura	6,3	11,3	43,3
multiplicação por escalar	21,1	56	—

Tabela 6: Comparação entre RSA e métodos de curvas elípticas.

5 Conceitos Sobre Criptografia

A criptografia (kriptós = escondido, oculto; grápho = grafia) pode ser entendida como um conjunto de métodos e técnicas para cifrar ou codificar informações legíveis por meio de um algoritmo, convertendo uma mensagem original em um texto ilegível (criptograma), sendo viável, mediante processo inverso, recuperar as informações originais [35]. A criptografia é mais antiga do que se imagina. Para se ter uma idéia, ela já estava presente nos sistemas de escrita hieroglífica dos egípcios. Os romanos, antes de Cristo, utilizavam códigos sigilosos para comunicação em tempos de guerra. Em 50 a.C, Júlio César usou sua cifra de substituição para criptografar comunicações governamentais. Para cifrar uma mensagem, César deslocava as letras em três posições no alfabeto, por exemplo, a letra D, se transformava em G. Apesar de muito simples, este tipo de criptografia foi utilizada pelos oficiais sulistas na Guerra de Secessão americana e também pelo exército russo em 1915. Algumas vezes, Júlio César aumentava a segurança do seu método substituindo letras latinas por letras gregas. Com o advento do computador, a criptografia cresceu significantemente incorporando complexos algoritmos matemáticos. Entre 1933 e 1945, a máquina Enigma, que havia sido criada por Arthur Scherbius, foi aperfeiçoada até se transformar na ferramenta criptográfica mais importante da Alemanha nazista. Este sistema foi totalmente quebrado pelo matemático polonês Marian Rejewski. Para tanto, Rejewski, se baseou apenas em textos ilegíveis que foram interceptados e em uma lista de chaves que foram obtidas por intermédio de um espião [12]. Outro grande

marco na história da criptografia foi o advento da criptografia assimétrica, em 1976 que possibilitou a comunicação segura em um canal de comunicação inseguro, iremos falar mais deste ocorrido em seções posteriores.

5.1 Criptografia Simétrica

Na criptografia simétrica, a chave que encripta o texto é a mesma que o decifra. Logo, tanto o remetente como o destinatário possuem uma mesma chave secreta compartilhada. Para quem não conhece a chave, deve ser computacionalmente inviável, obter a mensagem a partir do criptograma (mensagem criptografada). A figura 7 ilustra o processo de criptografia simétrica.

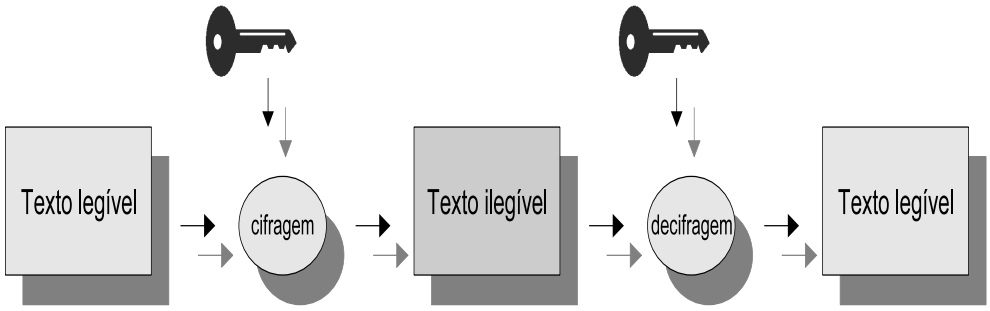

Figura 7: Funcionamento de um algoritmo simétrico.

Nos algoritmos simétricos (de chave privada), ocorre o chamado "problema de distribuição de chaves". Essa talvez seja a grande desvantagem de um algoritmo simétrico. Por exemplo, para uma comunicação segura entre k pessoas, neste caso, serão exigidas $\frac{k(k-1)}{2}$ chaves diferentes. Assim, estas chaves devem ser enviadas para todos os usuários autorizados. Isto resulta em um atraso de tempo e pode possibilitar que a chave não chegue a pessoa autorizada. Os algoritmos simétricos se dividem em: Algoritmos de bloco ou de fluxo. As cifras de bloco trabalham sobre blocos de dados. Antes de encriptar uma mensagem, ela é subdividida em blocos que normalmente são de 8 ou 16 bytes. O último bloco é completado, caso seja necessário, de acordo com um protocolo estabelecido (conhecido como *padding*). De acordo com Shannon [33], uma cifra de bloco precisar promover: difusão (cada bit da chave deve afetar o máximo de bits do criptograma) e confusão (cada bit do criptograma deve ser de difícil associação com bits da mensagem original). As cifras de fluxo criptografam a mensagem "bit a bit", em um fluxo contínuo,

diferentemente do que acontece com a cifra de bloco. Uma grande vantagem dos criptossitemas simétricos é o desempenho computacional. Estes criptossitemas chegam a ser milhares de vezes mais rápidos que os criptossitemas de chave pública (assimétricos). Na prática, é utilizada uma combinação de ambos os algoritmos. Atualmente, um criptossitema simétrico que se destaca é o AES (*Advanced Encryption Standard*), também conhecido como Rijndael. O AES foi anunciado como vencedor pelo NIST, em novembro de 2001, após um longo concurso para se tornar o padrão de criptografia simétrica do governo norte-americano. O AES substituiu o DES (Data Encryption Standard), que havia sido quebrado pela máquina *DES Cracker* com apenas 250 mil dólares. Hoje o AES tem sua segurança aceitável com uma chave de 80 bits.

5.2 Criptografia Assimétrica

Em 1976, Whitfield Diffie e Martin E. Hellman publicaram um artigo [7] do qual foi proposto uma interessante solução para o estabelecimento de uma chave secreta compartilhada em um canal de comunicação inseguro. Neste modelo, denominado Modelo de Chave Pública, cada usuário possui um par de chaves (S, P) sendo S sua chave secreta e P denota sua chave pública. Estas chaves possuem uma relação matemática de forma que [39]:

- Se m denota uma mensagem, e $S()$ é tal que transforma m em $S(m) = c$ onde c denota o texto criptografado, então $P(c) = m$. Em outras palavras, S é a chave inversa de S. De forma que $P(S(m)) = m$.
- É computacionalmente inviável obter S a partir de P.
- O cálculo do par de chaves é computacionalmente viável.
- Os cálculos de $S(m)$ e $P(c)$ são computacionalmente fáceis.

A figura 8 mostra um esquema de criptografia assimétrica (Modelo de Chave Pública).

A solução proposta por Diffie e Hellman servia para que dois usuários combinassem uma chave secreta compartilhada somente entre ambos. Após ter a chave combinada, os usuários poderiam trocar informações utilizando um algoritmo simétrico, uma vez que os algoritmos simétricos são significantemente mais rápidos que os assimétricos.

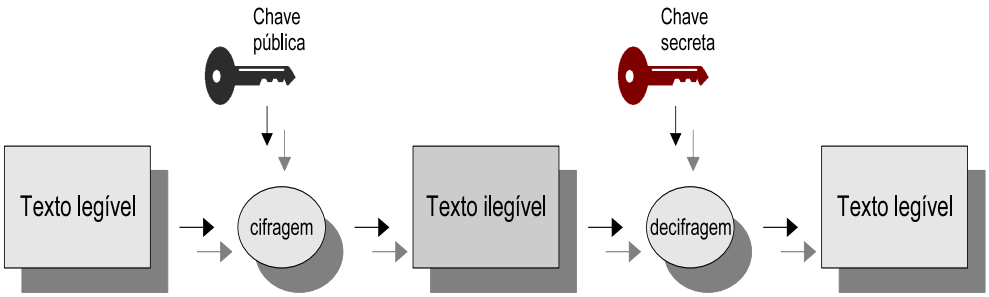

Figura 8: Esquema de um algoritmo assimétrico.

5.2.1 Diffie-Hellman

Nesta seção falaremos do primeiro algoritmo assimétrico (de chave pública) a ser utilizado: O Algoritmo Diffie-Hellman. A seguir mostraremos os passos para que Alice e Bob estabeleçam uma chave criptográfica compartilhada. Neste caso ambos tem conhecimento do primo p e do inteiro $1 < g < p$.

1. Alice escolhe um inteiro s_a como a sua chave privada e calcula $K_a = g^{s_a} \mod p$, envia para Bob K_a.
2. Analogamente Bob escolhe seu inteiro s_b e envia para Alice $K_b = g^{s_b} \mod p$.
3. Alice calcula $K_b^{s_a} \mod p$.
4. Bob calcula $K_a^{s_b} \mod p$.

Observe que $K_b^{s_a} = (g^{s_b})^{s_a} = g^{s_a s_b} = (g^{s_a})^{s_b} = K_a^{s_b} \mod p$. Após a execução deste protocolos, Alice e Bob poderiam utilizar um criptossitema simétrico para trocar informações entre eles. A segurança deste algoritmo está baseada no PLD sobre $\mathbb{Z}_p$. Este protocolo pode apresentar algumas falhas de segurança. Na próxima seção descreveremos uma destas falhas.

5.2.2 Ataque Ativo Man-in-the-Middle

Neste ataque consideraremos um intruso que não apenas lê as informações que trafegam sob um canal, mas como também as altera e injeta. Este caso caracteriza um ataque ativo do tipo *man-in-the-middle* (homem no meio). Em relação ao protocolo descrito na seção anterior, o intruso poderá se passar

tanto por Alice perante Bob quanto por Bob perante Alice. A figura 9 mostra este aspecto

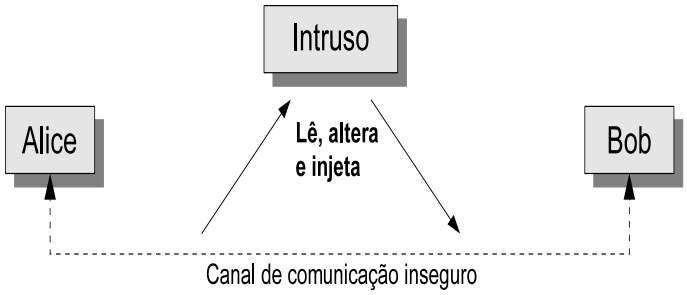

Figura 9: Ataque *man-in-the-middle* ativo.

Para ludibriar Alice e Bob o intruso procede da seguinte maneira:

1. No momento em que Alice envia K_a para Bob, o intruso intercepta e envia outro inteiro, digamos $K_i = g^{s_i} \mod p$. Lembre-se que os inteiros p, g são públicos.

2. Bob recebe K_i pensando que, na verdade, é K_a e faz os cálculos de acordo com o algoritmo. Assim Bob estabelece uma chave com o intruso.

3. Quando Bob envia K_b para Alice, o intruso procede da mesma maneira que no passo 2. E firma uma chave com Alice.

Assim o intruso tem uma chave para se comunicar com Bob e outra para se comunicar com Alice através, por exemplo, de um criptossistema simétrico. Para resolver este problema poderíamos modificar o algoritmo de maneira que cada usuário utilize como chave pública a tripla (p, g, R) onde $R = g^s \mod p$. Vamos supor que Alice queira enviar w para Bob. Aqui Bob publica previamente sua chave pública (p_b, g_b, R_b). Logo ficamos com a seguinte modificação:

1. Alice escolhe seu inteiro secreto s_a e calcula $g_b^{s_a} \mod p_b = X_a$ e $K_{ab} = R_b^{s_a} \mod p$. Agora Alice usa K_{ab} para criptografar w: $K_{ab}(w) = u$. Envia u e X_a para Bob.

2. Bob calcula $X_a^{s_b} \mod p = K_{ab}$ e utiliza esse valor para decriptar u e assim obtendo w

Observe que não existe mais o 'dialogo', neste caso só há um envio de dados.

5.2.3 RSA

O RSA (Ron **R**ivest, Adi **S**hamir e Len **A**dleman) foi publicado em 1978 [29] e é atualmente o principal criptossistema de chave pública usado em aplicações comerciais [6]. O Algoritmo RSA tem sua segurança baseada no problema da fatoração de inteiros. Em outras palavras, não existe um algoritmo eficaz, para inteiros de tamanho adequado, para fatoração. Hoje, o RSA tem sua segurança aceitável com 1024 bits (alguns bancos utilizam 2048). Inicialmente veremos como funciona o processo de geração de chaves (pública/privada).

Algoritmo 2: Geração de chaves RSA.

