Universidade de Brasília

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Estudo e Implementação do Protocolo ECDSA

Filipe Tancredo Barros

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Msc. João José Costa Gondim

Brasília
2015

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Coordenador: Prof. Dr. Homero Luiz Piccolo

Banca examinadora composta por:

Prof. Msc. João José Costa Gondim (Orientador) — CIC/UnB
Prof. Dr. Homero Luiz Piccolo — CIC/UnB
Dr. Dino Macedo Amaral —

CIP — Catalogação Internacional na Publicação

Barros, Filipe Tancredo.

Estudo e Implementação do Protocolo ECDSA / Filipe Tancredo Barros. Brasília : UnB, 2015.
213 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2015.

1. Assinatura, 2. Digital, 3. Curvas Elípticas, 4. Criptografia,
5. ECDSA, 6. ECC

CDU

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Estudo e Implementação do Protocolo ECDSA

Filipe Tancredo Barros

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Msc. João José Costa Gondim (Orientador)
CIC/UnB

Prof. Dr. Homero Luiz Piccolo   Dr. Dino Macedo Amaral
CIC/UnB

Prof. Dr. Homero Luiz Piccolo
Coordenador do Bacharelado em Ciência da Computação

Brasília, 5 de março de 2015

Dedicatória

Dedico este projeto minha família, que sempre foi meu porto seguro, que nunca deixou de me apoiar em minhas escolhas e a quem devo tudo que sou. Ao meu pai, que é um exemplo de vida profissional e pessoal, minha mãe que é um exemplo de pessoa, meu irmão, que é um exemplo de perseverança. E a meus falecidos avós, Lecy e Rubem, que são meus espelhos para o resto de minha vida.

Agradecimentos

Agradeço primeiramente a Deus, que sempre me da forças. Agradeço a meus pais que me de deram força quando eu já tinha desistido de mim, ao professor Gondim que aceitou me receber como orientando deste projeto depois que meu antigo orientador teve que me deixar, pois estava mudando de faculdade. Agradeço a todo corpo de professores que passaram por minha vida, tanto escolar quanto universitária, um agradecimento especial ao professor Marcus V. Lamar que me orientou durante o curso. Agradeço aos amigos que também me deram força.

Abstract

No mundo de hoje, a assinatura digital é uma ferramenta indispensável nos sistemas de reconhecimento. Cada vez mais, contas são pagas de forma eletrônica, pessoas se identificam de forma virtual, e a pergunta é como saber se este indivíduo é quem diz que é. Assim, vários protocolos foram criados para fazer uma maneira segura de prover esta autenticidade. Neste terreno, um novo conceito, as curvas elípticas, começou a ser usado. Neste documento falaremos do protocolo baseado em curvas elípticas, o ECDSA, e todos os conceitos necessários para entendê-lo, além de documentar um conjunto de módulos simples para linguagem C++, implementado para demonstrar a concretização do aprendizado.

Palavras-chave: Assinatura, Digital, Curvas Elípticas, Criptografia, ECDSA, ECC

Abstract

Nowadays, the digital signature is a indispensable tool in recognition systems. Increasingly, bills are paid in a electronic form, people identify themselves in a virtual way, and the question is how to know that the individual is who he says. Therefore, several protocols have been created to make a safe way to provide this authenticity. In this territory, a new concept, the elliptic curves, start to be used. In this document, we'll talk about a protocol based in elliptic curves, the ECDSA, and all the necessaries concepts to understand it, addition to documenting a set of simple modules in C++ language, implemented to show the embodiment of the learning.

Keywords: Signature, Digital, Elliptic Curve, Cryptography, ECDSA, ECC

Sumário

1 Introdução **1**

2 Conceitos Básicos **3**
- 2.1 Teoria dos números . . . 3
  - 2.1.1 Operação Modular . . . 3
  - 2.1.2 Algoritmo de Euclides e o EEA . . . 5
  - 2.1.3 Grupos Cíclicos . . . 7
  - 2.1.4 Campo de Galois . . . 8
- 2.2 Criptografia . . . 8
  - 2.2.1 Sistema Criptográfico . . . 10
  - 2.2.2 Serviços Criptográficos . . . 11
  - 2.2.3 Assinatura Digital . . . 14
  - 2.2.4 Função de Resumo Criptográfico . . . 17
  - 2.2.5 SHA(Secure Hash Algorithm) . . . 21
- 2.3 Curvas Elípticas . . . 24
  - 2.3.1 Operação em Curvas Elípticas . . . 25
  - 2.3.2 Definindo o Problema do Logaritmo Discreto com Curvas Elípticas 28
- 2.4 Segurança Computacional do Problema do Logaritmo Discreto . . . 30

3 Protocolo ECDSA **31**
- 3.1 Esquema criptográfico de Elgamal . . . 31
  - 3.1.1 Geração das Chaves . . . 31
  - 3.1.2 Cifragem . . . 32
  - 3.1.3 Decifragem . . . 32
- 3.2 Assinatura de Elgamal . . . 32
  - 3.2.1 Assinatura . . . 33
  - 3.2.2 Verificação . . . 33
  - 3.2.3 Prova de Validade . . . 33
- 3.3 Algoritmo de Assinatura Digital . . . 34
  - 3.3.1 Geração de Chaves . . . 34
  - 3.3.2 Assinatura . . . 34
  - 3.3.3 Verificação . . . 35
  - 3.3.4 Prova de Validade . . . 35
- 3.4 Algoritmo de Assinatura Digital em Curvas Elípticas . . . 35
  - 3.4.1 Geração de Chaves . . . 36
  - 3.4.2 Assinatura . . . 36

3.4.3 Verificação . . . 36
3.4.4 Prova de Validade . . . 37

4 Documentação do Módulo 38
4.1 basics.h . . . 38
4.2 bigint.h . . . 39
4.2.1 Tipo bigint . . . 41
4.2.2 init_bigint . . . 41
4.2.3 not_null . . . 42
4.2.4 neg_bigint . . . 42
4.2.5 random_bigint . . . 43
4.2.6 ler_bigint . . . 43
4.2.7 ler_hex_bigint . . . 44
4.2.8 fler_hex_bigint . . . 45
4.2.9 ler_decimal . . . 45
4.2.10 imprime_hex_bigint . . . 46
4.2.11 fimprime_hex_bigint . . . 47
4.2.12 imprime_char_bigint . . . 47
4.2.13 imprime_decimal . . . 48
4.2.14 eq_bigint e eqi_bigint . . . 49
4.2.15 lt_bigint e ltu_bigint . . . 50
4.2.16 let_bigint . . . 51
4.2.17 atrib_bigint e atribi_bigint . . . 51
4.2.18 append_int_bigint . . . 52
4.2.19 srl1_bigint e sll1_bigint . . . 53
4.2.20 add_bigint e addi_bigint . . . 53
4.2.21 sub_bigint e subi_bigint . . . 54
4.2.22 mult_bigint e multi_bigint . . . 55
4.2.23 div_mod_bigint e div_mod_i_bigint . . . 56
4.2.24 pow_modp_bigint . . . 58
4.3 sha256.h . . . 60
4.3.1 rightRot . . . 61
4.3.2 arq_sha256 . . . 61
4.3.3 tex_sha256 . . . 62
4.3.4 sha256 . . . 63
4.4 Algebra_op.h . . . 66
4.4.1 EEA . . . 67
4.4.2 gcd . . . 68
4.4.3 inverse_eea . . . 69
4.5 EC_Point.h . . . 69
4.5.1 Construtores . . . 70
4.5.2 atribuir . . . 71
4.5.3 Métodos get e set . . . 71
4.6 EC_op.h . . . 72
4.6.1 add . . . 72
4.6.2 dobrar . . . 74

4.6.3 mult . . . 75
4.7 ECDSA.h . . . 76
4.7.1 Registros pub_key e signature . . . 77
4.7.2 key_generator . . . 78
4.7.3 sign . . . 78
4.7.4 verify . . . 79
4.8 Interface de exemplo - 256ECDSA.cpp . . . 80
4.8.1 Escolha de parâmetros e Menu . . . 80
4.8.2 Gerar ou Inserir Chaves . . . 83
4.8.3 SHA-256 de arquivo ou texto . . . 86
4.8.4 Assinar arquivo ou texto . . . 88
4.8.5 Verificar Assinatura de arquivo ou texto . . . 90

5 Conclusão **94**

Referências **96**

Lista de Figuras

2.1 Digrama de Comunicação . . . 9
2.2 Cifragem Simétrica . . . 10
2.3 Autorização . . . 12
2.4 Autenticação Subjetiva . . . 13
2.5 Cifragem Assimétrica . . . 13
2.6 Autenticação Objetiva . . . 14
2.7 Assinatura Digital RSA . . . 16
2.8 Função de Hash . . . 18
2.9 Função de Hash - Integridade . . . 18
2.10 Função de Resumo criptográfico - Ataque de Aniversário . . . 19
2.11 Função de Resumo criptográfico - Resumo criptográfico Iterado . . . 20
2.12 Curva Elíptica - retirado de [13] pg.241 . . . 24
2.13 EC: soma de pontos - retirado de [13] pg.243 . . . 25
2.14 EC: dobra do ponto - retirado de [13] pg.243 . . . 26
2.15 EC: Elemento Neutro - retirado de [13] pg.245 - com alterações . . . 27

4.1 Diagrama de Comunicação . . . 39
4.2 Interface: Menu de Parâmetros . . . 82
4.3 Arquivo de Parâmetros . . . 83
4.4 Interface: Menu Principal . . . 83
4.5 Interface: Gerando Chave . . . 84
4.6 Chaves Geradas . . . 84
4.7 Chaves Não Inseridas . . . 86
4.8 Chaves Inseridas . . . 87
4.9 Interface do SHA-256 . . . 88
4.10 Assinatura de Arquivo . . . 89
4.11 Assinatura de Texto . . . 90
4.12 Verificação de Assinatura . . . 93
4.13 Verificação de Texto . . . 93

Lista de Tabelas

2.1 Tabela de Cifra de César 10

4.1 Tabela de Armazenamento do tipo bigint 44
4.2 Tabela de Armazenamento Retificado 44
4.3 Concatenação com descarte 52
4.4 Pre-processamento do SHA-256 61

Capítulo 1

Introdução

As interações sociais sempre se fizeram presentes, seja para simplesmente se fazer entender, seja para trocar idéias. O tempo foi passando e estas interações evoluíram para o comércio, acordos, entre outros tipos, em que a necessidade de garantir a autenticidade de certa pessoa era importante. Ainda é comum remeter à idéia de presença física sempre que falamos o termo *Sua Palavra* ao se referir a uma promessa ou algo do tipo.