Saída: A chave pública (n, e) e a chave privada (n, d)

início
  Escolher dois primos grandes, digamos p e q;
  Calcular $n = p \cdot q$;
  Calcular a função $\varphi(n) = (p-1)(q-1)$;
  Escolher um inteiro e tal que $1 < e < \varphi(n)$ e $mdc(e, \varphi(n)) = 1$;
  Calcular d, tal que $d \cdot e \equiv 1 \pmod{\varphi(n)}$;
  retorna chave pública: (n, e), chave privada: (n, d);
fim

Para geração de primos grandes olhe o apêndice A. Para criptografar uma mensagem procedemos da seguinte forma:

Cifragem:

Para criptografar uma mensagem $m < n$, sendo c o criptograma fazemos:

$$m^e = c \mod n.$$

Decifragem:

Para decriptar c procedemos da seguinte forma:

$$c^d = (m^e)^d = m \mod n.$$

Para o cálculo da exponenciação modular veja o apêndice B. Agora iremos demonstrar o processo de descriptação, ou seja iremos demonstrar o fato que $(m^e)^d = m \mod n$.

Demonstração: Como e é o inverso de d módulo $\varphi(n)$ temos que $d \cdot e = 1 + \varphi(n)k$, para algum $k \in \mathbb{N}$. Assim temos que $m^{de} = m^{1+\varphi(n)k} = (m^{\varphi(m)})^k m \pmod n$. Como $\varphi(n) = (p-1)(q-1)$ pelo *Pequeno Teorema de Fermat* temos que:

$$(m^{(p-1)})^{(q-1)} m \equiv m \pmod p$$

se $p \not| m$, (se $p|m$ então $m^{de} \equiv 0 \pmod p$). Analogamente podemos demonstrar que $m^{de} \equiv m \pmod q$. Como p e q são primos e $m^{de} \equiv m \pmod p$ e $m^{de} \equiv m \pmod q$ então $m^{de} \equiv m \pmod n$. Como $m < n$ a afirmação foi provada. Para consolidar o método desta seção iremos fazer um exemplo numérico.

Exemplo 5.2.1 *Iremos criptografar a sigla LNCC, para isso, inicialmente, recorremos à tabela ASCII para codificar as letras em um único inteiro. Depois de codificada e concatenada, temos que a sigla fica* $LNCC = 76786767$. *O próximo passo é a geração de chaves:*

Geração de Chaves:

1. *Escolher os primos, neste caso* $p = 4294967311$ *e* $q = 8616319249$.
2. *Calcular* $n = 37006809515595069439$.
3. *Calculat* $\varphi(37006809515595069439) = 37006809502683782880$.
4. *Escolhemos aleatoriamente* $e = 22987520831085250907$, *neste caso devemos ter* $mdc(e, n) = 1$.
5. *Pelo algoritmo de Euclides Estendido temos que* $d = 16954631775963686003$.

Nossa chave pública é o par

$$(n, e) = (37006809515595069439, 22987520831085250907)$$

e nossa chave privada fica

$$(n, d) = (37006809515595069439, 16954631775963686003).$$

Para criptografar a mensagem LNCC é necessário o conhecimento da chave pública do destinatário, que neste caso é o par (n, e)*. O próximo passo é encriptar a mensagem* $LNCC = 76786767$*, então*

$$76786767^{22987520831085250907} = 34891697997283948788 \mod 37006809515595069439.$$

Neste exemplo o criptograma é o inteiro $c = 34891697997283948788$*. Para decriptar* c *usamos nossa chave privada* (n, d) *de forma que:*

$$34891697997283948788^{16954631775963686003} = 76786767 \mod 37006809515595069439.$$

Recuperando assim a mensagem $LNCC = 76786767$*.*

5.3 Funções de Hash

Antes de falarmos sobre métodos assinatura digital iremos introduzir o conceito de funções de *hash*. Basicamente funções de *hash* de uma via só convertem uma *string* (seqüência de caracteres) de tamanho indefinido para outra de tamanho fixo (de maneira geral a *string* convertida é muito menor que a *string* de entrada). Estas funções são usadas em sistemas operacionais para verificação de *login*/senha, autenticidade através de algoritmos MAC (*Message Authentication Code*) e assinaturas digitais, integridade de arquivos entre outras aplicações. Qualquer algoritmo de função de *hash* leva um texto puro à um texto naturalmente sem nexo. O diferencial dessas funções é exatamente a falta de uma função inversa, que permite um grau mais elevado de segurança. Logo não existe a mínima possibilidade de recuperar o

texto original, se existisse tal função inversa, a função de *hash* seria uma excelente compactadora de dados, contradizendo o teorema da compressão de Shannon [33]. Contudo uma função de *hash* de uma via só destrói a mensagem de entrada, ou seja, para uma única mensagem autenticada por uma função *hash* de uma via só existem várias (infinitas) mensagens de entrada que levam a esta mensagem autenticada. É importante observar que as funções de *hash* não possuem chaves, fazendo uma ingênua analogia, cada pessoa possui uma impressão digital que a distingue das outras pessoas, no caso de uma função de *hash*, esta identifica exclusivamente uma mensagem de tamanho arbitrário das outras mensagens. Normalmente os valores *hash* são relativamente pequenos, entre 128 a 256 bits. Isto vem fortalecer a causa de que haja um grande número de colisões, ou seja, mensagens originais diferentes gerando o mesmo *hash*. É importante ressaltar que as funções de *hash*

de uma via só são projetadas de forma que seja praticamente impossível criar uma mensagem que resulte em um *hash* definido anteriormente, ou criar duas mensagens diferentes que resultem um *hash* de um mesmo valor. Esta é a grande diferencial de uma boa função de *hash*. Outra possibilidade é tentar um ataque de força bruta contra a função de *hash*, isso quer dizer, poderíamos gerar vários *hash* de diversas mensagens, uma após a outra, procurando por uma mensagem que resultasse em um valor de *hash* particular, ou duas mensagens que resultem em um mesmo valor de *hash*. Observe que a taxa de sucesso deste tipo de ataque depende diretamente do tamanho do valor de *hash*. Em suma, *hash* é uma função h que satisfaz as seguintes condições [37]:

1. A entrada x pode ter um comprimento arbitrário e o resultado $h(x)$ possui um número fixo de n bits.

2. Dados h e uma entrada x, é relativamente "fácil" calcular $h(x)$.

3. A função precisa ser unidirecional, ou seja, para um dado y na imagem de h, seja "difícil" achar uma mensagem x de modo que $h(x) = y$ e, dados x e $h(x)$, seja "difícil" encontrar uma mensagem x' diferente de x onde $h(x') = h(x)$.

Se quisermos analisar melhor a probabilidade de colisões vem a seguinte pergunta. Dada uma função com k valores possíveis, qual é a probabilidade de: para n entradas diferentes, pelo menos duas tenham o mesmo valor? Basicamente, o número de pares de entradas diferentes é dado por

$$\binom{n}{2} = \frac{n(n-1)}{2}.$$

Observe que a probabilidade dos 2 elementos de um par terem o mesmo valor é $\frac{1}{k}$ logo a probabilidade dos 2 elementos não terem o mesmo valor é

$$1 - \frac{1}{k}$$

a probabilidade de nenhum dos pares ter o mesmo valor é dado por

$$(1 - \frac{1}{k})^{\frac{n(n-1)}{2}}$$

logo a probabilidade de algum dos pares ter a mesma saída é

$$1 - (1 - \frac{1}{k})^{\frac{n(n-1)}{2}}.$$

O objetivo da função de *hash* na autenticação é gerar uma "impressão digital" de uma mensagem ou informação [37]. Uma outra aplicação é em autenticação de sistemas, que consiste em: verificar se um dado usuário tem autorização para acessar os dados que ele deseja. O usuário pode inserir como entrada para a tabela de *hash* uma senha e um *login* e a função de *hash* fica encarregada de verificar a veracidade da mesma. O valor gerado pela função é comparado com o valor armazenado na tabela de *hash*. Caso a comparação seja válida, o usuário fica autorizado ao acesso. Uma propriedade importante que uma função de *hash* deve satisfazer é: Para qualquer par (x, y) deve ser impossível encontrar $h(x) = h(y)$. Essa propriedade é conhecida também como resistência forte a colisões. A partir do momento em que duas chaves diferentes com a mesma função são encontradas, toda a segurança na autenticação é quebrada. Ou seja, no caso da autenticação, a possibilidade de colisões deve se reduzida ao máximo. Baseado nessas características, vemos que toda a segurança de uma sistema de autenticação baseado em *hash* tem como fundamento principal a escolha de uma boa função determinística [27]. Na verdade, a função determinística ideal seria aquela cujo o valor gerado pela função $h(x)$ tenha a mesma probabilidade de ser gerado em relação aos demais valores possíveis. Além de fornecer autenticação, um *hash* também fornece pode informações sobre integridade de dados.

5.3.1 SHA

O SHA (*Secure Hash Algorithm*) [26] é uma função de *hash* de uma via só invetanda no NSA (*National Security Agency*) e foi desenvolvido pelo NIST (*National Institute of Standards and Technology*), e publicada com um FIPS (*Federal Information Processing Standard*) em 1993. Mais precisamente o SHA ficou conhecido como o FIPS PUB 180. Em 1995, esta publicação foi revisada e ganhou a identificação de FIPS PUB 180–1 a qual é conhecida genericamente como SHA-1. Ele produz um valor de *hash* de 160 *bit* a partir de um tamanho arbitrário da mensagem. Os funcionamentos internos do SHA são bem semelhantes com os do MD4. Com exceção do ataque de força bruta, não se é conhecida uma forma de ataque criptoanalítica contra o SHA.

5.3.2 MD5

Foi desenvolvido em 1991 por Ronald Rivest para suceder ao MD4 que tinha alguns problemas de segurança. O MD5 (*Message–Digest algorithm 5*) [28] é

um algoritmo de *hash* de 128 bits unidirecional desenvolvido pela RSA Data Security, Inc., é principalmente usado verificação de integridade e *logins*. O método de verificação é, feito pela comparação das duas hash (uma da base de dados, e a outra da tentativa de *login*). Hoje o MD5 é bastante usado para verificar a integridade de um arquivo através, por exemplo, do programa *md5sum*, que cria a *hash* de um arquivo. Hoje, não é aconselhável a utilização do MD5 para aplicações onde são exigidos grandes níveis de segurança, pois existem ataques conhecidos e um grande número de colisões [18, 38].

5.4 Assinatura Digital

Assinatura digital é o método de autenticação de uma mensagem do emissor que utiliza criptografia assimétrica em conjunto com uma função de *hash*. Os principais objetivos da assinatura digital são:

- Integridade: a mensagem não foi modificada na transmissão.
- Autenticidade: o destinatário pode confiar que foi realmente o emissor que enviou a mensagem.

Na verdade, existem diversas maneiras de se utilizar um método de assinatura digital. De modo geral, é gerado um *hash* da mensagem que será transmitida e posteriormente esse valor é encriptado com a chave privada do emissor. O processo simples de assinatura digital é ilustrado na figura 10

Inicialmente gera um *hash* do texto original como o objetivo de não precisar cifrar a mensagem inteira, o que seria desnecessário, uma vez que o valor do *hash* pretende gerar um inteiro que identifica unicamente o texto original. O emissor encripta o *hash* gerado com sua chave privada, com intuito de garantir que a única chave que decripta o valor do *hash* é a sua chave pública (de conhecimento público). Esse valor do *hash* cifrado será a assinatura digital do texto original. Assim, qualquer receptor, poderá garantir que foi realmente o emissor que transmitiu o texto. A figura 11 mostra como o destinatário deve proceder quando recebe uma mensagem assinada digitalmente.

Ao receber o texto original e a assinatura, o destinatário calcula o valor do *hash* do texto original. Em paralelo, ele pode decriptar a assinatura recebida utilizando a chave pública do emissor. Agora ele compara o valor do *hash* com o valor que acabara de decifrar. Caso sejam iguais o receptor tem a certeza que o arquivo provém do verdadeiro emissor. Em outras palavras, como

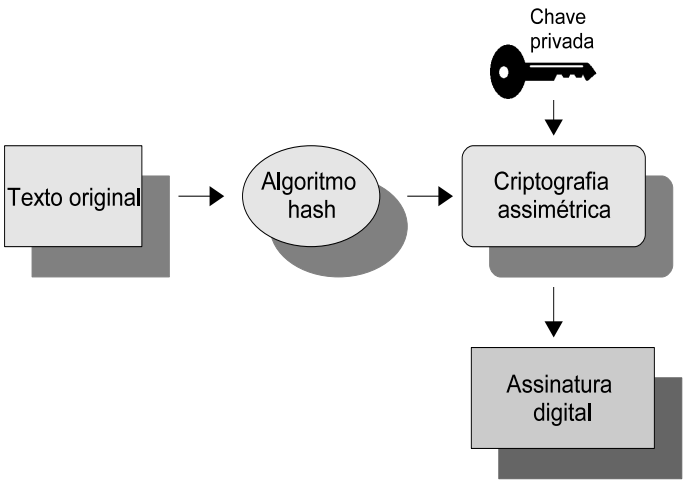

Figura 10: Esquema de assinatura digital.

ele conseguiu decifrar o *hash* com a chave pública do emissor quer dizer que a única chave que poderia encriptar tal valor seria a chave privada do emissor, visto que essa chave só é de conhecimento único e exclusivo do emissor. Esse processo também garante que o texto original não foi modificado na transmissão. Estes fatos estão relacionados aos termos já citados: autenticidade e integridade, respectivamente. É importante ressaltar que os algoritmos de assinatura digital não garantem a privacidades dos dados transmitidos. Uma atratividade particular do algoritmo RSA é a maneira natural de como ele é utilizado para assinar mensagens [15]. As figuras 10 e 11 mostram exatamente como poderíamos utilizar o RSA para assinar mensagens. No entanto, não é possível assinar mensagens utilizando o algoritmo Diffie-Hellman.

5.4.1 DSA

O DSA (Digital Signature Algorithm) é um algoritmo proposto pelo NIST em agosto de 1991. O DSA tornou-se U.S. Federal Information Processing Standard 186 (FIPS 186) [2] especificado pelo padrão de assinatura DSS (Digital Signature Standard). O DSS requer explicitamente o uso do algoritmo de *hash* SHA-1. Este algoritmo, diferente do RSA, tem sua segurança baseada no PLD. O algoritmo 3 mostra como funciona a geração de chaves para o algoritmo DSA [21].

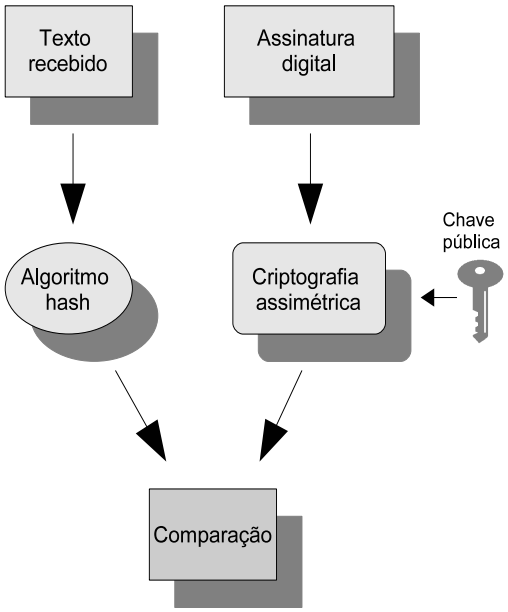

Figura 11: Comparação da assinatura.

Para a assinar uma mensagem m utilizamos o algoritmo 4:

Onde $H(m)$ representa o valor de 160 bits do *hash* gerado a partir do *Secure Hash Algorithm* SHA-1. O par (r, s) totalizam 320 bits. O algoritmo 5 trata da verificação da assinatura digital por parte do receptor.

6 Aplicação de Curvas Elípticas em Criptografia

Nesta seção iremos expor dois métodos usados para criptografar mensagens usando curvas elípticas. Ilustraremos com exemplos numéricos tais algoritmos, tendo em vista facilitar o entendimento. Iremos trabalhar com ambos os corpos $\mathbb{F}_p$ e $\mathbb{F}_{2^m}$ para descrever o algoritmo Diffie-Hellman e ElGamal ambos sobre curvas elípticas, no entanto o outro algoritmo (Menezes-Vanstone) que mostraremos posteriormente é aplicado apenas em corpos primos $\mathbb{F}_p$.

Algoritmo 3: Geração de chaves do DSA.

Saída: Chave pública e privada

início

  Escolha um primo q onde $2^{159} < q < 2^{160}$;

  Selecione um inteiro t, $0 \leq t \leq 8$ e escolha um primo p onde $2^{511+64t} < p < 2^{512+64t}$ tal que $q|(p-1)$;

  Escolha um inteiro $g \in \mathbb{Z}_p^*$ e calcule $\alpha \leftarrow g^{\frac{(p-1)}{q}} \mod p$;

  se $\alpha = 1$ **então**

    volte para a linha 5;

  Selecione um inteiro aleatório a, tal que $1 \leq a \leq (q-1)$;

  Calcule $y = \alpha^a \mod p$;

  retorna chave pública (p, q, α, y) e a chave privada a;

fim

Algoritmo 4: Geração da assinatura DSA.

Entrada: A mensagem m

Saída: A assinatura da mensagem m

início

  Escolha um inteiro secreto aleatório $0 < k < q$;

  Calcule $r = (\alpha^k \mod p) \mod q$;

  Compute $k^{-1} \mod q$;

  Calcule $s = k^{-1}(H(m) + ar) \mod q$;

  retorna (r, s);

fim

Algoritmo 5: Verificação da assinatura DSA.

Entrada: A mensagem m e a assinatura (r, s)

início

  Calcule $w = s^{-1} \mod q$;

  Compute $u_1 = w \cdot H(m) \mod q$ e $u_2 = rw \mod q$;

  Calcule $v = (\alpha^{u_1} y^{u_2} \mod p) \mod q$;

  Aceite a assinatura se, e somente se, $v = r$;

fim

6.1 O Protocolo Diffie-Hellman Sobre Curvas Elípticas

O protocolo Diffie-Hellman de troca de chave poderá ser reorganizado para trabalhar sobre o grupo formado pelos pontos de uma curvas elípticas, já sabemos que tais pontos formam um grupo, portanto podemos usar o PLD bem como o Protocolo Diffie-Hellman. Vimos na seção anterior que no caso de pontos de uma curva elíptica Ω, a dificuldade está em: dado um par de pontos $P, Q \in \Omega$ encontrar um inteiro k tal que$P = kQ$. Sobre o grupo multiplicativo $\mathbb{Z}_p^*$ a operação que usamos é a exponenciação, no entanto, em pontos de uma curva elíptica se usa a multiplicação de um inteiro por um ponto. O Algoritmo abaixo mostra dois usuários, Alice e Bob, estabelecendo uma chave secreta compartilhada, para isso, Alice e Bob conhecem publicamente um ponto $G \in \Omega$. Após a execução deste algoritmo, a chave estabelecida poderá ser usada em um criptossistema simétrico (chave privada).
Algoritmo:

1. Alice escolhe um inteiro s_a como a sua chave privada e calcula $K_a = s_aG$, envia para Bob K.

2. Analogamente Bob escolhe seu inteiro s_b e envia para Alice $K_b = s_bG$.

3. Alice calcula s_aK_b.

4. Bob calcula s_bK_a.

Repare que $s_aK_b = s_as_bG = s_bs_aG = s_bK$. Para facilitar a compreensão vamos a um exemplo numérico usando uma curva do tipo $\Omega(\mathbb{Z}_p)$.

Exemplo 6.1.1 *Neste exemplo usaremos a curva* $\Omega : y^2 = x^3 + 101x + 552$ *sobre o corpo finito primo* $\mathbb{Z}_{4229}$*, sendo 4229 um primo. Alice e Bob conhecem publicamente o ponto* $G = (4197, 231) \in \Omega(\mathbb{Z}_{4229})$.

1. *Alice escolhe sua chave secreta* s_a*, digamos* $s_a = 1011$ *e calcula* $K_a = s_aG = 1011(4197, 231) = (3617, 263)$*, envia para Bob* K_a.

2. *Bob escolhe seu inteiro* s_b*, neste caso* $s_b = 1402$ *e envia para Alice* $K_b = s_bG = 1402(4197, 231) = (392, 2766)$.

3. *Alice calcula* $s_aK_b = 1011(392, 2766) = (3990, 3565)$.

4. *Bob calcula* $s_bK_a = 1402(3617, 263) = (3990, 3565)$.

Ambos possuem o ponto $P = (3990, 3565)$ *que será a chave secreta compartilhada. No próximo exemplo usaremos um corpo do tipo* $\mathbb{F}_{2^m}$*, para deixar bem claro o que o algoritmo Diffie-Hellman pode ser usado em ambos os corpos.*

Exemplo 6.1.2 *Considere a curva elíptica* $\Omega : y^2 + xy = x^3 + g^{22}x^2 + g^{198}$ *(*$a = g^{22}$ *e* $b = g^{198}$*) sobre* $\mathbb{F}_{2^8}$*, sendo* $g = x^7 + x^5 + x^4 + x = (10110010)$ *um polinômio primitivo (gerador) de* $\mathbb{F}_{2^8}$*. Usaremos como polinômio irredutível o polinômio* $f(x) = x^8 + x^4 + x^3 + x + 1$*. Vamos supor que Alice e Bob conheçam publicamente o ponto* $G = (g^{207}, g^{173}) = ((01101011), (01011111))$*. Então*

1. *Alice escolhe sua chave secreta* s_a*,* $s_a = 57$ *e calcula* $K_a = s_aG = 57(g^{207}, g^{173}) = (g^{130}, g^{75})$*, envia para Bob* $K_a = (g^{130}, g^{75}) = ((01110100), (11010011))$*.*

2. *Bob escolhe seu inteiro secreto* $s_b = 31$ *e envia para Alice* $K_b = s_bG = 31(g^{207}, g^{173}) = (g^{198}, g^{90}) = ((00110111), (00110110))$*.*

3. *Alice calcula* $s_aK_b = 57(g^{198}, g^{90}) = (g^{43}, g^{53}) = ((10000111), (01011010))$*.*

4. *Bob calcula* $s_bK_a = 31(g^{130}, g^{75}) = (g^{43}, g^{53}) = ((10000111), (01011010))$*.*

Sendo assim eles estabeleceram uma chave secreta compartilhada. Que neste caso será, na representação binária, $P = ((10000111), (01011010))$*.*

6.2 O Algoritmo ElGamal Sobre Curvas Elípticas

Como o algoritmo ElGamal trabalha com um grupo, podemos usá-lo sobre curvas elípticas. Vamos supor que Alice deseja enviar uma mensagem para Bob, seja m esta mensagem mapeada num ponto de uma curva elíptica, para tanto vamos ao algoritmo [10].

Algoritmo:

1. Bob escolhe, e mantém em segredo, um inteiro $b \in \mathbb{N}^*$ e envia à Alice $K = bP$, sendo P um ponto da curva elíptica conhecido publicamente.

2. Alice escolhe inteiro $a \in \mathbb{N}^*$ (também o guarda para si) e computa $c_1 = aP$ e $c_2 = m + aK$.

3. Para decifrar a mensagem m Bob calcula $c_2 - bc_1 = m + abP - baP = m$.

Para facilitar o entendimento, vamos a um exemplo.

Exemplo 6.2.1 *Iremos criptografar a palavra LNCC, neste caso iremos cifrar em blocos de duas letras, primeiro mapeamos as letras LN e depois CC em pontos de uma curva elíptica. Neste exemplo iremos mapear da seguinte maneira: cada letra do alfabeto estará associada a um número entre* 01 *e* 26 *de forma que A*= 01*, B*= 02*, ..., Z*= 26*, porém, precisamos mapear esta mensagem em pontos de uma curva elíptica. Para isso iremos multiplicar o número associado à letra por um ponto P, logo iremos concatenar os números associados e depois multiplicar por P. Para ilustrar esta situação, vamos supor que iremos mapear AB em uma curva elíptica* $y^2 = x^3 + 373x + 402$ *sobre* $\mathbb{F}_{3697}$*, como A*= 01 *e B*= 02 *concatenamos* 01 *e* 02*, obtemos* 0102 *e multiplicamos por P, digamos* $P = (551, 1946)$*, logo* $102P = 102(551, 1946) = (3108, 1065)$*. Logo, AB*=$(3108, 1065)$*. Vamos supor que Alice enviará a palavra LNCC para Bob, para tanto*

1. *Bob escolhe um número inteiro b, digamos* $b = 919$*, e envia à Alice* $P = (551, 1946)$ *e* $K = bP = 919(551, 1446) = (301, 3454)$ *(observe que é um problema inviável descobrir b a partir de K e P). Observe que K pertence a curva elíptica* $y^2 = x^3 + 373x + 402$ *sobre* $\mathbb{F}_{3697}$ *.*

2. *Alice escolhe um inteiro* $a = 815$*, e multiplica* $c_1 = 815P = 815(551, 1946) = (958, 14)$*. Agora Alice precisa mapear a string LN, procedendo do modo que fizemos acima. A string foi mapeada no ponto* $m = LN = (2309, 2502)$*. Precisamos somar este ponto ao ponto obtido por* $aK = 815K = 815(301, 3454) = (837, 2461)$*. Agora Alice mascara o ponto* $m = LN = (2309, 2502)$*, somando-o ao ponto aK. Logo,* $c_2 = m + aK = (2309, 2502) + (837, 2461) = (1518, 14)$*. Alice transmite a Bob o par de pontos cifrados* $(c_1, c_2) = ((958, 14), (1518, 14))$