Com a evolução da comunicação a presença física em uma interação se fez desnecessária, com o telefone e mais adiante com a internet. Entretanto, como citado anteriormente, existem interações que necessitam de garantias de autenticidade, para estas o *corpo a corpo* ainda era necessário. Mas como acabar com esta necessidade?

Deste ponto iniciou-se a busca por métodos de autenticação pessoal. Entre as propostas, as que realmente se provaram eficazes foram a assinatura manual e a impressão digital, onde a idéia é que cada uma destas poderia apenas ser feita pelo dono da mesma. No caso da impressão digital, é conhecida a unicidade de cada uma, em outras palavras, cada pessoa possui uma digital, e nenhuma outra possui uma igual. Mas provar a autenticidade de uma digital necessita de um banco de armazenamento de digitais e um perito, pois é complicado perceber as diferenças existentes entre elas. Assim, antes da evolução dos computadores e da automatização dos processos de perícia, a assinatura manual foi o método escolhido para a maioria das interações em que se fazia necessária a autenticação.

Com a evolução da computação e da matemática, principalmente no que se diz respeito a criptografia, um novo método de autenticação começou a ser estudado. A este deu-se o nome de assinatura digital. Neste caso, cada pessoa ou entidade possui um par de chaves único em que uma das chaves é utilizada para assinar um documento digital e a outra é utilizada para verificar se a assinatura foi feita pela chave anterior.

Acontece que a assinatura digital nada mais é que um conjunto de métodos matemáticos de alta complexidade que apenas computadores podem executar e que processos de cálculos inversos feitos na tentativa de descobrir a chave que assina um documento são computacionalmente exaustivos. Com a evolução dos métodos matemáticos e da velocidade de processamento dos computadores, vários padrões criados começaram a se provar inseguros. Aumentou assim, a necessidade de novos padrões, mais robustos e resistentes aos ataques conhecidos.

No meio da década de 80 dois matemáticos, Neal Koblitz [8] e Victor S. Miller [10] de forma independente propuseram um novo método criptográfico baseado em curvas elípticas, método que, por se basear em uma área da matemática pouco explorada, além

de algumas razões políticas que fizeram esse ser visto como inferior ao método utilizado na época pela NSA [1], não foi imediatamente aceito. Na verdade apenas no final do século, em 1999 para ser mais exato, é que esta técnica começou a ser utilizada em processos de alta importância, como por exemplo na rede *BitCoin*, criada por Satoshi Nakamoto [2] [11], em que praticamente tudo é baseado em criptografia, inclusive a própria moeda utilizada na rede. Mais adiante, outras empresas e agências, inclusive a NSA, se renderam a este novo método e começaram a usá-lo.

Dado o grande crescimento do uso do padrão baseado em curvas elípticas, que ainda não foi muito estudado em relação a outros padrões, este documento tem como objetivo estudar e explicar como funciona o processo de assinatura digital baseado em curvas elípticas, mais especificamente o protocolo [3] ECDSA [4] de 256 bits [5], e documentar uma implementação de módulos simples feita em C++ que simula este protocolo, com o objetivo de demonstrar a concretização do conhecimento obtido no estudo deste projeto.

Com o estudo, é esperado poder explicar o porque deste crescimento no uso das curvas elípticas dentro da criptografia, o seu alto desempenho, e a demora para esta matemática ser aceita pelas agências de segurança do mundo.

No capítulo 2 será descrito alguns conceitos básicos para o melhor entendimento da assinatura, como alguns conceitos de teoria dos números, criptografia em geral e matemática de curvas elípticas. No capítulo 3 será explicado como é o protocolo ECDSA, por fim, no capítulo 4, está a documentação do conjunto de módulos que simulam este mesmo protocolo.

[1] National Security Agency: agência Nacional de Segurança - Estados Unidos.

[2] Satoshi Nakamoto é um pseudônimo de uma pessoa ou um grupo de pessoas que além de ter sido o primeiro a possuir 1 bitcoin, produziu o paper explicando o protocolo da rede BitCoin. Há apenas especulações de quem realmente seja.

[3] Padrão definido por uma entidade para conexão, comunicação ou transferência de dados entre dois sistemas computacionais.

[4] Elliptic Curve Digital Signature Algorithm: protocolo de assinatura digital que é baseado em curva elíptica.

[5] símbolo que define o estado de um transistor na memória do computador.

Capítulo 2

Conceitos Básicos

Para começar o estudo é preciso abordar alguns conceitos que são necessários para o entendimento da assinatura digital de curvas elípticas. Como o sistema de assinatura digital pertence ao mundo da criptografia, será abordado os conceitos que giram em torno da mesma e suas diferentes aplicações. Como parte do estudo criptográfico será explanado alguns conceitos dentro da teoria dos números.

Depois de falar da criptografia de modo geral, será abordado o sistema de assinatura digital, explicando como ele funciona de forma geral e aproveitando para descrever o conceito de *resumo criptográfico* [1] e a *função de resumo criptográfico.*

Por fim será introduzido a matemática que gira em torno das curvas elípticas e os conceitos algébricos que giram em torno dela, para, por fim, será descrito o protocolo ECDSA.

2.1 Teoria dos números

Esta seção abordará alguns conceitos importantes em teoria dos números, começando por operações modulares.

2.1.1 Operação Modular

No contexto da criptografia, tanto simétrica quanto assimétrica, termos que serão abordados depois, praticamente todos os algoritmos se utilizam de um grupo finito de elementos, mas quase todos os conjuntos numéricos conhecidos são infinitos, até mesmo o grupo dos naturais a qual tem como elementos $0, 1, 2, ...$ até o infinito. Como então definir um grupo que seja finito?

Primeiro vamos selecionar um grupo de elementos G, tal que:

$G = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

Agora vamos selecionar operações para serem feitas entre eles, mas que o resultado continue dentro deste grupo, por exemplo:

[1] Em inglês definido como hash.

$$
\begin{aligned}
2 + 2 &= 4 \\
3 + 6 &= 9 \\
3 * 3 &= 9 \\
2 * 5 &= 10
\end{aligned}
\tag{2.1}
$$

Mas e $4 * 3 =?$ ou $7 + 8 =?$ Ambos dão resultados que estão fora do Grupo. Para este caso será introduzida uma regra: depois de toda operação deverá ser calculado o resto da divisão por 11, sendo este o resultado final e a este resultado é denotado de resultado modulo 11. Por exemplo:

$7 + 8 = 15 \Rightarrow 15/11 = 1$ com resto 4, assim $7 + 8 = 15 \equiv 4 \pmod{11}$

ou como no outro exemplo:

$$4 * 3 = 12 \equiv 1 \pmod{11} \tag{2.2}$$

Dados estes exemplos, define-se *operação modular* assim:

Dado $r, a, m \in Z$ e $m > 0$. Afirma-se que $a \equiv r \pmod{m}$ se, e somente se, $m|(a - r)$ onde $|$ significa *divide*. Neste caso m é chamado de módulo e r de resíduo, ou resto.

Um ponto importante a ser levantado é que o resíduo não é único, visto que, pela definição da operação, tanto 1 quanto -10 podem ser resíduos 12 módulo 11, veja:

$$
\begin{aligned}
12 \equiv 1 \pmod{11} &\Leftrightarrow 11|(12 - 1) \Rightarrow 11|11 \\
12 \equiv -10 \pmod{11} &\Leftrightarrow 11|(12 - (-10)) \Rightarrow 11|22
\end{aligned}
\tag{2.3}
$$

Ou seja, ambos estão corretos. A este grupo de elementos que possuem equivalência módulo m, onde $m \in Z$, são chamados de *classe de equivalência*, mas para fins criptográficos, preferimos usar o menor inteiro não negativo da classe, pois facilita os cálculos.

Percebe-se também que as características dos grupos de equivalência facilitam outros cálculos, como por exemplo, achar o resíduo de 3^8 módulo 7. Poderíamos fazer da forma tradicional e calcular $3^8 = 6561$ para depois verificar que $6561 = 937 * 7 + 2$ e assim concluir que $3^8 \equiv 2 \pmod{7}$. Mas pode-se usar as classes de equivalência para facilitar nossas contas, veja:

$$
\begin{gathered}
3^8 = 3^4 * 3^4 = 81 * 81 \equiv 4 * 4 \pmod{7} \\
16 \equiv 2 \pmod{7}
\end{gathered}
\tag{2.4}
$$

Anéis de inteiros

Um Anel é um conceito algébrico em que se define um grupo de elementos G que possuem duas operações $\circ$ e $\bullet$, tal que:

- $\forall a, b \in G; a \circ b = c \in G$(operação $\circ$ é fechada em G)
- $(a \circ b) \circ c = a(\circ b \circ c); \forall a, b, c \in G$($\circ$ é associativa)
- $\exists \alpha \in G; a \circ \alpha = a; \forall a \in G$(elemento neutro de $\circ$)
- $\forall a \in G; \exists a^{-1}; a \circ a^{-1} = \alpha$(elemento neutro)(inversa da operação $\circ$)
- $\forall a, b \in G; a \bullet b = c \in G$(operação $\bullet$ é fechada em G)
- $(a \bullet b) \bullet c = a(\bullet b \bullet c); \forall a, b, c \in G$($\bullet$ é associativa)
- $\exists \alpha \in G; a \bullet \alpha = a; \forall a \in G$(elemento neutro de $\bullet$)
- $\forall a \in G; \exists a^{-1}; a \bullet a^{-1} = \alpha$(elemento neutro)(inversa da operação $\bullet$)

Com o conceito acima pode-se verificar que é possível usando a álgebra modular definir um anel de inteiros.