3. *Para decriptar Bob deve calcular* $m = c_2 - bc_1$*. Primeiro Bob calcula o valor de* $bc_1 = 919c_1 = 919(958, 14) = (837, 2461)$*. Prosseguindo, temos que* $m = c_2 - bc_1 = (1518, 14) - (837, 2461) = (1518, 14) + (-(837, 2461)) = (1518, 14) + (837, -2461) = (2309, 2502)$*, repare que* $LN = (2309, 2502)$*.*

4. *Para criptografar as letras CC o procedimento é o mesmo, agora Alice só precisa calcular o ponto* $c_2 = m+aK$ *CC*= $0303P = 303(551, 1946) =$

$(3023, 762) = m$, $c_2 = (3023, 762) + (837, 2461) = (3084, 2426)$ *e envia* $c_2 = (3084, 2426)$ *para Bob.*

5. *Bob novamente calcula* $m = c_2 - bc_1 = (3084, 2426) - (837, 2461) = (3084, 2426)+(-(837, 2461)) = (3084, 2426)+(837, -2461) = (3023, 762)$, *recuperando a string CC, e assim encerra o processo.*

Neste trabalho, expomos dois corpos para serem usados sobre curvas elípticas, $\mathbb{F}_p$ e $\mathbb{F}_{2^m}$. No exemplo anterior usamos o corpo $\mathbb{F}_p$, agora usaremos como exemplo um corpo finito do tipo $\mathbb{F}_{2^m}$, usando também o algoritmo ElGamal para criptografar a palavra LNCC. A conveniência em usar o corpo $\mathbb{F}_{2^m}$ é que podemos representá-lo por *strings* de m *bits*, ou seja, cada elemento de $\mathbb{F}_{2^m}$ pode ser representado por um número binário entre 0 e $2^m - 1$. Usaremos a curva elíptica $\Omega : y^2 + xy = x^3 + g^{140}x^2 + g^{97}$. Observe que $a = g^{140} = x^6+x^5+x^3+x^2 = (01101100)$ e $b = g^{97} = x^6+x^2+x = (01000110)$, onde $g = x^7+x^3+x^2+x$ é um polinômio gerador de $\mathbb{F}_{2^8}$. Portanto, os pontos desta curva serão um par de polinômios pertencentes a $\mathbb{F}_{2^8}$. Por exemplo, o ponto (g^{63}, g^{81}) pertence a Ω, visto que

$$(g^{81})^2 + g^{63}g^{81} = (g^{63})^3 + g^{140}(g^{63})^2 + g^{97}$$
$$g^{162} + g^{144} = g^{189} + g^{11} + g^{97}$$
$$(10001100) \oplus (10110110) = (11010001) \oplus (10101101) \oplus (01000110)$$
$$(00111010) = (00111010).$$

Exemplo 6.2.2 *Novamente, é Alice quem enviará uma mensagem para Bob. Vamos supor que, depois de mapeado na curva elíptica* Ω, $LN = (g^{137}, g^{40})$ *e* $CC = (g^{82}, g^{102})$.

1. *Bob escolhe um inteiro,* $b = 77$, *e envia para Alice* $P = (g^{58}, g^{27}) = ((10011000), (10011110))$ *(a partir de agora, para a notação não ficar muito carregada, iremos usar* (g^{115}, g^{49}) *no lugar de* $((00110011), (10110010)$ *por exemplo) e* $K = bP = 77(g^{58}, g^{27}) = (g^{23}, g^{13})$, *sendo* P *um ponto da curva elíptica* $\Omega : y^2 + xy = x^3 + g^{140}x^2 + g^{97}$.

2. *Alice escolhe um inteiro* $a = 52$, *e multiplica* $c_1 = 52P = 52(g^{58}, g^{27}) = (g^{182}, g^{99})$. *Como a string já foi mapeada,* $m = LN = (g^{137}, g^{40})$. *Precisamos somar este ponto ao ponto obtido por* $aK = 52K = 52(g^{23}, g^{13}) = (g^2, g^{14})$. *Agora Alice esconde o ponto* $m = LN = (g^{137}, g^{40})$ *somando-o ao ponto* aK, *logo* $c_2 = m + aK = (g^{137}, g^{40}) + (g^2, g^{14}) = (g^{174}, g^7)$, *e envia os pontos cifrados* $(c_1, c_2) = ((g^{182}, g^{99}), (g^{174}, g^7))$.

3. *Bob agora precisa descriptografar o texto ilegível* $(c_1, c_2) = ((g^{182}, g^{99}), (g^{174}, g^7))$ *que recebeu de Alice, para isso deve calcular* $m = c_2 - bc_1$. *Primeiro Bob calcula o valor de* $bc_1 = 77c_1 = 77(g^{182}, g^{99}) = (g^2, g^{14})$. *Vimos que, se* $P = (x, y)$, *então* $-P(x, x+y)$. *Fazendo isso temos que* $-bc_1 = (g^2, g^2+g^{14}) = (g^2, g^{77})$. *Prosseguindo temos que* $m = c_2 - bc_1 = (g^{174}, g^7) - (g^2, g^{14}) = (g^{174}, g^7) + (-(g^2, g^{14})) = (g^{174}, g^7) + (g^2, g^{77}) = (g^{137}, g^{40})$; *observe que* $LN = (g^{137}, g^{40})$ *recuperado assim a primeira parte da mensagem.*

4. *Alice repete o mesmo procedimento, só que agora já temos o ponto* $c_1 = (g^{182}, g^{99})$. *A mensagem CC foi mapeada no ponto* $CC = (g^{87}, g^{102})$, *como foi dito anteriormente. Para obter o ponto* c_2, *ela computa* $c_2 = m + aK = (g^{87}, g^{102}) + (g^2, g^{14}) = (g^{66}, g^{135})$, *e envia para Bob o ponto* $c_2 = (g^{66}, g^{135})$.

5. *Bob agora calcula* $m = (g^{66}, g^{135}) + (g^2, g^{77}) = (g^{87}, g^{102})$. *Lembre-se que Bob já havia calculado o valor de* $-bc_1 = (g^2, g^{77})$, *recuperando a string CC e, assim, finalizando o algoritmo.*

6.3 O Criptossistema Menezes-Vanstone

Outra técnica muito usada para criptografar dados usando curvas elípticas é o método criptográfico Menezes-Vanstone [20]. Neste sistema, a mensagem é um par ordenado $m = (x_1, x_2)$, com $x_1, x_2 \in \mathbb{F}_p^*$, sendo que m não é um ponto da curva elíptica em questão, diferentemente do sistema anterior. O criptograma será uma tripla ordenada $r = (y_0, y_1, y_2)$, onde $y_1, y_2 \in \mathbb{F}_p^*$ e y_0 é um ponto da curva elíptica. Segue abaixo o algoritmo usado neste método.

Algoritmo:

Para criptografar $m = (x_1, x_2)$.

1. Bob escolhe um inteiro $k \in \mathbb{F}_p^*$ e calcula $y_0 = kP$ (lembre-se que Bob conhece publicamente o ponto $P \in \Omega$).

2. Bob computa $(c_1, c_2) = kQ$, $y_1 = c_1 x_1 \mod p$ e $y_2 = c_2 x_2 \mod p$. Envia para Alice a tripla $r = (y_0, y_1, y_2)$.

Para descriptografar $r = (y_0, y_1, y_2)$.

1. Alice calcula $sy_0 = skP = kQ = (c_1, c_2)$, onde s é um inteiro selecionado por Alice. Observe que $Q = sP$.

2. Logo após, Alice calcula $x_1 = y_1(c_1)^{-1} \mod p$ e $x_2 = y_2(c_2)^{-1} \mod p$, recuperando a mensagem $m = (x_1, x_2)$.

Vamos a um exemplo da aplicação deste sistema criptográfico.

Exemplo 6.3.1 *Bob irá enviar a mensagem MCT para Alice, tal mensagem será codificada na tabela ASCII. Então temos que* $M = 77$, $C = 67$ *e* $T = 84$. *Como a mensagem original será o par* $m = (x_1, x_2)$, *vamos separar em dois caracteres. Logo, a mensagem ficará* $m = (7767, 84)$. *Para Bob criptografar tal palavra ele precisa conhecer os pontos* $P = (1355793, 621792) \in \Omega$ *e* $Q = (949594, 812871) \in \Omega$, *onde* $\Omega : y^2 = x^3 + 67110x + 262147$ *está sobre* $\mathbb{F}_{2097421}$ *e* $Q = 78771 \cdot P$, *ou seja, Alice escolheu o inteiro* $s = 78771$.

1. *Bob escolhe* $k = 23358$, *e logo após computa* $y_0 = kP = (1390038, 1344654)$.

2. *Bob agora calcula* $kQ = (647014, 449701) = (c_1, c_2)$, $c_1 \cdot x_1 = 647014 \cdot 7767 = 2034443 \mod p$ *e* $c_2 \cdot x_2 = 449701 \cdot 84 = 21306 \mod p$. *Bob envia* $r = (y_0, y_1, y_2)$ *para Alice. Repare que* y_0 *é um ponto da curva elíptica e* y_1 *e* y_2 *são inteiros pertencentes a* $\mathbb{F}_{2097421}$.

3. *Para descriptografar Alice calcula* $s \cdot y_0 = (647014, 449701) = (c_1, c_2)$. *Para finalizar calcula* $x_1 = y_1(c_1)^{-1} = 7767 \mod p$ *e* $x_2 = y_2(c_2)^{-1} = 84 \mod p$, *recuperando a mensagem MCT.*

7 Detalhes da Implementação

Nesta seção apresentaremos detalhes pertinentes a implementação da API proposta nesta monografia. Na verdade, o grande objetivo deste trabalho é conceber uma API básica para curvas elípticas, ou seja, não estamos interessados na construção de protocolos baseados em curvas elípticas, estamos interessados em fornecer os pilares para construção dos sistemas criptográficos baseados em curvas elípticas. Portanto, a idéia é desenvolver uma ferramenta que possa dar total apoio para a concepção de protocolos que utilizam curvas elípticas. Em outras palavras, iremos fornecer uma implementação sólida para as operações no grupo formado pelos pontos de uma curva elíptica, no entanto, não estaremos preocupados em implementar o protocolo Menezes-Vanstone, por exemplo.

7.1 Linguagens e Ferramentas Utilizadas

As operações aritméticas básicas em curvas elípticas foram implementadas em `Java`. Esta linguagem dá o aparato para o desenvolvimento orientado a objeto. Este era um dos principais requisitos para a implementação da API exposta neste trabalho. Assim podemos ter uma API flexível a mudanças que provêm a oportunidade de criar e implementar componentes totalmente reutilizáveis usando o paradigma de orientação a objetos. O fato de ser uma linguagem multiplataforma motivou a escolha da linguagem `Java`. A grande popularidade que o `Java` alcançou na última década também teve um peso na escolha desta linguagem. Outro objetivo é fornecer a um maior espectro de programadores uma API para curvas elípticas. Na implementação foi necessário, de alguma maneira, utilizar inteiros de tamanho arbitrário. Em curvas elípticas, se utiliza números de aproximadamente 160 bits para uma segurança aceitável. Na linguagem `Java` é nativo o uso da classe `BigInteger` [3]. A mesma fornece todas as operações aritméticas para o desenvolvimento da API proposta. Uma operação demasiadamente utilizada é a obtenção do MDC entre dois inteiros. O trecho de código abaixo `Java` mostra, por exemplo, como se obtém o MDC entre dois inteiros

```
BigInteger a = new BigInteger("108101431022627944547");
BigInteger b = new BigInteger("343189580439649955 93");

BigInteger mdc = a.gcd( b );
```

7.2 Modelagem Orientada a Objetos

A modelagem pode ser dividida em relação a dois aspectos definidos neste trabalho: curvas elípticas sobre $\mathbb{Z}_p$ e curvas elípticas sobre $GF(2^m)$. Na modelagem usando o corpo $\mathbb{Z}_p$ temos basicamente duas classes: `Point` e `Curve`. As principais operações ficam a cargo da classe `Point`. O apêndice F.1 documenta os construtores e métodos da implementação sobre $\mathbb{Z}_p$. Em ambos os corpos ($GF(2^m)$ e $\mathbb{Z}_p$) as classes `Point` e `Curve` mantém o mesmo nome diferenciando apenas pelo pacote (*package*) que as contém. A figura 12 mostra a estrutura de pacotes de classes da API.