Escolhe-se um grupo de elementos que sejam resíduos distintos de um módulo $m > 0$, este grupo G, tem como elementos $G = 0, 1, 2, 3, ..., m - 1$. Define-se como operação deste grupo a soma $+$ e a multiplicação $*$.

Foi abordado anteriormente que as duas operações são fechadas no grupo e associativas, falta então definir o elemento neutro e provar que todo elemento possui inversa para as operações.

No caso da soma o elemento neutro é o 0, presente no grupo, e no caso da multiplicação tem como elemento neutro o elemento 1, também presente. Precisamos agora analisar as inversas.

Seja $a \in G$ temos que α é sua inversa na soma se, e somente se, $a + \alpha \equiv 0 \pmod{m}$. Com a classe de equivalência podemos substituir 0 por m, assim tendo:

$$a + \alpha \equiv m \pmod{m} \Rightarrow \alpha \equiv m - a \pmod{m}. \tag{2.5}$$

Sabemos que $0 \leq a \leq m - 1$, logo $0 \leq m - a \leq m - 1$, que está dentro de G.

A inversa multiplicativa é um pouco mais complicada, ainda mais por que só é possível que todos os elementos, exceto o 0, de G a possuam caso m seja primo. Para os fins da criptografia m que a partir de agora chamaremos de p será primo. Utilizando métodos como *Pequeno Teorema de Fermat* [2] e o *Algoritmo de Euclides* que será explicado mais a frente é possível encontrar um elemento $a^{-1} \in G$ tal que $a * a^{-1} \equiv 1 \pmod{p}$ para qualquer $a \in G^*$.

Com isso pode-se definir um anel de inteiros e utilizá-lo na criptografia.

2.1.2 Algoritmo de Euclides e o EEA

Como visto anteriormente, o inverso multiplicativo módulo m só é possível para todos os valores $0 < x \leq m - 1$ se o valor m for primo, isto se dá pelo fato de que um número

[2] Não será explicado neste documento

a só possui inversa módulo m se, e somente se, $mdc(a,m) = 1$ [3], ou seja, se a e m forem *primos entre si* [4]. Assim, caso m não fosse primo, ele teria outros divisores além do 1 no grupo, e estes por sua vez não possuem inversa.

A função *phi de Euler*, denotada pelo símbolo $\phi(n)$ diz quantos primos entre si com n existem no intervalo de 1 até $n-1$. No caso dos primos este número sempre é $p-1$ para qualquer primo p. O que nos mostra que todos os elementos do intervalo são *relativamente primos* [5] com p.

Como dá para perceber, o cálculo do mdc é importante para o cálculo da inversa multiplicativa módulo m, para tal cálculo existem alguns algoritmos e o que será contemplado neste documento pois faz parte da implementação do módulo é o algoritmo de Euclides.

O algoritmo de Euclides é baseado no simples fato de que:

$$mdc(a,b) = mdc(b, a-b), \forall a, b, a-b \in Z_+^*. \tag{2.6}$$

A prova da equação 2.6 não será feita neste documento, mas é explicada em [13].

Segue da equação 2.6 que:

$$mdc(a,b) = mdc(a-b,b) = mdc(a-2b,b) = ... = mdc(a-nb,b) \tag{2.7}$$

enquanto $a - nb > 0$. Percebe-se a semelhança com a definição de módulo pois se sendo $a > b$ tem-se que se $a - nb = r$ com $n, r \in Z_+^*$ então é pssível dizer que $r \equiv a \pmod{b}$. Utilizando isto na equação 2.7 tem-se:

$$mdc(a,b) = mdc(b, a \pmod{b}) \tag{2.8}$$

Aplicando recursivamente a equação 2.8 chegamos ao ponto $mdc(g,0)$ onde $g > 0$, mas o mdc de qualquer número com 0 é o próprio número, logo:

$$\begin{gathered} mdc(a,b) = mdc(b, a \pmod{b}) = ... = mdc(g,0) = g \\ mdc(a,b) = g \end{gathered} \tag{2.9}$$

Para exemplificar o algoritmo, será calculado o $mdc(27,21)$.

[3]mdc(a,b) = maior divisor comum entre a e b.

[4]Um número é dito primo entre si de outro quando o único divisor natural comum entre os dois é 1.

[5]outro modo de dizer que um número é primo entre si com outro.

$$
\begin{aligned}
mdc(27,21) = mdc(21, 27 \pmod{21}) = mdc(21,6) \\
mdc(21,6) = mdc(6, 21 \pmod{6}) = mdc(6,3) \\
mdc(6,3) = mdc(3, 6 \pmod{3}) = mdc(3,0) \\
mdc(3,0) = 3 \\
mdc(27,21) = 3
\end{aligned} \tag{2.10}
$$

O próximo passo é como calcular a inversa, para tal existe uma extensão do algoritmo de euclides, denominado de algoritmo de euclides extendido, ou sua sigla em inglês EEA [6].

Este algoritmo tem como base a combinação linear da fórmula do mdc. Ou seja:

$$mdc(a,b) = s * a + t * b \tag{2.11}$$

Onde $s, t \in Z$. O que o algoritmo faz é escrever o resíduo da vez como uma combinação linear de a e b. Assim caso a e b forem primos entre si, tem-se em uma das iterações a equação 2.12

$$1 = s * a + t * b \tag{2.12}$$

Trabalhando com a equação 2.12 tem-se:

$$s * a = 1 - t * b \tag{2.13}$$

O que na visão modular significa a mesma coisa que:

$$1 \equiv t * b \pmod{a} \tag{2.14}$$

Conclui-se da equação 2.14 que t é a inversa de b módulo a, caso $a > b$, e de maneira análoga, se $b > a$, tem-se que s é a inversa de a módulo b.

Com este algoritmo tem-se uma maneira simples de calcular inversas módulo p. O *pseudo-código* [7] deste algoritmo será descrito na seção 4.4.1.

2.1.3 Grupos Cíclicos

O conceito de grupo é nada mais que um conceito mais generalizado de anel. Eis a definição:

"Um Grupo é um conjunto de elementos G, junto com um operador $\circ$ fechado no conjunto G, que tenha as seguintes propriedades:

- $\forall a, b \in G, a \circ b = c \in G$(operação fechada)
- $a \circ (b \circ c) = (a \circ b) \circ c, \forall a, b, c \in G$(operação associativa)

[6] Extended Euclidean Algorithm

[7] algoritmo descrito em uma linguagem próxima da humana.

- $\exists \vartheta \in G / \vartheta \circ a = a \forall a \in G$(elemento neutro)
- $\forall a \in G, \exists a^{-1} \in G / a \circ a^{-1} = \vartheta$(inversa)"

Se além disso, também for verdadeiro que $a \circ b = b \circ a, \forall a, b \in G$, ou seja, a operação é comutativa, dizemos que este Grupo é um *Grupo Abeliano*.

Perceba que um anel, descrito anteriormente, é um grupo em que existe uma segunda operação que segue as mesmas propriedades da primeira.

Quando o operador $\circ$ é uma soma chama-se o grupo de um *grupo aditivo*, caso a operação seja uma multiplicação, chama-se de *grupo multiplicativo*.

Quando um grupo G possui um número finito de elementos, denomina-se este como sendo *Grupo Finito*, a este grupo é associado um valor chamado ordem ou cardinalidade, definido pelo símbolo $|G|$, que nada mais é do que o número de elementos do grupo.

Também existe o conceito de ordem de um elemento $a \in G$, definido pelo símbolo $ord(a)$, que nada mais é do que o menor número positivo k tal que:

$a^k = a \circ a \circ ... \circ a(kvezes) = \vartheta$, onde ϑ é o elemento neutro da operação.

Com esses conceitos é possível definir um *Grupo Cíclico* da seguinte forma:

"Sendo G um grupo finito e α o elemento de maior ordem deste grupo. G é dito um grupo cíclico se $ord(\alpha) = |G|$. E o elemento α é denominado *elemento gerador ou primitivo* de G."

O elemento gerador é assim chamado, pois suas potências geram todos os outros elementos do grupo.

2.1.4 Campo de Galois

Antes de definir o que é um campo de Galois é necessário definir o que é um campo.

"Um campo C é um conjunto de elementos que segue as seguintes propriedades:

- Todos os elementos de C formam um grupo aditivo e possui o elemento neutro aditivo.
- Todos os elementos de C formam um grupo multiplicativo e possui o elemento neutro multiplicativo.
- Quando misturamos as operações é mantida a propriedade distributiva, ou seja, $a * (b + c) = a * b + a * c, \forall a, b, c \in C$"

Se o número de elementos de um campo é finito, definimos este como sendo um Campo Finito ou Campo de Galois, que pode ser representado pelo símbolo $GF()$, mas este só existe se o número de elementos do campo, denominado ordem ou cardinalidade do campo for uma potência de primo. Definimos um campo de Galois de ordem p com o símbolo $GF(p)$.

2.2 Criptografia

Para começar a aprofundar os conceitos dentro do estudo criptográfico será definido o que é criptografia.

Desde o início da comunicação humana existe a necessidade de que algumas mensagens a serem transmitidas, via fala, escrita ou interfaces eletrônicas, tenham um certo sigilo, ou seja, algumas pessoas não podem ter acesso à informação da mensagem, como por exemplo, uma carta com instruções para um batalhão do exército em uma guerra, que não pode ser interceptada pelo inimigo. A proposta para este intento é transmitir esta em um canal de comunicação seguro, onde a premissa é que durante todo o percurso, nenhuma informação seja interceptada até chegar ao seu destino, ou na falta de tal canal confiável, seria necessário disfarçar a mensagem de um jeito que ninguém a não ser o destinatário consegui-se ver através deste disfarce. Criar este disfarce é exatamente a função da criptografia.

Para abordar os próximos conceitos será usado uma nomenclatura tradicional como em [18], onde o remetente da mensagem será Alice e o destinatário será Bob, além do atacante, ou seja, uma pessoa que não pode saber a informação da mensagem, mas tenta interceptá-la, este chamaremos de Oscar.