As classes `NAF` e `Utils` possuem em sua maioria métodos estáticos e serve para auxiliar o desenvolvimento da API. Veremos mais a frente detalhes sobre o funcionamento da classe `NAF`. A classe `GaloisField` simula um Corpo

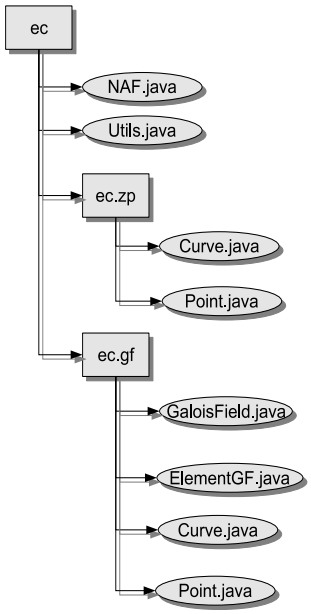

Figura 12: Árvore da API.

de Galois $GF(2^m)$. Na verdade, esta classe apenas encapsula um polinômio primitivo e seu respectivo grau. A classe `ElementGF` implementa as principais operações aritméticas de elementos pertencentes à um Corpo de Galois. Ambas as classes, bem com a classe `NAF`, estão documentadas no apêndice F.2. A figura 13 ilustra o diagrama de classe UML para implementação da API proposta sobre o Corpo de Galois $GF(2^m)$.

7.3 Algoritmos Utilizados

Basicamente os algoritmos mais importantes na aritmética das curvas elípticas são: multiplicação por um inteiro, soma entre pontos e dobra de um ponto. Nesta seção iremos mostrar em separado os algoritmos de soma e duplicação de um ponto, cada um usando um corpo finito ($GF(2^m)$ e $\mathbb{Z}_p$). Depois dos algoritmos específicos de cada corpo, mostraremos algoritmos genéricos para a multiplicação por escalar.

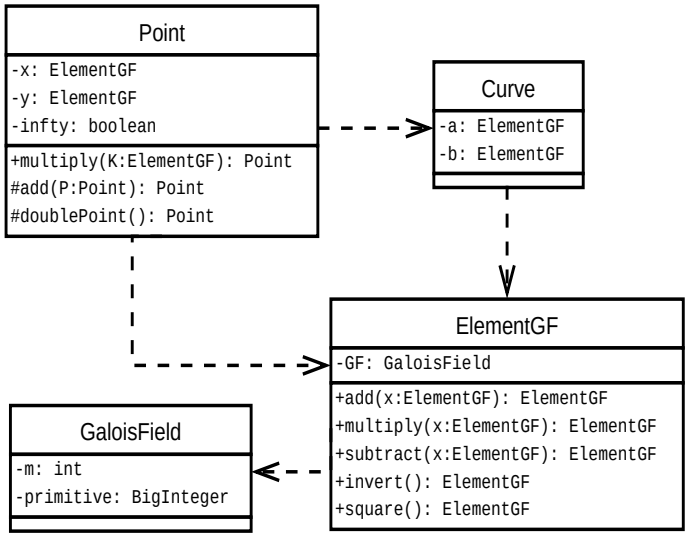

Figura 13: Diagrama de classes UML para implementação sobre $GF(2^m)$.

7.3.1 Algoritmos Sobre $\mathbb{Z}_p$

Agora iremos mostrar os algoritmos que utilizamos em curvas sobre $\mathbb{Z}_p$. O algoritmo 6 descreve a dobra de um ponto pertencente a uma curva elíptica

Este algoritmo segue diretamente da fórmula da equação (3) vista na seção 3.1. Já o algoritmo 7 mostra a soma entre dois pontos pertencentes a uma curva elíptica. Este também segue diretamente das fórmulas das equações (3).

7.3.2 Algoritmos Sobre $GF(2^m)$

Agora mostraremos os algoritmos de dobra e soma entre pontos que foram utilizados em curvas elípticas sobre Corpos de Galos $GF(2^m)$. Assim como na seção anterior, iremos começar pelo algoritmo de dobra (algoritmo 8).

O algoritmo 9 mostra como se calcula a soma entre pontos em $GF(2^m)$.

7.3.3 Algoritmo Binário para Multiplicação por Escalar

Nesta parte mostraremos uma técnica de multiplicação por escalar baseada no método de exponenciação binário. Os algoritmos que serão mostrados não dependem do tipo de corpo que a curva está definida, diferentemente das seções anteriores. O algoritmo 10 mostra o funcionamento do algoritmo

Algoritmo 6: Dobrando um ponto em $\Omega(\mathbb{Z}_p)$.

Entrada: Um ponto $P(x, y)$ pertencente a curva Ω e os parâmetros a, b e p da curva Ω
Saída: O ponto $2P \in \Omega$

início
  se $P = \infty \mid mdc(2y, p) \neq 1$ **então**
    retorna ∞;
  $t \leftarrow \frac{3x^2+a}{2y} \pmod{p}$;
  $x_r \leftarrow t^2 - 2x \pmod{p}$;
  $y_r \leftarrow t(x - x_r) - y \pmod{p}$;
  retorna $(x_r, y_r) \in \Omega$;
fim

Algoritmo 7: A soma $P + Q$ em $\Omega(\mathbb{Z}_p)$.

Entrada: $P(x_1, y_1), Q(x_2, y_2) \in \Omega$ e os parâmetros a, b e p da curva Ω
Saída: O ponto $P + Q \in \Omega$

início
  se $P = Q$ **então**
    retorna Dobra do ponto P (algoritmo 6);
  se $P = \infty$ **então**
    retorna Q;
  se $Q = \infty$ **então**
    retorna P;
  se $P = -Q$ **então**
    retorna ∞;
  $t \leftarrow \frac{y_2-y_1}{x_2-x_1} \pmod{p}$;
  $x_r \leftarrow t^2 - x_1 - x_2 \pmod{p}$;
  $y_r \leftarrow t(x_1 - x_r) - y \pmod{p}$;
  retorna $(x_r, y_r) \in \Omega$;
fim

Algoritmo 8: Dobrando um ponto em $\Omega(GF(2^m))$.

Entrada: Um ponto $P(x, y)$ pertencente a curva Ω e o parâmetro a da curva Ω
Saída: O ponto $2P \in \Omega$

início
  se $P = \infty$ **então**
    retorna ∞;
  $t \leftarrow x_1 + \frac{y_1}{x_1}$;
  $x_r \leftarrow t^2 + t + a$;
  $y_r \leftarrow x_1^2 + (t + 1)x_3$;
  retorna $(x_r, y_r) \in \Omega$;
fim

Algoritmo 9: A soma $P + Q$ em $\Omega(GF(2^m))$.

Entrada: $P(x_1, y_1), Q(x_2, y_2) \in \Omega$ e os parâmetros a, b da curva Ω
Saída: O ponto $P + Q \in \Omega$

início
  se $P = Q$ **então**
    retorna Dobra do ponto P (algoritmo 8);
  se $P = \infty$ **então**
    retorna Q;
  se $Q = \infty$ **então**
    retorna P;
  se $P = -Q$ **então**
    retorna ∞;
  $t \leftarrow \frac{y_2+y_1}{x_2+x_1}$;
  $x_r \leftarrow t^2 + t + x_1 + x_2 + a$;
  $y_r \leftarrow t(x_1 + x_r) + x_r + y_1$;
  retorna $(x_r, y_r) \in \Omega$;
fim

para multiplicação por escalar binário, na qual processa os bits de k da direita para esquerda.

Algoritmo 10: Multiplicação por escalar (método binário: versão direita para esquerda).

Entrada: Um inteiro $K = \sum_{i=0}^{j} 2^i k_i$ onde $k_i \in \{0, 1\}$(representação em base binária), um ponto $P \in \Omega$ e a curva Ω
Saída: O ponto $k \cdot P \in \Omega$

```
1 início
2     Q ← ∞;
3     para i = 0 até j faça
4         se k_i = 1 então
5             Q ← Q + P;
6         P ← 2 · P;
7     retorna Q ∈ Ω;
8 fim
```

Alternativamente podemos ter o algoritmo 11 que varre os bits de k da esquerda para direita.

A densidade média da representação binária de k é $\frac{1}{2}$, logo o tempo de execução esperado dos algoritmos 10 e 11 é de aproximadamente $(j + 1)/2$ adições de pontos (A) e $j + 1$ duplicações de pontos (D) denotado [25] por

$$\left(\frac{j+1}{2}\right) A + (j+1)D.$$

7.3.4 Algoritmo Baseado em NAF Binário

Para algum K inteiro podemos representá-lo da seguinte maneira:

$$K = \sum_{i=0}^{j} k_i 2^i$$

onde $k_i \in \{0, \pm 1\}$. A forma não-adjacente NAF, é a representação binária de K, onde $k_i k_{i+1} = 0$ para todo $i \geq 0$. A grande vantagem de se usar a representação NAF de um inteiro é que, de maneira geral, terá uma proporção maior de coeficientes iguais a zero. Assim fica reduzido o número de adições

Algoritmo 11: Multiplicação por escalar (método binário: versão esquerda para direita).

Entrada: Um inteiro $K = \sum_{i=0}^{j} 2^i k_i$ onde $k_i \in \{0, 1\}$(representação em base binária), um ponto $P \in \Omega$ e a curva Ω
Saída: O ponto $k \cdot P \in \Omega$

```
1 início
2 |  Q ← ∞;
3 |  para i = j até 0 faça
4 |  |  Q ← 2 · Q;
5 |  |  se k_i = 1 então
6 |  |  |  Q ← Q + P;
7 |  retorna Q ∈ Ω;
8 fim
```

na multiplicação. Outro importante fato é que a negação de um ponto tem tempo de execução desprezível. Assim, a eficiência desse algoritmo não é comprometida. F. Morain e J. Olivos [24] mostraram que o número esperado de coeficientes não nulos em uma representação NAF para um inteiro K com j bits é $j/3$. Na representação binária espera-se $j/2$ bits não nulos.

Exemplo 7.3.1 *Fazemos* $K = 52419574521$.
$(K)_2 = 1100001101000111001100111010111111001$
$NAF(K) = 10\bar{1}0001010\bar{1}0100100\bar{1}010\bar{1}0100010\bar{1}0\bar{1}0000\bar{1}001$
Onde $\bar{1} = -1$.

Abaixo algumas propriedades da representação NAF de um inteiro K [25]:

- Cada inteiro K possui única representação NAF.
- O comprimento da representação NAF de K é no máximo o tamanho da representação binária mais 1 [24].
- A densidade de coeficientes não nulos na representação NAF de K é $j/3$, onde $j = \log_2 K$.

O valor de NAF(K) pode ser calculado utilizado o algoritmo 12. O algoritmo usa o conceito de resto principal, que será denotado por (mods)

visto na seção 2.7. O algoritmo usa o fato que se K é ímpar, então o resto $r \in \{1, -1\}$ é escolhido de tal forma que K na próxima iteração seja par. Isto garante que o próximo dígito do NAF(K) seja 0.

Algoritmo 12: Cálculo da representação NAF.

Entrada: Um inteiro K
Saída: NAF(K)

```
1  início
2      i ← 0;
3      enquanto K ≥ 1 faça
4          se K é ímpar então
5              k_i ← K (mods 4);
6              K ← K − k_i;
7          senão
8              k_i ← 0;
9          K ← K ≫ 1;
10         i ← i + 1;
11     retorna (k_j, k_{j−1}, ..., k_0);
12 fim
```

O algoritmo 13 usa a representação NAF do inteiro K ao invés de sua representação binária. O tempo de execução para este algoritmo é de aproximadamente

$$\left(\frac{j+1}{3}\right) A + (j+1)D$$

onde $j = \log_2 K$, A é o tempo de execução para adição e D é o tempo de execução para duplicação de um ponto.

7.3.5 Algoritmo Baseado em NAF com Dimensão w

O valor de $\text{NAF}_w(K)$ para $w \geq 2$ é o uma representação $K = \sum_{i=0}^{j} k_i 2^i$ onde cada $|k_i| < 2^{w-1}$. Fazendo uma analogia com o NAF mostrado na seção anterior temos que $\text{NAF}_2(K) = \text{NAF}(K)$.

Exemplo 7.3.2 *Considere* $K = 52419574521$.
$NAF_2(K) = 10\bar{1}0001\bar{0}1\bar{0}10010\bar{0}1\bar{0}1\bar{0}10\bar{0}0\bar{0}1\bar{0}0\bar{0}0\bar{0}1001$

Algoritmo 13: Multiplicação por escalar usando NAF binário.

Entrada: Um inteiro K e um ponto $P \in \Omega$
Saída: $KP \in \Omega$

```
1  início
2      Calcule a representação NAF de K, Σ_{i=0}^{j} k_i 2^i =NAF(K)
       (algoritmo 12);
3      Q ← ∞;
4      para i = j até 0 faça
5          Q ← 2Q;
6          se k_i = 1 então
7              Q ← Q + P;
8          se k_i = −1 então
9              Q ← Q − P;
10     retorna Q;
11 fim
```

$NAF_3(K) = 3000100\bar{3}00100\bar{3}00\bar{3}00\bar{1}00300001001$
$NAF_4(K) = 3000007000\bar{7}000\bar{3}000\bar{3}00000\bar{5}00000007$

Abaixo algumas propriedades da expansão $\mathrm{NAF}_w(K)$

- Cada inteiro K possui única representação $\mathrm{NAF}_w(K)$.
- O comprimento da representação NAF de k é no máximo o tamanho da representação binária mais 1.
- A densidade de coeficientes não nulos na representação $\mathrm{NAF}_w(K)$ é $j/(w+1)$, onde $j = \log_2 K$.
- No máximo um de w coeficientes de $\mathrm{NAF}_w(K)$ (seguidos) é diferente de zero.

O cálculo de $\mathrm{NAF}_w(K)$ pode computado via o algoritmo 14. Na interação se K é ímpar então o resto $r \equiv K \pmod{2^w}$ é escolhido de forma que $(K-r)/2$ seja escolhido de forma que os próximos $(w-1)$ coeficientes sejam iguais a

0. O algoritmo 15 utiliza, assim como na seção anterior a representação não-adjacente para multiplicar um ponto em uma curva elíptica por um inteiro. Este algoritmo possui tempo esperado de aproximadamente:

$$((2^{w-2} - 1) + j/(w + 1))A + jD$$

onde $j = \log_2 K$.

Algoritmo 14: Cálculo de $\text{NAF}_w(K)$.

Entrada: Um inteiro K
Saída: $\text{NAF}_w(K)$

```
1  início
2      i ← 0;
3      enquanto K ≥ 1 faça
4          se K é ímpar então
5              k_i ← K (mods 2^w);
6              K ← K − k_i;
7          senão
8              k_i ← 0;
9          K ← K ≫ 1;
10         i ← i + 1;
11     retorna (k_j, k_{j−1}, ..., k_0);
12 fim
```

7.4 Desempenho

Nesta seção mostraremos o desempenho da API proposta. Na verdade, por se tratar da operação mais importante, só coletamos os tempos da operação de multiplicação por escalar em curvas elípticas tanto em $\mathbb{Z}_p$ quanto em $GF(2^m)$. Para os testes utilizamos os parâmetros recomendados pelo NIST (ver apêndice E). Multiplicamos a ordem do ponto (n) pelo próprio ponto, ambos definidos no apêndice E. O processador utilizado foi um Intel Pentium 4 ® com freqüência 2.40GHz. As tabelas 7 e 8 mostram os resultados em milissegundos das implementações sobre $\mathbb{Z}_p$ e $GF(2^m)$, respectivamente.

Algoritmo 15: Multiplicação por escalar usando NAF com dimensão w.

Entrada: Um inteiro K e um ponto $P \in \Omega$
Saída: $KP \in \Omega$

início
  Calcule a representação NAF de K, $\sum_{i=0}^{j} k_i 2^i = \mathrm{NAF}_w(K)$ (algoritmo 14);
  Precomputação: Calcule $P_i = iP$ para $i \in \{1, 3, 5, 7, \ldots, 2^{w-1} - 1\}$;
  $Q \leftarrow \infty$;
  para $i = j$ **até** 0 **faça**
    $Q \leftarrow 2Q$;
    se $k_i \neq 0$ **então**
      se $k_i > 0$ **então**
        $Q \leftarrow Q + P_{k_i}$;
      senão
        $Q \leftarrow Q - P_{-k_i}$;
  retorna Q;
fim

Curva	Método de Multiplicação por Escalar							
	Binário	NAF_2	NAF_3	NAF_4	NAF_5	NAF_6	NAF_7	NAF_8
P-192	105	38	30	33	33	44	60	97
P-224	58	43	43	41	43	51	71	118
P-256	75	58	54	55	57	66	89	144
P-384	207	144	138	138	141	156	194	287
P-521	442	300	312	288	292	317	376	524

Tabela 7: Desempenho da Multiplicação por Escalar ($\mathbb{Z}_p$).

Curva	Método de Multiplicação por Escalar							
	Binário	NAF_2	NAF_3	NAF_4	NAF_5	NAF_6	NAF_7	NAF_8
B-163	194	155	142	146	162	198	298	534
B-233	370	348	336	345	361	425	588	972
B-283	809	556	536	535	563	647	856	1361
B-409	1492	1414	1357	1363	1389	1539	1889	2773
B-571	5057	3449	3336	3290	3365	3597	4221	5763

Tabela 8: Desempenho da Multiplicação por Escalar ($GF(2^m)$).

8 Conclusões e Trabalhos Futuros

Concluímos que os métodos de criptografia sobre grupos de curvas elípticas possuem segurança equivalente ao RSA, entretanto, com quantidade de bits consideravelmente menor na chave criptográfica, o que possibilita um ganho substancial de processamento. Uma vez que a chave criptográfica pode ser reduzida, podemos ter criptografia com curvas elípticas em dispositivos com pouco poder computacional, por exemplo, cartões de banco. Isto se deve ao fato de que o PLD sobre pontos de uma curva elíptica é mais difícil de ser resolvido do que sobre um corpo primo, como foi mostrado na seção 4. Além disto, os portais de bancos na Internet poderão processar muito mais requisições sem a necessidade de aumentar sua capacidade de processamento. Assim, concluímos que deve ser recomendada a substituição do RSA por métodos de criptografia baseados em curvas elípticas. Como trabalhos futuros pretendemos melhorar o desempenho da API principalmente em curvas sobre o corpo $GF(2^m)$. Na verdade, é necessário melhorar os algoritmos da

aritmética entre elementos do Corpo de Galois. Só assim melhoraremos a performance das operações sobre curvas elípticas.

Referências

[1] *Recommended elliptic curves for federal government use*, 1999, Available at http://csrc.nist.gov/encryption.

[2] *Digital signature standard (DSS)*, National Institute of Standards and Technology, Washington, 2000, URL: `http://csrc.nist.gov/publications/fips/`. Note: Federal Information Processing Standard 186-2.

[3] *Java TM 2 Platform, Standard Edition, v 1.4.2 API Specification*, 2008, Available at http://java.sun.com/j2se/1.4.2/docs/api/.

[4] Leonard M. Adleman and Jonathan DeMarrais, *A subexponential algorithm for discrete logarithms over all finite fields*, CRYPTO '93: Proceedings of the 13th Annual International Cryptology Conference on Advances in Cryptology (London, UK), Springer-Verlag, 1994, pp. 147–158.

[5] P. Barreto, *Curvas elípticas e criptografia - conceitos e algoritmos*, 1999.

[6] Severino Collier Coutinho, *Números inteiros e criptografia RSA*, IMPA, Rio de Janeiro, RJ, Brasil, 1997.

[7] Whitfield Diffie and Martin E. Hellman, *New directions in cryptography*, IEEE Trans. Information Theory **IT-22** (1976), no. 6, 644–654. MR MR0437208 (55 #10141)

[8] Taher ElGamal, *A public key cryptosystem and a signature scheme based on discrete logarithms*, IEEE Trans. Inform. Theory **31** (1985), no. 4, 469–472. MR MR798552 (86j:94045)

[9] Daniel M. Gordon and Kevin S. McCurley, *Massively parallel computation of discrete logarithms*, Lecture Notes in Computer Science **740** (1993), 312–323.

[10] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone, *Guide to elliptic curve cryptography*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[11] L. H. Jacy Monteiro, *Elementos de álgebra*, 1 ed., LTC, 1974.

[12] David Kahn, *The codebreakers: the story of secret writing*, Scribner, New York, NY, USA, 1996.

[13] Neal Koblitz, *Elliptic curve cryptosystems*, Mathematics of Computation **48** (1987), no. 177, 203–209.

[14] Neal Koblitz, Alfred Menezes, and Scott Vanstone, *The state of elliptic curve cryptography*, Des. Codes Cryptography **19** (2000), no. 2-3, 173–193.

[15] Neal Koblitz and Alfred J. Menezes, *A survey of public-key cryptosystems*, SIAM Rev. **46** (2004), no. 4, 599–634 (electronic). MR MR2124678 (2006e:94036)

[16] Fabian Kuhn and Ren Struik, *Extensions of pollard's rho algorithm for computing multiple discrete logarithms*, SAC '01: Revised Papers from the 8th Annual International Workshop on Selected Areas in Cryptography (London, UK), Springer-Verlag, 2001, pp. 212–229.

[17] Endmund Landau, *Teoria elementar dos números*, Rio de Janeiro Editora Ciência Moderna, 1999.

[18] Jie Liang and Xuejia Lai, *Improved collision attack on hash function md5*, Cryptology ePrint Archive, Report 2005/425, 2005, http://eprint.iacr.org/.

[19] Alfred Menezes, Scott Vanstone, and Tatsuaki Okamoto, *Reducing elliptic curve logarithms to logarithms in a finite field*, STOC '91: Proceedings of the twenty-third annual ACM symposium on Theory of computing (New York, NY, USA), ACM Press, 1991, pp. 80–89.

[20] Alfred Menezes and Scott A. Vanstone, *Elliptic curve cryptosystems and their implementations.*, J. Cryptology **6** (1993), no. 4, 209–224.

[21] Alfred J. Menezes, Paul C. Van Oorschot, Scott A. Vanstone, and R. L. Rivest, *Handbook of applied cryptography*, 1997.

[22] Gary L. Miller, *Riemann's hypothesis and tests for primality*, Seventh Annual ACM Symposium on Theory of Computing (Albuquerque, N.M., 1975), Assoc. Comput. Mach., New York, 1975, pp. 234–239. MR MR0480296 (58 #470b)

[23] Victor S. Miller, *Use of elliptic curves in cryptography*, Advances in cryptology—CRYPTO '85 (Santa Barbara, Calif., 1985), Lecture Notes in Comput. Sci., vol. 218, Springer, Berlin, 1986, pp. 417–426. MR MR851432 (88b:68040)

[24] F. Morain and J. Olivos, *Speeding up the computations on an elliptic curve using addition-subtraction chains*, Theoretical Informatics and Applications **24** (1990), 531–543.

[25] A. M. Neto, *Multiplicação escalar eficiente em curvas elípticas*, Dissertação de Mestrado, IME – USP, 2006.

[26] NIST, *Secure hash standard*, FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION, 1995.

[27] Julio Heitor Silva Nóbrega, *Segurança em redes de computadores: Funções de hash em autenticação, paradoxo do aniversário e aplicações no protocolo wep*, COPPE – UFRJ, 2006.

[28] R. Rivest, *The md5 message-digest algorithm*, 1992.

[29] R. L. Rivest, A. Shamir, and L. Adleman, *A method for obtaining digital signatures and public-key cryptosystems*, Commun. ACM **21** (1978), no. 2, 120–126.

[30] R. Schoof, *Elliptic curves over finite fields and the computation of square roots mod p*, Mathematics of Computation **44** (1985), 483–494.

[31] R. Schoof, *Counting points on elliptic curves over finite fields*, J. Th'eor. Nombres Bordeaux (219–254) (1995).

[32] Igor A. Semaev, *An algorithm for evaluation of discrete logarithms in some nonprime finite fields*, Math. Comput. **67** (1998), no. 224, 1679–1689.

[33] C. E. Shannon, *Communication theory of secrecy systems*, Bell System Tech. J. **28** (1949), 656–715. MR MR0032133 (11,258d)

[34] S. Shokranian, M. Soares, and H. Godinho, *Teoria dos números*, Editora Universidade de Brasília, 1999.

[35] Simon Singh, *The code book: The evolution of secrecy from mary, queen of scots, to quantum cryptography*, Doubleday, New York, NY, USA, 1999.

[36] N. P. Smart, *The discrete logarithm problem on elliptic curves of trace one*, Journal of Cryptology: the journal of the International Association for Cryptologic Research **12** (1999), no. 3, 193–196.

[37] William Stallings, *Cryptography and network security: Principles and practice*, third ed., Pearson Education, 2002.

[38] Marc Stevens, *Fast collision attack on md5*.

[39] Routo Terada, *Segurança de dados: Criptografia em redes de computadores*, 1 ed., Edgard Blucher, 2000.

[40] Edlyn Teske, *Speeding up pollard's rho method for computing discrete logarithms*, ANTS-III: Proceedings of the Third International Symposium on Algorithmic Number Theory (London, UK), Springer-Verlag, 1998, pp. 541–554.

[41] E. De Win and B. Preneel, *Elliptic curve public-key cryptosystems — an introduction*, State of the Art in Applied Cryptography – Lecture Notes in Computer Science, Springer-Verlag **1528** (1998), 131–141.

Apêndice

A Teste de Primalidade Miller-Rabin

O teste Miller-Rabin [22] é um algoritmo probabilístico para testar a primalidade de um inteiro ímpar n. Este teste está baseado nos seguintes fatos:

1. Se n é um primo ímpar as únicas raízes quadradas de $1 \pmod{n}$ são 1 e $-1(\equiv (n-1))$. Se n fosse composto e não fosse uma potência de um primo, então 1 possui outras raízes quadradas módulo n.