Com o uso da criptografia tem-se que Alice antes de enviar a mensagem por um canal não necessariamente seguro irá cifrar, ou seja, transformar a mensagem em um texto codificado, que não faça sentido, ou pelo menos, esconda o sentido original da mensagem. Logo depois envia o texto cifrado para Bob que o decifra de volta na mensagem original. Assim, mesmo que Oscar intercepte a mensagem ele não saberá o conteúdo, pois esta estará cifrada e só Bob poderá decifrar. Como mostrado na figura 2.1.

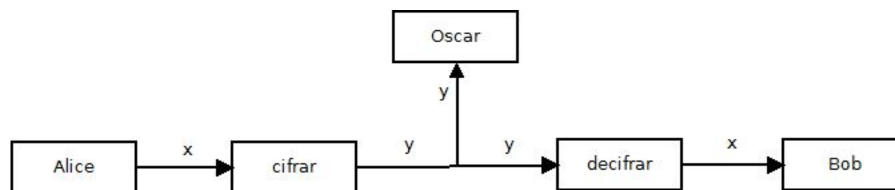

Figura 2.1: Digrama de Comunicação

Porém, do modo em que se encontra a figura 2.1, para que seja possível que Alice cifre a mensagem de modo que Oscar não consiga decifrar, o algoritmo para decifrar o código deve ser desconhecido por Oscar. Mas o ideal é que o segredo não seja o algoritmo de cifragem e decifragem, principalmente por que se este vaza e Oscar o descobre, Alice e Bob teriam que criar um novo algoritmo. Assim, algoritmos criptográficos cifram e decifram textos utilizando além do texto uma chave como entrada para funcionar, deste modo o único segredo necessário é a chave, que é muito mais fácil de ser substituída caso esta seja descoberta por Oscar. Para que tanto Alice quanto Bob possuam a chave, estes devem combiná-la em um canal de confiança.

Ter a necessidade de um canal de confiança para combinar chaves parece contraditório, pois a cifragem foi criada para que a comunicação fosse feita sem a necessidade de um canal de confiança. Entretanto fica menos contraditório a partir do momento que este canal só será necessário uma vez para o compartilhamento da chave, depois disso é possível utilizar o canal não confiável várias vezes para transmitir a mensagem cifrada. Com isso o nosso diagrama da figura 2.1 terá um acréscimo e ficará como na figura 2.2.

Para ilustrar melhor o que está sendo mostrado na Figura 2.2 será utilizado um exemplo histórico de um dos primeiros protocolos criptográficos criados, este protocolo utilizava a hoje conhecida como *Cifra de César*, e como pode-se deduzir pelo nome, era usado em

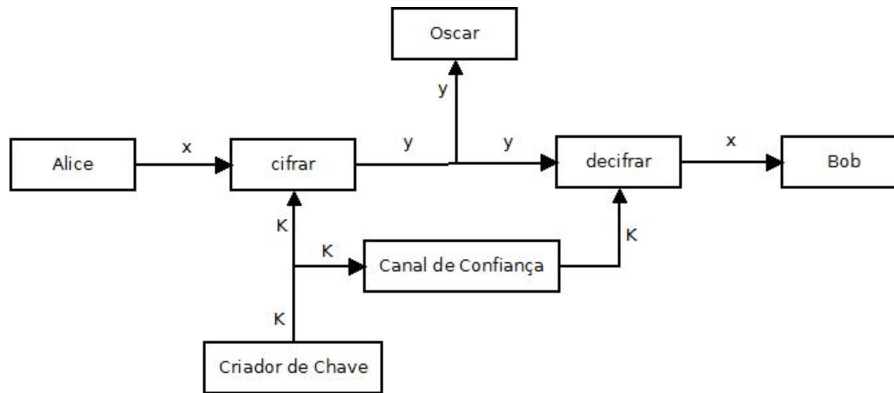

Figura 2.2: Cifragem Simétrica

Roma pelo então imperador Júlio César para se comunicar com seus generais. Neste protocolo nas reuniões fechadas entre o imperador e os generais eram distribuídas as chaves para cada general para que quando tivessem que se comunicar de forma secreta e estivessem longe, mandariam suas cartas cifradas com a tal chave para que o mensageiro a entregasse ao imperador. Uma vez que só o imperador além do próprio general possuía a chave só ele poderia decifrar a mensagem, assim não corriam o risco de que o mensageiro ou qualquer outro soubesse das informações secretas. Fazendo a analogia com a figura 2.2 é possível dizer que Alice é o general, Bob é o imperador, o canal não confiável é o mensageiro, Oscar pode ser o próprio ou alguém que roube o mensageiro, e o canal de confiança é a reunião fechada para o imperador e os generais.

2.2.1 Sistema Criptográfico

Em [18], sistema criptográfico é definido como sendo uma 5-tupla (P, C, K, E, D), onde P é o conjunto de todas as possíveis mensagens, C o conjunto de todas as possíveis cifras, K é o conjunto de possíveis chaves, E é o conjunto de todas os algoritmos para cifrar um texto, onde para cada $e \in E$ existe um $d \in D$, sendo este o conjunto de todos os algoritmos que decifram um texto.

Aproveitando o exemplo utilizado anteriormente será ilustrado o conceito exposto com o sistema da *Cifra de César*. Neste sistema a chave é um número que representa o deslocamento que a letra terá, por exemplo, se a chave for 3, a letra A do texto original será substituída por D. Para um alfabeto americano apenas com as letras maiúsculas temos 26 letras, logo, primeiro associa-se um número para cada letra do alfabeto, desse jeito:

Tabela 2.1: Tabela de Cifra de César

A	B	C	D	E	F	G	H	I	J	K	L	M
0	1	2	3	4	5	6	7	8	9	10	11	12

N	O	P	Q	R	S	T	U	V	W	X	Y	Z
13	14	15	16	17	18	19	20	21	22	23	24	25

Em seguida é feita a seguinte substituição para cada letra do texto.

$$c = (m + k) \bmod 26 \tag{2.15}$$

Utilizando a mesma associação de número com letra visto acima transforma-se o 'c' resultante em letra. Por exemplo na palavra EXEMPLO com a chave 3, tem-se:

$$\begin{gathered}(5+3) \bmod 26 = 8(H)\\(23+3) \bmod 26 = 0(A)\\(5+3) \bmod 26 = 8(H)\\(12+3) \bmod 26 = 15(P)\\(15+3) \bmod 26 = 18(S)\\(11+3) \bmod 26 = 14(O)\\(14+3) \bmod 26 = 17(R)\end{gathered} \tag{2.16}$$

Assim se tem o texto cifrado HAHPSOR.

Para decifrar é só inverter a conta assim:

$$m = (c - k) \bmod 26 \tag{2.17}$$

Como pode ser visto, este não é o melhor algoritmo, dado que o número de chaves com resultados diferenciados é o tamanho do alfabeto, em outras palavras, no exemplo exposto, Oscar necessitaria de no máximo 26 tentativas para acertar a chave. Assim é possível dizer que um dos requisitos de segurança de um sistema criptográfico é a necessidade de um espaço grande de chaves para tornar inviável a descoberta da chave pela busca exaustiva.

2.2.2 Serviços Criptográficos

Serviços criptográficos são as aplicações dos diferentes métodos criptográficos.

Até agora foi abordado a cifragem, que é uma aplicação na qual uma mensagem é cifrada e quando chega ao destino é decifrada e é necessário que Alice e Bob compartilhem uma mesma chave, a esta chama-se de cifragem simétrica.

Neste documento é usado a classificação de serviços como em [16].

São estas:

- Autorização
- Cifragem Simétrica/Assimétrica
- Autenticação Subjetiva/Objetiva
- Certificação

Dentro do objetivo deste documento o importante é a autenticação objetiva que se dá com o uso da assinatura digital, logo esta será descrita em mais detalhes, enquanto as outras terão uma explicação mais sucinta.

Autorização

Este serviço é principalmente usada para sistemas de *login*, sua função é restringir acessos a dados ou recursos, permitindo acesso apenas com o par certo de identificador e senha.

Neste sistema um usuário entra com seu identificador e senha para se cadastrar ou tentar acessar os recursos obtidos no cadastro anterior. Ao entrar com seus dados a senha é processada com um método não inversível, ou seja, não há um algoritmo computacionalmente trivial de reverter a codificação. A premissa é que o algoritmo de processamento gere um único texto codificado para a mesma mensagem, que seja resistente a colisão e resistente a inversão, em outras palavras se existe uma senha x e outra x', tal que $x \neq x'$ então $h(x) \neq h(x')$, onde $h(_)$ é a função de processamento, e que achar um $h^{-1}(_)$ é computacionalmente inviável.

A senha cifrada é comparada com a do banco de dados, caso sejam iguais o usuário tem acesso aos recursos. Como mostrado na figura 2.3:

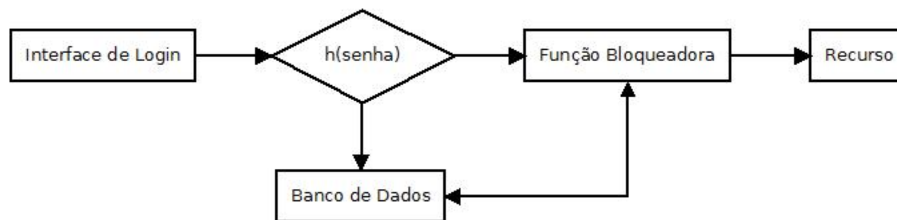

Figura 2.3: Autorização

Cifragem Simétrica

A cifragem simétrica já foi abordada no início da seção 2.2 e seu diagrama é demonstrado na figura 2.2, é utilizado para transmissão de mensagem com sigilo.

Autenticação Subjetiva

A autenticação subjetiva é utilizada para garantir a integridade da mensagem, ou seja, que esta não foi alterada no percurso da entrega, e para confirmar a origem do conteúdo. Assim como na cifragem simétrica Alice e Bob compartilham uma mesma senha, quando Alice quer transmitir uma mensagem sem necessidade de sigilo, mas Bob quer certificar-se que a mensagem entregue é a mesma que foi enviada por Alice, e que foi realmente Alice que a enviou. Assim, Alice ao mandar a mensagem cria um autenticador s tal que $s = h(m\|k)$, onde $m\|k$ significa a concatenação de m(mensagem) e k(chave), e $h(_)$ é uma função de resumo criptográfico [8], a qual será discutido mais a frente. Depois é enviado a Bob $m\|s$.

Quando Bob recebe a mensagem ele faz novamente o resumo criptográfico de $m\|k$ e compara com o s, se for idêntico a mensagem esta autenticada. Como mostrado na figura 2.4.

[8] Também conhecido com função de hash.

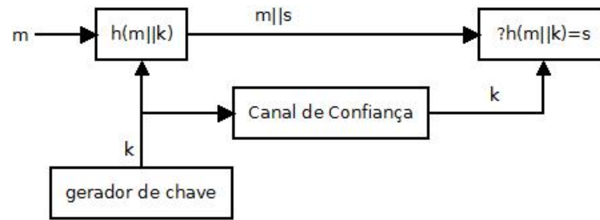

Figura 2.4: Autenticação Subjetiva

Cifragem Assimétrica

A cifragem assimétrica é utilizada para troca de mensagem em sigilo sem a necessidade de compartilhamento de chave secreta ou canal de confiança. Ela funciona com a idéia de par de chaves, a qual será detalhada quando for abordada a autenticação objetiva. Mas a idéia neste caso é que Bob gerará um par de chaves onde uma cifrará mensagens e a outra decifrará as mensagens cifradas pela primeira. A chave que cifra será definida como chave pública e a que decifra será a chave privada. A chave pública é compartilhada com Alice e qualquer outra pessoa na rede que queira se comunicar com Bob. Alice usa a chave pública de Bob para cifrar a mensagem e envia para Bob que recebe e usa a sua chave privada para decifrar. Assim mesmo que Oscar intercepte a mensagem cifrada e tenha posse da chave pública de Bob ele não poderá decifrar, pois necessitaria da chave privada.

Como podemos ver, a premissa para que de certo é de que a chave privada esteja única e exclusivamente com Bob. Este serviço é demonstrado na figura 2.5.

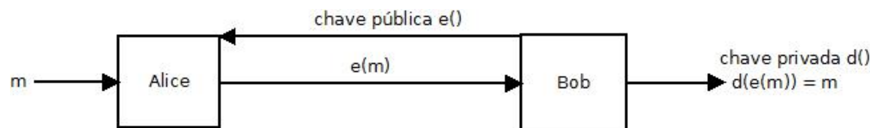

Figura 2.5: Cifragem Assimétrica

Autenticação Objetiva

Como abordado no início da seção, este serviço é o mais importante no estudo deste projeto, mas ela será aprofundada quando abordar a sua aplicação em assinaturas digitais. Nesta seção será explicado sucintamente o seu funcionamento.

Assim como na autenticação subjetiva sua finalidade é a de garantir a integridade da mensagem durante seu trajeto no canal e a identificação do autor da mensagem, mas assim como na cifragem assimétrica é utilizado ao invés de uma chave secreta compartilhada entre Alice e Bob, um par de chaves pública e privada de Alice e a chave pública de Alice é mandada para Bob sem a necessidade de proteger contra vazamento para Oscar. Porém a premissa de propriedade única da chave privada de Alice estar com a mesma continua valendo. Assim, Alice manda uma mensagem concatenada com seu autenticador, gerado pelo $d(h(m))$, onde h é uma função de hash e $d(_)$ é uma função que se utiliza da chave privada de Alice. Alice envia a concatenação para Bob que calcula $e(s)$, sendo $e(_)$ é uma função que se utiliza da chave pública de Alice e s é o autenticador de Alice, e compara com o $h(m)$. Caso sejam iguais, é comprovado a autoria da mensagem como sendo de Alice e que a mensagem está integra. O diagrama desta premissa é mostrado na figura 2.6.

Em algumas assinaturas, inclusive a do ECDSA não é o $h(m)$ que é recalculado pela função $e(_)$ e sim outro valor que é enviado como parte da assinatura, isto será abordado no capítulo referente ao protocolo ECDSA.

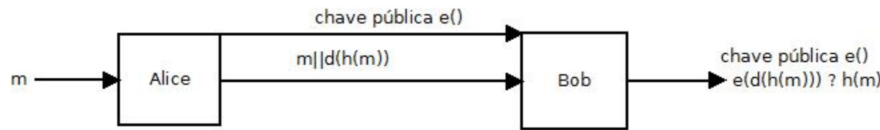

Figura 2.6: Autenticação Objetiva

Certificação

A Certificação é uma aplicação recursiva da autenticação objetiva. onde em uma mensagem de Alice para Bob além do autenticador calculado pela chave privada de Alice também a o autenticador de uma terceira pessoa de confiança ou TTP [9], ou seja, um terceiro personagem a qual, ambos os agentes confiam, este esta lá para confirmar que o par de chaves dito ser da Alice é realmente da Alice. Porém o autenticador deste TTP pode ser posto a prova por outro certificado. Deste jeito, temos uma recursão que só acabará quando o TTP da última camada for uma entidade de confiabilidade inquestionável por todos os envolvidos.

2.2.3 Assinatura Digital

Assinatura Digital é uma aplicação dentro da criptografia que procura levar para o meio digital a autenticação objetiva física que temos na assinatura. Dentro do meio jurídico, uma assinatura representa a concordância do dono da mesma com o contexto em que ela é imposta. Dentro de uma perícia, tal assinatura é comparada com assinaturas anteriores da mesma pessoa, onde são vistas semelhanças.

Nas perícias mais simples apenas uma comparação visual é feita, como por exemplo ao pagar algo com cheque e o atendente lhe pede sua identidade para comparar sua assinatura com a da mesma. Em outras perícias mais detalhadas é visto outras características detectáveis na forma e quantidade de tinta no papel, como pressão aplicada na hora, velocidade de escrita. Detalhes que para o assinante é algo muito mais inconsciente do que técnico e por isso difícil de se copiar.

Dentro de uma autenticação objetiva temos algumas propriedades semiológicas que a assinatura de punho possui.

- Inforjabilidade
- Inviolabilidade
- Irrecuperabilidade
- Irrefutabilidade ou Irretratabilidade

[9] Third trusted person.

Logo, na idéia de fazer que seja válida a aplicação da assinatura digital como autenticação objetiva devemos apresentar maneiras de fazer com que estas propriedades sejam alcançadas. A seguir abordaremos cada propriedade, mostrando como ela é vista no meio físico e no meio digital.

As explicações dadas abaixo só podem ser provadas dada como premissas os seguintes items em um sistema de assinatura digital.

- A posse da chave privada de um agente é única e exclusivamente deste.
- Existe a confiança de Bob que a chave pública que ele possui é realmente de Alice, para qualquer Alice. Normalmente este é obtido com a certificação digital.
- É presumido que ao assinar um documento com sua chave privada, Alice manifesta a vontade no conteúdo do documento.

Inforjabilidade

Inforjabilidade é a função que dá confiança ao verificador sobre a identidade do assinante ou da falsificação da assinatura. Na assinatura física, como dito anteriormente, é feita com perícia na forma da assinatura e traços específicos de pressão aplicada na folha e grossura do risco. A questão é como tornar uma assinatura única no meio digital.

No mundo digital, uma assinatura é uma seqüencia de símbolos representado por outra camada de símbolos denominados *bits*, porém todos são abstrações e facilmente copiados e manipulados de maneiras diferentes e imperceptíveis para o usuário. E é para tentar fazer com que uma assinatura seja unicamente feita pelo dono da assinatura que a criptografia entra, mais precisamente o serviço de autenticação objetiva.

Como visto na seção 2.2.2 este serviço necessita de um par de chaves pública e privada e que como está descrito nas premissas, a chave privada seja exclusiva do dono do par. Com isso, para assinar um documento digitalmente, Alice utiliza a sua chave privada para cifrar o resumo criptográfico da mensagem e envia a concatenação da mensagem com o resultado da cifragem. A chave pública que está com Bob decifra o texto gerado pela chave privada tendo de volta o resumo criptográfico da mensagem. Assim é certo que apenas a chave privada de Alice pode ter cifrado o resumo criptográfico da mensagem, e dada a premissa de posse única da chave privada, podemos dizer que foi realmente Alice que assinou. É visto na figura 2.7 um resumo.

Este padrão de assinatura digital descrito acima é o RSA [10], atualmente existe o padrão DSA [11] e suas variantes, as quais o método de verificação da assinatura não se baseia em decifrar o autenticador para chegar ao resumo criptográfico da mensagem, mas sim em encontrar um valor intermediário do processo feito com a chave privada, utilizando a chave pública. Esta será mais explicada quando for abordado o ECDSA [12].

[10] Recebe este nome graças aos seus três criadores, professores do MIT e fundadores da RSA Data Security,Inc., Ronald Rivest, Adi Shamir, Leonard Adleman.

[11] Digital Signature Algorithm, algoritmo de assinatura digital desenvolvido pelo NIST(National Institute of Standard and Technology).

[12] Variação da DSA que se utiliza de pontos em curvas elípticas.