2. Se n é um primo ímpar podemos escrever $n-1 = 2^s r$ onde r é ímpar. Seja $a \in \mathbb{Z}$ e $mdc(a,n) = 1$. Então $a^r \equiv 1 \pmod{n}$ ou $a^{2^j r} \equiv -1 \pmod{n}$ para algum j, $0 \leq j \leq (s-1)$.

Estas idéias motivas a seguinte definição [21]:

Definição A.0.1 *Seja n um inteiro ímpar e onde $n-1 = 2^s r$ onde r é um ímpar. E seja a definido no intervalo $1 \leq a \leq n-1$*

i) Se $a^r \not\equiv 1 \pmod{n}$ e $a^{2^j r} \not\equiv -1 \pmod{n}$ para todo j, $0 \leq j \leq (s-1)$ então a é definido como "forte testemunho" da não primalidade de n.

ii) Se, no entanto, $a^r \equiv 1 \pmod{n}$ ou $a^{2^j r} \equiv -1 \pmod{n}$ para algum j, $0 \leq j \leq (s-1)$ então chamamos n de "forte pseudo-primo" para a base a. E o inteiro a é definido como "falso testemunho" para a primalidade de n.

O parâmetro de segurança t do algoritmo que veremos indica qual a probabilidade de um inteiro composto ímpar n ser declarado como primo. Neste caso, temos uma chance menor que $\left(\frac{1}{4}\right)^t$. Com estas informações o algoritmo 16 mostra o funcionamento do teste de primalidade devido a miller-rabin.

Algoritmo 16: Teste probabilístico de primalidade Miller-Rabin.

Entrada: Um inteiro ímpar n e um parâmetro de segurança t
Saída: A reposta para a pergunta: "n é ou não um primo?"
início
  Escreva $n-1 = 2^s r$ onde r é um ímpar;
  para $i \leftarrow 1$ **até** t **faça**
    Escolha um inteiro aleatório a, $2 \leq a \leq n-2$;
    $y \leftarrow a^r \mod n$;
    se $y \neq 1$ E $y \neq (n-1)$ **então**
      $j \leftarrow 1$;
      enquanto $j \leq (s-1)$ E $y \neq (n-1)$ **faça**
        $y \leftarrow y^2 \mod n$;
        se $y = 1$ **então retorna** *"Composto"*;
        $j \leftarrow j+1$;
      se $y \neq (n-1)$ **então retorna** *"Composto"*;
  retorna *"Primo"*;
fim

B Algoritmo para Exponenciação Modular

O algoritmo que exporemos nesta seção (algoritmo 17) é crucial em muitos protocolos de criptografia assimétrica [21]. É importante que ele seja bastante eficiente, dada sua importância. O algoritmo que mostraremos se baseia na seguinte idéia onde k tem a representação binária $k = \sum_{i=0}^{t} k_i 2^i$ para $k_i \in \{0, 1\}$

$$a^k = \prod_{i=0}^{t} a^{k_i 2^i} = (a^{2^0})^{k_0} (a^{2^1})^{k_1} \ldots (a^{2^t})^{k_t}$$

Algoritmo 17: Exponenciação Modular.

Entrada: Inteiros a, k onde $0 \leq k < n$ e tem a representação binária $k = \sum_{i=0}^{t} k_i 2^i$
Saída: $a^k \mod n$
início
  $b \leftarrow 1$ **se** $k = 0$ **então**
    retorna b;
  $A \leftarrow a$;
  se $k_0 = 1$ **então**
    $b \leftarrow a$;
  para $i = 0$ **até** t **faça**
    $A \leftarrow A^2 \mod n$;
    se $k_0 = 1$ **então**
      $b \leftarrow A \cdot b \mod n$;
  retorna b;
fim

C Função de Möbius

Definimos a função de Möbius [17]

$$\mu : \mathbb{N} \to \mathbb{Z}$$

por

$$\mu(m) = \begin{cases} 1 & \text{se } m = 1 \\ (-1)^n & \text{se } m = p_1 \dots p_n \text{ com } p_1 \dots p_n \text{ primos distintos} \\ 0 & \text{se o quadrado de algum primo divide } m \end{cases}$$

Assim, $\mu(1) = \mu(6) = \mu(10) = 1$, $\mu(2) = \mu(3) = \mu(5) = \mu(7) = -1$ e $\mu(4) = \mu(8) = \mu(9) = 0$.

D O Algoritmo de Euclides Estendido para Polinômios

O algoritmo que segue abaixo terá como função calcular o inverso multiplicativo de um elemento em $GF(2^m)$ módulo um polinômio irredutível $m(X)$[37].

Algoritmo 18: Estendido de Euclides para Polinômios.

Entrada: Entrada: $f(X)$ (polinômio da qual deseja se obter o inverso) e $m(X)$ (polinômio irredutível)
Saída: $f^{-1}(X)$
início
  $A1(X) \leftarrow 1,\ A2(X) \leftarrow 0,\ A3(X) \leftarrow m(X);$
  $B1(X) \leftarrow 0,\ B2(X) \leftarrow 1,\ B3(X) \leftarrow f(X);$
  enquanto *Verdade* **faça**
    se $B3(X) = 0$ **então retorna** "$f(X)$ não possui inversa";
    se $B3(X) = 1$ **então retorna** $B2(X)$;
    $Q(X) \leftarrow A3(X)/B3(X);$
    $T1(X) \leftarrow A1(X) + Q(X) \cdot B(X);$
    $T2(X) \leftarrow A2(X) + Q(X) \cdot B2(X);$
    $T3(X) \leftarrow A3(X) + Q(X) \cdot B3(X);$
    $A1(X) \leftarrow B1(X),\ A2(X) \leftarrow B2(X),\ A3(X) \leftarrow B3(X);$
    $B1(X) \leftarrow T1(X),\ B2(X) \leftarrow T2(X),\ B3(X) \leftarrow T3(X);$
fim

E Curvas Recomendadas pelo NIST

As 5 curvas abaixo são curvas elípticas sobre $\mathbb{F}_p$ recomendadas pelo NIST. Os primos p referente ao corpo $\mathbb{F}_p$ foram escolhidos para dar maior velocidade na operação de redução $\mod p$. A escolha de $a = -3$ está relacionado com um melhor desempenho das operações aritméticas entre pontos de uma curva elíptica [1]. A seguir o significado de cada parâmetro

p	Primo referente a $\mathbb{F}_p$.
a, b	Coeficientes da curva $y^2 = x^3 + ax + b$.
n	Ordem do ponto base P. Ou seja, $n \cdot P = \infty$.
x, y	Coordenadas do ponto $P(x, y)$.
h	O cofator. Em outras palavras, $h = \#\Omega(\mathbb{F}_p)/n$.

Curva P-192: $p = 2^{192} - 2^{64} - 1, a = -3, h = 1$
$b =$ `0x 64210519 E59C80E7 0FA7E9AB 72243049 FEB8DEEC C146B9B1`
$n =$ `0x FFFFFFFF FFFFFFFF FFFFFFFF 99DEF836 146BC9B1 B4D22831`
$x =$ `0x 188DA80E B03090F6 7CBF20EB 43A18800 F4FF0AFD 82FF1012`
$y =$ `0x 07192B95 FFC8DA78 631011ED 6B24CDD5 73F977A1 1E794811`

Curva P-224: $p = 2^{224} - 2^{96} + 1, a = -3, h = 1$
$b =$ `0x B4050A85 0C04B3AB F5413256 5044B0B7 D7BFD8BA 270B3943 2355FFB4`
$n =$ `0x FFFFFFFF FFFFFFFF FFFFFFFF FFFF16A2 E0B8F03E 13DD2945 5C5C2A3D`
$x =$ `0x B70E0CBD 6BB4BF7F 321390B9 4A03C1D3 56C21122 343280D6 115C1D21`
$y =$ `0x BD376388 B5F723FB 4C22DFE6 CD4375A0 5A074764 44D58199 85007E34`

Curva P-256: $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1, a = -3, h = 1$
$b =$ `0x 5AC635D8 AA3A93E7 B3EBBD55 769886BC 651D06B0 CC53B0F6 3BCE3C3E 27D2604B`
$n =$ `0x FFFFFFFF 00000000 FFFFFFFF FFFFFFFF BCE6FAAD A7179E84 F3B9CAC2 FC632551`
$x =$ `0x 6B17D1F2 E12C4247 F8BCE6E5 63A440F2 77037D81 2DEB33A0 F4A13945 D898C296`
$y =$ `0x 4FE342E2 FE1A7F9B 8EE7EB4A 7C0F9E16 2BCE3357 6B315ECE CBB64068 37BF51F5`

Curva P-384: $p = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1, a = -3, h = 1$
$b =$ `0x B3312FA7 E23EE7E4 988E056B E3F82D19 181D9C6E FE814112 0314088F 5013875A C656398D 8A2ED19D 2A85C8ED D3EC2AEF`

$n =$ 0x FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
C7634D81 F4372DDF 581A0DB2 48B0A77A ECEC196A CCC52973
$x =$ 0x AA87CA22 BE8B0537 8EB1C71E F320AD74 6E1D3B62 8BA79B98
59F741E0 82542A38 5502F25D BF55296C 3A545E38 72760AB7
$y =$ 0x 3617DE4A 96262C6F 5D9E98BF 9292DC29 F8F41DBD 289A147C
E9DA3113 B5F0B8C0 0A60B1CE 1D7E819D 7A431D7C 90EA0E5F

Curva P-521: $p = 2^{521} - 1, a = -3, h = 1$
$b =$0x 00000051 953EB961 8E1C9A1F 929A21A0 B68540EE A2DA725B
99B315F3 B8B48991 8EF109E1 56193951 EC7E937B 1652C0BD
3BB1BF07 3573DF88 3D2C34F1 EF451FD4 6B503F00
$n =$ 0x 000001FF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
FFFFFFFF FFFFFFFF FFFFFFFA 51868783 BF2F966B 7FCC0148
F709A5D0 3BB5C9B8 899C47AE BB6FB71E 91386409
$x =$ 0x 000000C6 858E06B7 0404E9CD 9E3ECB66 2395B442 9C648139
053FB521 F828AF60 6B4D3DBA A14B5E77 EFE75928 FE1DC127
A2FFA8DE 3348B3C1 856A429B F97E7E31 C2E5BD66
$y =$ 0x 00000118 39296A78 9A3BC004 5C8A5FB4 2C7D1BD9 98F54449
579B4468 17AFBD17 273E662C 97EE7299 5EF42640 C550B901
3FAD0761 353C7086 A272C240 88BE9476 9FD16650

As próximas curvas também recomendadas pelo NIST. No entanto, estas estão sobre corpos de Galois da forma $\mathbb{F}_{2^m}$. Abaixo, uma explanação dos parâmetros

m	Expoente em $\mathbb{F}_{2^m}$.
$f(X)$	Polinômio irredutível de grau m.
a, b	Coeficientes da curva $y^2 + xy = x^3 + ax + b$.
n	Ordem do ponto base P. Ou seja, $n \cdot P = \infty$.
x, y	Coordenadas do ponto $P(x, y)$.
h	O cofator.