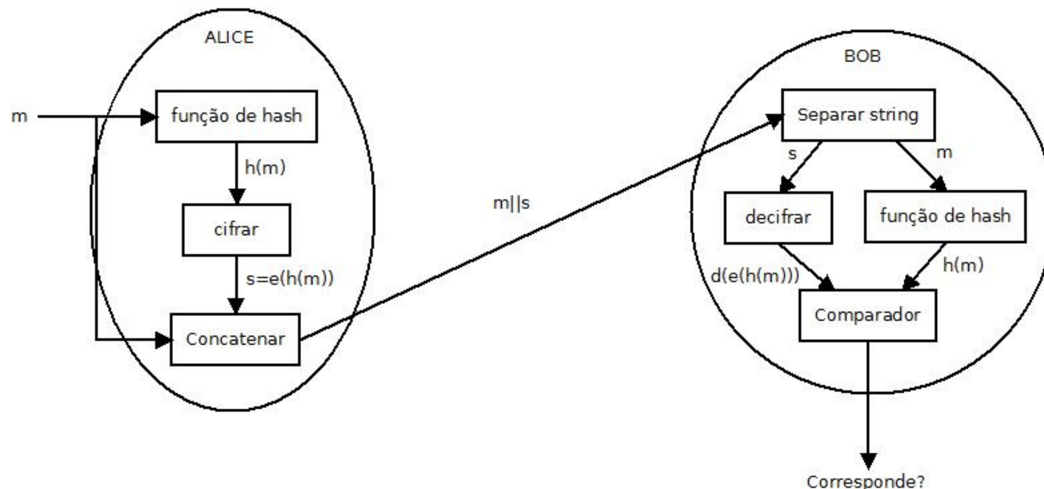

Figura 2.7: Assinatura Digital RSA

Inviolabilidade

A inviolabilidade dá ao verificador a segurança de que o documento não foi alterado após ser assinado. Para tal, a assinatura física têm na própria natureza física a resposta para esta função, pois qualquer alteração no meio físico deixa rastros, seja um desgaste na folha ao tentar apagar algo, a diferença de quão fresca a tinta está de algo que foi inserido posteriormente, rasuras, entre outras que tornam as alterações perceptíveis para um perito. Mas como explicado anteriormente, o documento digital é apenas um conjunto de símbolos que pode ser replicado e alterado, sem traços visíveis nas camadas de interface.

Porém como mostrado no diagrama da figura 2.7 a mensagem faz parte do processo de assinatura, logo qualquer alteração na mensagem após a assinatura faria com que no momento final de autenticação em que o resumo criptográfico da mensagem tem que ser igual ao texto decifrado, daria errado. Pois se uma função de resumo criptográfico, como será visto mais a frente, tem como premissa ser resistente a colisões, ou seja, dado m e m' mensagens em que $m \neq m'$, então $h(m) \neq h(m')$. De uma forma mais formal:

$$\begin{gathered} ALICE \\ s = e(h(m)) \\ BOB \\ m' \| s \\ d(e(h(m))) = h(m) \\ h(m) \neq h(m') \end{gathered} \tag{2.18}$$

Irrecuperabilidade

A irrecuperabilidade é a função que dá ao verificador a certeza que é inviável o reuso de assinaturas em outros documentos. No meio físico é trivial visto que a idéia de recortar

uma assinatura e colar em outro documento é facilmente perceptível, além do fato de que a interface visual do documento é o próprio meio físico. Ou seja, uma pessoa que tenha o cuidado de ler o documento, verá claramente o seu conteúdo sem ilusões. No meio digital temos o problema da interpretação dos dados ser feita em várias camadas. Afinal, uma mesma seqüência de *bits* pode representar um texto se interpretado por um editor de texto, como também pode ser uma música, ou imagem se for interpretado por um player ou um editor de imagens. Para evitar este problema, é esperado que na mensagens existam metadados sobre onde e como o arquivo deva ser interpretado.

Para melhor exemplificar, imaginem a seguinte situação:

Alice assina um arquivo enviado por Bob, que ao ser interpretado como .doc apresenta uma proposta de venda de seu carro para Bob no valor de 30mil reais. Porém este mesmo arquivo pode ser interpretado como um *.mp3* onde se encontra a fala de uma pessoa falando que a compra do carro se daria por 10 reais. Apesar deste exemplo ser esdrúxulo, é de fácil entendimento que há meios de convencer alguém de estar assinando algo que na verdade está disfarçado.

Irrefutabilidade

A irrefutabilidade é a função que garante ao verificador a inviabilidade de negar a autoria da assinatura perante um terceiro pessoa de confiança(TTP) que atua como juiz. No mundo físico a perícia que convencer o juiz de que sua análise comprova a veracidade ou a falsidade da assinatura tem como provado sua tese. Apesar de que na lei brasileira existe o mau uso da fé pública, ou seja, aquele que está certo até que se prove o contrário. Na assinatura digital a tentativa é de fazer a mesma regra visto que o único jeito de forjar uma assinatura é na quebra da primeira premissa, em que haja o vazamento da chave privada. Assim, aquele que quiser provar que não foi ele deve provar que a chave privada vazou e a quem chegou.

É definido então que não há um meio satisfatório de prover irrefutabilidade no meio digital.

2.2.4 Função de Resumo Criptográfico

Uma função de hash é um algoritmo que recebe como entrada uma seqüencia de dados de tamanho qualquer, e gera como saída uma outra seqüência de dados com tamanho fixo. por exemplo um texto de 12 Kb pode ser submetido a uma função de hash, como por exemplo o SHA-1 e com isso temos na saída um texto de 160 b. Pelo fato de que o domínio da função de hash D pode ter um tamanho infinito, mas o contra-domínio tem um tamanho fixo, temos na maioria dos casos que para um $x \in D$ e um correspondente $y \in CD$, em que CD é o contra-domínio, tal que $h(x) = y$, sendo $h(_)$ uma função de resumo criptográfico, $|x| > |y|$; assim, define-se uma função de resumo criptográfico como uma função de compressão, também é dito que uma mensagem submetida a uma função de resumo criptográfico é digerida.

Matematicamente, uma função de hash é definida em[18] como sendo uma 4-tupla (X, Y, K, H), onde X é o conjunto de todas as possíveis mensagens que podem ser mandadas para entrada, tal que o $|x|$ onde $x \in X$ pode ser infinito, Y é o conjunto de todas as saídas possíveis da função de resumo criptográfico, onde o $|y|$, tal que $y \in Y$, é fixo e

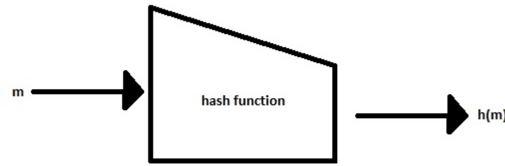

Figura 2.8: Função de Hash

finito, K é um conjunto finito de possíveis chaves para o resumo criptográfico, e H é o conjunto de possíveis algoritmos que definem uma função $h_k(x) : X \rightarrow Y$.

As aplicações de uma função de hash são várias, principalmente para teste de integridade de documentos, pois podemos afirmar que uma função de hash cria uma ”impressão digital” do texto que é submetido a ela. Assim caso Alice queira enviar um texto e sua única preocupação é a integridade deste, ela pode enviar junto com o texto o hash deste, utilizando uma função conhecida apenas por Alice e Bob, assim, Bob pode utilizar o mesmo hash na mensagem é comparar com a mensagem digerida que veio de Alice. Caso Oscar altere o texto ou o texto digerido os valores testados por Bob não vão bater. Pois se $x \neq x'$, logo $h(x) \neq h(x')$.

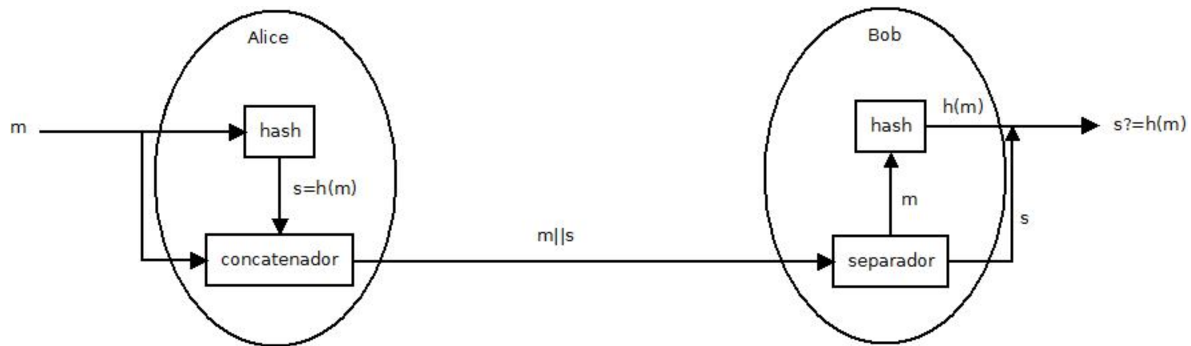

Figura 2.9: Função de Hash - Integridade

Como vimos na Seção 2.2.3, a função de resumo criptográfico é utilizada na assinatura digital, ao fazer a cifragem com a chave privada ao invés de fazer $e_k(m)$ faz-se $e_k(h(m))$, pois deixa o sistema mais resistente a alterações, pós assinatura, no documento, ou seja, reforça a inviolabilidade, vista também na Seção 2.2.3.

Para averiguar a segurança do sigilo e integridade necessita-se de algumas provas de resistência. São essas:

- Também chamado de inversão ou primeira pré-imagem. Dado uma função de resumo criptográfico $h : X \rightarrow Y$ e um elemento $y \in Y$. Encontrar um $x \in X$ tal que $h(x) = y$.
- Também chamado de segunda pré-imagem. Dado uma função de resumo criptográfico $h : X \rightarrow Y$ e um elemento $x \in X$. Encontrar um $x' \in X$ tal que $x' \neq x$ e $h(x') = h(x)$.
- Também chamado de teste de Colisão. Dado uma função de resumo criptográfico $h : X \rightarrow Y$. Encontrar $x, x' \in X$ tal que $h(x) = h(x')$.

No primeiro item tem-se uma função de resumo criptográfico e um y, se quer um método eficiente para descobrir um possível x que produza aquele valor y de resumo criptográfico. Caso exista, a pessoa que o use poderá forjar um documento que gere um resumo criptográfico desejável para iludir o outro agente da rede de que a mensagem que ele deveria receber era ela mesma. Exemplo, Alice envia uma mensagem concatenada com seu respectivo resumo criptográfico s. Oscar poderia interceptar a mensagem e a partir do resumo criptográfico s achar qualquer outra mensagem que gera esse resumo criptográfico e substituí-la, atrapalhando a comunicação entre Alice e Bob. Caso não seja possível encontrar um modo eficiente de fazê-lo é dito que ele é resistente à inversão ou primeira pré-imagem.