Curva B-163: $m = 163, f(X) = X^{163} + X^7 + X^6 + X^3 + 1, a = 1, h = 2$
$b =$ 0x 00000002 0A601907 B8C953CA 1481EB10 512F7874 4A3205FD
$n =$ 0x 00000004 00000000 00000000 000292FE 77E70C12 A4234C33
$x =$ 0x 00000003 F0EBA162 86A2D57E A0991168 D4994637 E8343E36

$y =$ 0x 00000000 D51FBC6C 71A0094F A2CDD545 B11C5C0C 797324F1

Curva B-233: $m = 233, f(X) = X^{233} + X^{74} + 1, a = 1, h = 2$
$b =$ 0x 00000066 647EDE6C 332C7F8C 0923BB58 213B333B 20E9CE42 81FE115F 7D8F90AD
$n =$ 0x 00000100 00000000 00000000 00000000 0013E974 E72F8A69 22031D26 03CFE0D7
$x =$ 0x 000000FA C9DFCBAC 8313BB21 39F1BB75 5FEF65BC 391F8B36 F8F8EB73 71FD558B
$y =$ 0x 00000100 6A08A419 03350678 E58528BE BF8A0BEF F867A7CA 36716F7E 01F81052

Curva B-283: $m = 283, f(X) = X^{283} + X^{12} + X^7 + X^5 + 1, a = 1, h = 2$
$b =$ 0x 027B680A C8B8596D A5A4AF8A 19A0303F CA97FD76 45309FA2 A581485A F6263E31 3B79A2F5
$n =$ 0x 03FFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFEF90 399660FC 938A9016 5B042A7C EFADB307
$x =$ 0x 05F93925 8DB7DD90 E1934F8C 70B0DFEC 2EED25B8 557EAC9C 80E2E198 F8CDBECD 86B12053
$y =$ 0x 03676854 FE24141C B98FE6D4 B20D02B4 516FF702 350EDDB0 826779C8 13F0DF45 BE8112F4

Curva B-409: $m = 409, f(X) = X^{409} + X^{87} + 1, a = 1, h = 2$
$b =$ 0x 0021A5C2 C8EE9FEB 5C4B9A75 3B7B476B 7FD6422E F1F3DD67 4761FA99 D6AC27C8 A9A197B2 72822F6C D57A55AA 4F50AE31 7B13545F
$n =$ 0x 01000000 00000000 00000000 00000000 00000000 00000000 000001E2 AAD6A612 F33307BE 5FA47C3C 9E052F83 8164CD37 D9A21173
$x =$ 0x 015D4860 D088DDB3 496B0C60 64756260 441CDE4A F1771D4D B01FFE5B 34E59703 DC255A86 8A118051 5603AEAB 60794E54 BB7996A7
$y =$ 0x 0061B1CF AB6BE5F3 2BBFA783 24ED106A 7636B9C5 A7BD198D 0158AA4F 5488D08F 38514F1F DF4B4F40 D2181B36 81C364BA 0273C706

Curva B-571: $m = 571, f(X) = X^{571} + X^{10} + X^5 + X^2 + 1, a = 1, h = 2$
$b =$ 0x 02F40E7E 2221F295 DE297117 B7F3D62F 5C6A97FF CB8CEFF1
CD6BA8CE 4A9A18AD 84FFABBD 8EFA5933 2BE7AD67 56A66E29
4AFD185A 78FF12AA 520E4DE7 39BACA0C 7FFEFF7F 2955727A
$n =$ 0x 03FFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
FFFFFFFF FFFFFFFF FFFFFFFF E661CE18 FF559873 08059B18
6823851E C7DD9CA1 161DE93D 5174D66E 8382E9BB 2FE84E47
$x =$ 0x 0303001D 34B85629 6C16C0D4 0D3CD775 0A93D1D2 955FA80A
A5F40FC8 DB7B2ABD BDE53950 F4C0D293 CDD711A3 5B67FB14
99AE6003 8614F139 4ABFA3B4 C850D927 E1E7769C 8EEC2D19
$y =$ 0x 037BF273 42DA639B 6DCCFFFE B73D69D7 8C6C27A6 009CBBCA
1980F853 3921E8A6 84423E43 BAB08A57 6291AF8F 461BB2A8
B3531D2F 0485C19B 16E2F151 6E23DD3C 1A4827AF 1B8AC15B

F Manual de Referência

F.1 Implementação sobre $\mathbb{Z}_p$

Classe `Curve`

Esta classe encapsula os parâmetros usuais de uma curva elíptica sobre $\mathbb{Z}_p$.

Construtores:

Constrói uma curva elíptica sobre $\mathbb{Z}_p$

```
public Curve(java.math.BigInteger a,
             java.math.BigInteger b,
             java.math.BigInteger p)
```

Parâmetros:
`a` - parâmetro da curva $y^2 = x^3 + ax + b$
`b` - parâmetro da curva $y^2 = x^3 + ax + b$
`p` - número primo que define o corpo finito $\mathbb{Z}_p$

Construtor por cópia

`public Curve(Curve E)`

Parâmetro:
`E` - curva que será copiada

Métodos:

`public java.math.BigInteger getP()`

Retorna:
O primo que define $\mathbb{Z}_p$

`public java.math.BigInteger getA()`

Retorna:
O parâmetro a da curva $y^2 = x^3 + ax + b$

`public java.math.BigInteger getB()`

Retorna:
O parâmetro b da curva $y^2 = x^3 + ax + b$

Compara a curva `this` com a curva E

`public boolean equals(Curve E)`

Parâmetro:
`E` - Curva que será comparada
Retorna:
`true` se são iguais, `false` caso contrário

Classe `Point`

Esta classe implementa as principais operações aritméticas de um ponto $P \in \Omega(\mathbb{Z}_p)$.

Construtores:

Constrói um ponto pertencente a um curva elíptica sobre $\mathbb{Z}_p$

```
public Point(java.math.BigInteger x,
             java.math.BigInteger y,
             boolean infty,
             Curve E)
```

Parâmetros:
`x` - Primeira coordenada
`y` - Segunda coordenada
`infty` - `true` caso seja um ponto no infinito, `false` caso contrário
`E` - Curva que este ponto pertence

Construtor para pontos que não sejam no infinito

```
public Point(java.math.BigInteger x,
             java.math.BigInteger y,
             Curve E)
```

Parâmetros:
`x` - Primeira coordenada
`y` - Segunda coordenada
`E` - Curva que este ponto pertence

Construtor por cópia

```
public Point(Point P)
```

`P` - Ponto que será copiado

Métodos:

```
public java.lang.String getString()
```

Retorna:
O ponto `this` formatado

`public java.math.BigInteger getX()`

Retorna:
A primeira coordenada x

`public java.math.BigInteger getY()`

Retorna:
A segunda coordenada y

`public boolean isInftyPoint()`

Retorna:
`true` se é um ponto no infinito, `false` caso contrário

`public ec.zp.Curve getCurve()`

Retorna:
A curva que o ponto `this` pertence

Compara o ponto `this` com o ponto `P`

`public boolean equals(Point P)`

Parâmetro:
`P` - ponto que será comparado
Retorna:
`true` se são iguais, `false` caso contrário

Calcula o inverso aditivo do ponto `this`

`public Point negate()`

Retorna:
$-$`this`

Multiplica o escalar `K` pelo ponto `this`

```
public Point multiply(java.math.BigInteger K)
```

Parâmetro:
`K` - Inteiro que multiplicará o ponto `this`
Retorna:
`R`, onde `R = K · this`

F.2 Implementação sobre $GF(2^m)$

Classe `GaloisField`

Esta classe encapsula o grau do Corpo de Galois, bem como o seu polinômio primitivo, onde toda aritimética será calculada módulo ele.

Construtores:

```
public GaloisField(int m,
                   java.math.BigInteger primitive)
```

Parâmetros:
`m` - Grau do Corpo de Galois $GF(2^m)$
`primitive` - Polinômio primitivo de $GF(2^m)$

```
public GaloisField(int m,
                   java.lang.String primitive)
```

Parâmetros:
`m` - Grau do Corpo de Galois $GF(2^m)$
`primitive` - Polinômio primitivo de $GF(2^m)$

Constrói um objeto do tipo `GaloisField` onde o polinômio primitivo é da forma: $f(X) = X^m + X^{k1} + X^{k2} + X^{k3} + 1$

```
public GaloisField(int m,
                   int k1,
                   int k2,
                   int k3)
```

Métodos:

`public int getDegree()`

Retorna:
O grau do polinômio primitivo

`public java.math.BigInteger getPrimitive()`

Retorna:
O polinômio primitivo

Compara dois objetos do tipo `GaloisField`

`public boolean equals(GaloisField GF)`

Parâmetro:
`A` - Objeto que será comparado
Retorna:
`true` se são iguais, `false` caso contrário

Classe `ElementGF`

Esta classe abstrai grande parte das funcionalidades de um elemento do Corpo de Galois $GF(2^m)$. As principais operações aritméticas, incluindo soma, subtração, multiplicação, divisão, elevação ao quadrado e inversão foram implementadas.

Construtores:

Constrói um elemento nulo (zero) do Corpo de Galois

`public ElementGF(GaloisField GF)`

Parâmetro:
`GF` - Corpo que o elemento `this` pertencerá

Constrói um elemento do Corpo de Galois

```
public ElementGF(java.math.BigInteger element,
                 GaloisField GF)
```

Parâmetros:
`element` - Valor numérico do elemento `this`
`GF` - Corpo que o elemento `this` pertence

Constrói um elemento do Corpo de Galois

```
public ElementGF(java.lang.String element,
                 GaloisField GF)
```

Parâmetros:
`element` - Valor numérico do elemento `this`
`GF` - Corpo que o elemento `this` pertence

Constrói um elemento do Corpo de Galois

```
public ElementGF(java.lang.String element,
                 int base,
                 GaloisField GF)
```

Parâmetros:
`element` - Valor numérico do elemento `this`
`base` - base numérica que `elemet` está representada
`GF` - Corpo que o elemento `this` pertence

Métodos:

```
public java.math.BigInteger get()
```

Retorna:
O valor numérico do elemento `this`

```
public GaloisField getGF()
```

Retorna:
O Corpo de Galois na qual o `this` pertence

Calcula o resto da divisão entre elementos de $GF(2^m)$

```
public ElementGF mod(ElementGF m)
```

Retorna:
`this` mod `m`

Compara dois objetos do tipo `ElementGF`

```
public boolean equals(ElementGF x)
```

Parâmetro:
`x` - Objeto que será comparado
Retorna:
`true` se são iguais, `false` caso contrário

Calcula o quadrado de `this`

```
public ElementGF square()
```

Retorna:
$\texttt{this}^2$

Operação de multiplicação em $GF(2^m)$

```
public ElementGF multiply(ElementGF x)
```

Parâmetro:
`x` - Elemento que será multiplicado ao elemento `this`
Retorna:
O resultado de `this` $\cdot$ `x`

Operação de soma entre elementos de $GF(2^m)$

```
public ElementGF add(ElementGF x)
```

Parâmetro:
`x` - Elemento que será somado ao elemento `this`
Retorna:
O resultado de `this` + `x`

Operação de subtração entre elementos de $GF(2^m)$

```
public ElementGF subtract(ElementGF x)
```

Parâmetro:
`x` - Elemento que será subtraído ao elemento `this`
Retorna:
O resultado de `this` − `x`

Cálculo do inverso multiplicativo em $GF(2^m)$

```
public ElementGF inverse()
```

Retorna:
$\texttt{this}^{-1}$

Operação de divisão em $GF(2^m)$

```
public ElementGF divide(ElementGF x)
```

Parâmetro:
`x` - Elemento que será dividirá `this`
Retorna:
`this` / `x`

```
public int deg()
```

Retorna:
O grau do elemento `this`

Classe `Curve`

Esta classe encapsula os parâmetros usuais de uma curva elíptica $\Omega : y^2+xy = x^3 + ax^2 + b$ sobre $GF(2^m)$.

Construtores:

Constrói uma curva elíptica sobre $\mathbb{Z}_p$

```
public Curve(ElementGF a,
             ElementGF b)
```

Parâmetros:
`a` - parâmetro da curva $y^2 + xy = x^3 + ax^2 + b$
`b` - parâmetro da curva $y^2 + xy = x^3 + ax^2 + b$

Construtor por cópia

```
public Curve(Curve E)
```

Parâmetro:
`E` - curva que será copiada

Métodos:

```
public ElementGF getA()
```

Retorna:
O parâmetro a da curva $y^2 = x^3 + ax + b$

```
public ElementGF getB()
```

Retorna:
O parâmetro b da curva $y^2 = x^3 + ax + b$

Compara a curva `this` com a curva E

```
public boolean equals(Curve E)
```

Parâmetro:
`E` - Curva que será comparada
Retorna:

`true` se são iguais, `false` caso contrário

Classe `Point`

Esta classe implementa as principais operações aritméticas de um ponto $P \in \Omega(GF(2^m))$.

Construtores:

Constrói um ponto de uma curva elíptica $\Omega(GF(2^m))$

```
public Point(ElementGF x,
             ElementGF y,
             boolean infty,
             Curve E)
```

Parâmetros:
`x` - Primeira coordenada
`y` - Segunda coordenada
`infty` - `true` caso seja um ponto no infinito, `false` caso contrário
`E` - Curva que este ponto pertence

Construtor para pontos que não sejam no infinito

```
public Point(ElementGF x,
             ElementGF y,
             Curve E)
```

Parâmetros:
`x` - Primeira coordenada
`y` - Segunda coordenada
`E` - Curva que este ponto pertence

Construtor por cópia

```
public Point(Point P)
```

`P` - Ponto que será copiado

Métodos:

`public ElementGF getX()`

Retorna:
A primeira coordenada x

`public ElementGF getY()`

Retorna:
A segunda coordenada y

`public boolean isInftyPoint()`

Retorna:
`true` se é um ponto no infinito, `false` caso contrário

`public ec.gf.Curve getCurve()`

Retorna:
A curva que o ponto `this` pertence

Compara o ponto `this` com o ponto `P`

`public boolean equals(Point P)`

Parâmetro:
`P` - ponto que será comparado Retorna:
`true` se são iguais, `false` caso contrário

Calcula o inverso aditivo do ponto `this`

`public Point negate()`

Retorna:
$-$`this`

Multiplica o escalar `K` pelo ponto `this`

`public Point multiply(java.math.BigInteger K)`

Parâmetro:
`K` - Inteiro que multiplicará o ponto `this`
Retorna:
`R`, onde `R = K · this`