No segundo item tem-se uma função de resumo criptográfico e um elemento $x \in X$, se quer um método eficiente para achar um x' que submetido ao mesmo resumo criptográfico produza o mesmo valor, ou seja, $h(x') = h(x)$. Veja que, diferente da 1ª prova, para este é necessário ter o x, enquanto a outra só o seu resumo criptográfico. Caso não seja possível, é dito que a função é resistente à segunda pré-imagem.

No terceiro item tem-se a função de resumo criptográfico apenas, se quer um método eficiente de encontrar duas mensagens distintas x e x' que possuam o mesmo resumo criptográfico. Caso seja possível, Bob poderia encontrar estes dois elementos em que um teria, suponhamos, um contrato simples que Alice assinaria sem preocupações, mas a outra mensagem contém um contrato que prejudica de alguma forma Alice. Pelo fato dos dois produzirem mesmo resumo criptográfico o autenticador de uma mensagem assinada por Alice seria idêntico para as duas mensagens, assim, Bob poderia depois de ter o autenticador de Alice apenas trocar as mensagens. Este ataque é conhecido na criptografia como ataque de aniversário. A figura 2.10 ilustra o que exemplificamos aqui. Caso não seja possível tal método eficiente a função é definida resistente à colisão.

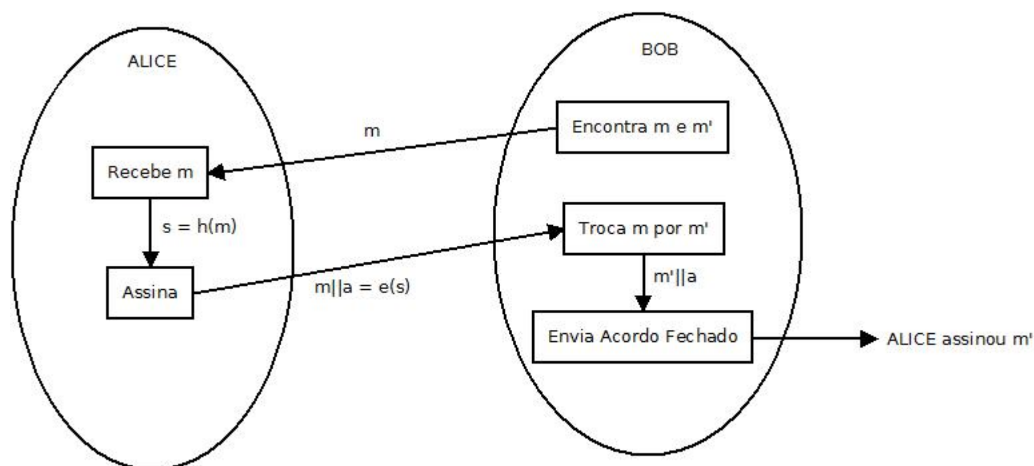

Figura 2.10: Função de Resumo criptográfico - Ataque de Aniversário

Uma função para ser um resumo criptográfico deve primeiramente ser uma função não inversível, ou seja, não há método conhecido para inverter a função. Para que seja considerada segura e eficiente deve ter resistência aos três items acima listados.

Também existe o conjunto das funções de resumo criptográfico sem chaves, a definição é a mesma, e seus testes de resistência também, na verdade pensar num resumo criptográfico sem uma chave é como pensar em um resumo criptográfico comum onde o conjunto de

chaves possíveis tem tamanho 1. Ou seja, sempre que o algoritmo é executado a mesma chave é utilizada, assim, não mais este elemento é uma chave e sim uma constante do programa.

Por fim, existe a classe de funções de resumo criptográfico denominadas função iterada de resumo criptográfico, está que, atualmente, é mais usada do que as outras. Afinal esta técnica é a que realmente possibilita que o conjunto do domínio da função seja infinito. Para melhor entendimento e também para posicionar as próximas definições no contexto do projeto, as mensagens serão trabalhadas como sendo cadeias de 0 e 1, ou seja, cadeias de *bits*.

Suponha uma função de resumo criptográfico definida como uma compressão do tipo $c : \{0,1\}^{m+t} \rightarrow \{0,1\}^m$ onde $t \geq 1$. Constrói-se uma função iterada de resumo criptográfico aplicando de maneira sucessiva esta função. Como a qualquer função de resumo criptográfico é definida por $h : X \rightarrow Y$, deve-se começar por um $x \in X$ e deve-se ter como resultado final um $y \in Y$, onde |y| é fixo e o |x| pode ser qualquer um. Os passos são:

- Preprocesse x com um algoritmo e obtenha z, tal que $|z| \equiv 0 mod(t)$.
- Defina $z = z_1 \| z_2 \| ... \| z_r$, onde $|z_i| = t \forall 1 \leq i \leq r$.
- Defina um y_0 tal que $|y_0| = m$.
- Faça $y_i = c(y_{i-1} \| z_i)$ para i indo de 1 até r.
- Defina $y = y_r$.

O preprocessamento mais simples para a primeira etapa é conhecido como $pad(x)$, onde a função concatena x com uma seqüência de zeros até que se consiga o tamanho necessário, algo similar pode ser feito com y, caso o tamanho desejado como saída da função iterada de resumo criptográfico seja maior que m.

Este procedimento todo é também conhecido como paradigma de Merkle-Damgord.

A ilustração do processo está na figura 2.11.

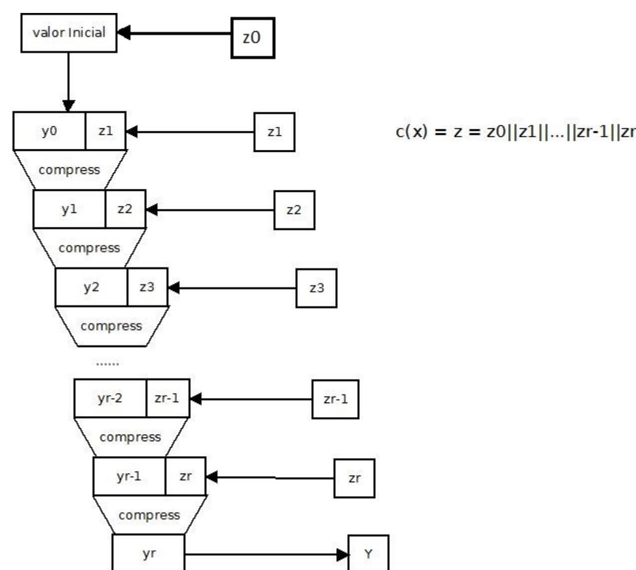

Figura 2.11: Função de Resumo criptográfico - Resumo criptográfico Iterado

2.2.5 SHA(Secure Hash Algorithm)

Agora que foi definido uma função de resumo criptográfico, é possível analisar um dos algoritmos mais utilizados de sua criação até hoje em aplicações criptográficas. O *Secure Hash Algorithm* ou simplesmente SHA é o algoritmo criado pela NSA [13] em 1993 e definido como padrão pelo governo norte-americano. Sua primeira versão publicada foi posteriormente denominada de SHA-0, pois foram criadas novas versões deste algoritmo. Atualmente, as versões são classificadas em três famílias: SHA-0, que foi a primeira versão; SHA-1, que foi desenvolvida 2 anos depois e foi muito utilizada; e o SHA-2 que engloba as versões SHA-224, SHA-256, SHA-384 e SHA-512.

As famílias SHA-0 e SHA-1 já tiveram sua vulnerabilidade comprovada, pois ataques aplicados com sucesso sobre os dois algoritmos foram publicados, mas nenhum ataque foi publicado sobre os algoritmos da família SHA-2. Apesar disso, muitos pesquisadores ainda estão atentos as possibilidades de ataques nesta última família, pois a base do algoritmo das três é o mesmo. Nesta seção será descrito o SHA-1, pois, como abordado, foi um dos mais utilizados, e de melhor compreensão do que a família SHA-2. Porém, durante o capítulo de documentação dos módulos será descrito o SHA-256, utilizado na implementação do ECDSA a qual este documento se refere.

O SHA-1 é descrito em [18] como um algoritmo construído por operações orientadas a palavras em seqüencias de *bits*, onde cada palavra possui 32 *bits* e o resultado final é uma cadeia de 160bits. As operações usadas no algoritmo são as seguintes:

- $X \wedge Y$ operação "e" de X e Y
- $X \vee Y$ operação "ou" de X e Y
- $X \oplus Y$ operação "ou exclusivo" de X e Y
- $\neg X$ operação de "negação" de X
- $X + Y$ operação de "adição inteira" entre X e Y módulo 2^{32}
- $ROTL^s(X)$ operação de "deslocamento de bits à esquerda" de X em s posições.

Primeiro define-se o *padding* necessário para o SHA-1, esta função altera a mensagem tal que o número de *bits* da saída seja fixo e adequado para o resto do algoritmo. Assim o algoritmo $PAD(x)$ consiste de:

- observação: $|x| \leq 2^{64} - 1$.
- $d \leftarrow (447 - |x|) \bmod 512$.
- $l \leftarrow$ a representação binária de $|x|$, onde $|l| = 64$.
- $y \leftarrow x \| 1 \| 0^d \| l$

Como se observa, o $PAD(x)$ retorna um y tal que $|y| \equiv 0 \bmod 512$. Sabe-se que o tamanho da representação binária de x é l e dado que o maior valor de x é $2^{64} - 1$, logo o $|l| < 64$, então adiciona-se 0's a representação binária de x até que $|l| = 64$. Adiciona-se um *bit* 1 ao x e concatenamos com 0's até que seu tamanho seja congruente a 448 mod 512.

[13] Nacional Security Agency, agencia de segurança dos Estados Unidos

Finalmente, junta-se o que obtivemos para ter uma seqüencia de *bits* de tamanho divisível por 512. Assim é possível escrevê-lo como uma concatenação de blocos de tamanho 512.

$y = M_1 \| M_2 \| ... \| M_n$.

Define-se algumas funções denominadas $f_0, ..., f_{79}$ e que possuem as seguintes formações:

- $f_t(B, C, D) = (B \wedge C) \vee ((\neg B) \wedge D)$ se $0 \leq t \leq 19$
- $f_t(B, C, D) = B \oplus C \oplus D$ se $20 \leq t \leq 39$
- $f_t(B, C, D) = (B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$ se $40 \leq t \leq 59$
- $f_t(B, C, D) = B \oplus C \oplus D$ se $60 \leq t \leq 79$

Onde B,C e D são as entradas de 32bits e o resultado de f_t é uma palavra de 32bits. Então é possível definir o algoritmo do SHA-1 deste jeito mostrado no algoritmo 1. Para este algoritmo ainda define-se as constantes $K_0, ..., K_{79}$ como sendo:

- $5A827999$ se $0 \leq t \leq 19$
- $6ED9EBA1$ se $20 \leq t \leq 39$
- $8F1BBCDC$ se $40 \leq t \leq 59$
- $CA62C1D6$ se $60 \leq t \leq 79$

Como é possível ver pelo algoritmo do SHA-1, esta é uma versão de função de resumo criptográfico de iteração, onde há uma função $PAD(_)$ inicial que cria de x uma seqüencia de *bits*, cujo tamanho é múltiplo de 512, e a função de compressão recebe uma seqüencia de $160 + 512$ bits e retorna como saída uma seqüencia de 160bits.

SHA-1 é um dos vários exemplos que temos de funções de iteração de resumo criptográfico, o primeiro deles foi o MD4 proposto em 1990 por Rivest, este o modificou tempos depois e criou o MD5 em 1992. Como já dito, o SHA-1 é uma variante do SHA-0, proposto em 1993, o SHA-1 foi proposto em 1995, sendo uma pequena variação do anterior, onde a grande diferença era a omissão da rotação de 1bit na construção dos W_i no SHA-0.

Essas várias implementações foram feitas devido a massiva incorporação destes algoritmos na área de segurança, por isso também, os pesquisadores começaram a fazer provas de segurança destes códigos. No meio dos anos 90, foi demonstrado que os algoritmos MD4 e MD5 não eram resistentes a colisão e em 1998 foi mostrado que o SHA-0 possuía uma fraqueza que possibilitava que colisões fossem encontradas em aproximadamente 2^{61} passos, que é melhor, por exemplo, que as probabilidades do ataque de aniversário, já descritos nas seções anteriores que precisa de aproximadamente 2^{80} passos.

Por fim, em março de 2005, os pesquisadores Wang, Yin e Yu conseguiram reduzir em 58 iterações o trabalho de achar colisões no SHA-1, e também foi levantado a hipótese de conseguir obter colisões com menos de 2^{69} passos. Assim, nem mesmo o SHA-1 é considerado robusto. Porém a família SHA-2 ainda permanece sem contra-provas de sua robustez. Uma das diferenças desta família é o aumento do tamanho da saída da função de compressão, estas que são mostradas no próprio nome do algoritmo, por exemplo, o SHA-512 possui uma saída da compressão com tamanho de 512bits.

Algorithm 1 Algoritmo de SHA-1

external$PAD(_)$
global$K_0, ..., K_{79}$

1: $y \leftarrow PAD(x)$
2: denote $y = M_1 \| M_2 \| ... \| M_n$ onde M_i é um bloco de 512bits
3: $H_0 \leftarrow 67452301$
4: $H_1 \leftarrow EFCDAB89$
5: $H_2 \leftarrow 98BADCFE$
6: $H_3 \leftarrow 10325476$
7: $H_4 \leftarrow C3D2E1F0$
8: **for** $i \leftarrow 1$ to n **do**
9:     denote $M_i = W_0 \| W_1 \| ... \| W_{15}$, onde cada W_j tem 32bits
10:     **for** $t \leftarrow 16$ to 79 **do**
11:         $W_t \leftarrow ROTL^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16})$
12:     **end for**
13:     $A \leftarrow H_0$
14:     $B \leftarrow H_1$
15:     $C \leftarrow H_2$
16:     $D \leftarrow H_3$
17:     $E \leftarrow H_4$
18:     **for** $t \leftarrow 0$ to 79 **do**
19:         $temp \leftarrow ROTL^5(A) + f_t(B, C, D) + E + W_t + K_t$
20:         $E \leftarrow D$
21:         $D \leftarrow C$
22:         $C \leftarrow ROTL^{30}(B)$
23:         $A \leftarrow temp$
24:     **end for**
25:     $H_0 \leftarrow H_0 + A$
26:     $H_1 \leftarrow H_1 + B$
27:     $H_2 \leftarrow H_2 + C$
28:     $H_3 \leftarrow H_3 + D$
29:     $H_4 \leftarrow H_4 + E$
30: **end for**
31: **return** $(H_0 \| H_1 \| H_2 \| H_3 \| H_4)$

2.3 Curvas Elípticas

Nesta seção será abordado a matemática em torno das curvas elípticas. Para este fim tomarei como base [13].

Uma Curva Elíptica é definida em [13] como *um tipo especial de equação polinomial.* Acrescido da definição em [9], uma curva elíptica sobre um campo K é um grupo de pontos (x, y) que satisfaz a equação $y^2 = x^3 + ax + b$ onde $a, b \in K$.

Dentro da aplicação da criptografia é necessário utilizarmos apenas pontos discretos, para isso faremos calculos sobre o modulo de um número primo p, em outras palavras um campo finito Z_p. Mas apesar de ao usar estes pontos não se descrever uma curva propriamente dita, é possível, para fins didáticos, desenhar uma curva que represente tal equação.

Para este exemplo será utilizado o apresentado em [13], definido pela equação $y^2 = x^3 - 3x + 3$ e a figura 2.12 correspondendo esta curva.

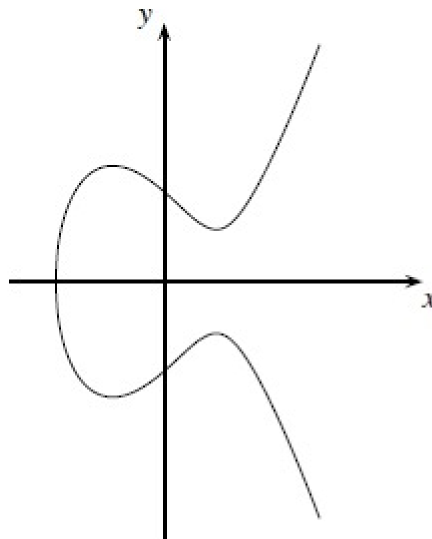

Figura 2.12: Curva Elíptica - retirado de [13] pg.241

Na curva, alguns pontos interessantes podem ser vistos, como o caso de que a curva é simétrica em relação ao eixo x, isto se deve ao fato de que para qualquer y, tanto y quanto o $-y$ darão sempre o mesmo resultado x. Outro ponto importante de ser notado é o ponto em que $y = 0$, este é único no exemplo acima, pois essa equação possui uma raiz real e duas complexas, porém é possível ter mais pontos de intersecção, mas para o estudo das curvas usadas na criptografia é necessário apenas aquelas em que existe apenas um.

Para utilizar os pontos da curva como elementos de um grupo que será utilizado na criptografia, é necessária uma operação fechada no grupo de elementos para defini-los como um grupo cíclico finito, o que é necessário para a meta do projeto.

2.3.1 Operação em Curvas Elípticas

Como visto na seção 2.1.3 existem certas regras que uma operação deve ter. Define-se a operação que será explicada a seguir como *soma* e o símbolo utilizado para representá-la será o +, esta escolha é arbitrária, mas a será usada pois é a forma mais usada na literatura.

Uma operação deve ser executada entre dois elementos do grupo, então serão adotados os pontos $P(x_1, y_1)$ e $Q(x_2, y_2)$ para o fazer, de tal forma que $P + Q = R$, onde R também é um ponto dentro do grupo de elementos. Tem-se que:

$$P + Q = R \tag{2.19}$$

ou em outras palavras:

$$(x_1, y_1) + (x_2, y_2) = (x_3, y_3) \tag{2.20}$$

A operação é definida aparentemente de forma arbitrária, mas se analisar sua representação geométrica no plano real é possível entender como que esta operação funciona.

Esta operação se diferencia quando se tem dois pontos distintos e quando se tem um único ponto *somado* com ele mesmo. Seguindo a literatura denomina-se estes dois casos de *adição ou soma de pontos* e *dobra de ponto*.

Na *adição de pontos* se tem dois pontos na curva P e Q onde $P \neq Q$ e a operação resultará em um terceiro ponto R tal que é dito que $P + Q = R$. Isto, geometricamente, é feito traçando uma reta que intercepta os pontos P e Q, ao fazer isto um terceiro ponto é interceptado que será chamado de R'. O ponto inverso de R' em relação ao eixo x, lembrando da simetria da curva já mencionada, será o $R = P + Q$, ou seja, se $R' = (x, y)$ então $R = (-x, y)$, como mostrado na figura 2.13:

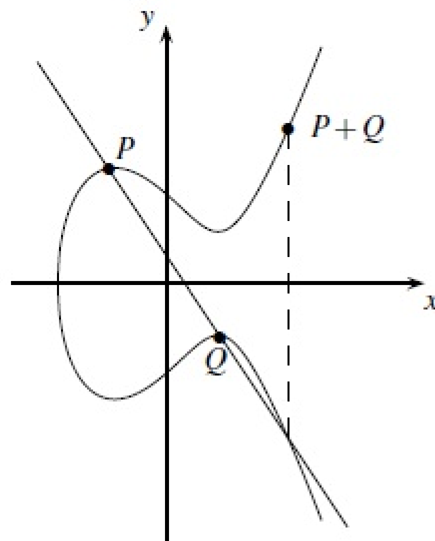

Figura 2.13: EC: soma de pontos - retirado de [13] pg.243

Na *dobra de ponto* será mudada, um pouco, a linha de raciocínio, na verdade, se levar em consideração as primeiras lições de cálculo que dizem respeito a limites e derivadas,

é possível ver que é basicamente a mesma coisa, pois neste caso se tem novamente dois pontos P e Q, porém agora tem-se que $P = Q$, logo o que seria $P + Q$ acaba se tornando $P + P$ ou, como será definido, $2P$, então o que se quer encontrar é o ponto $R = 2P$, o que resultará não em uma reta secante que intercepta os dois pontos P e Q, mas uma tangente que passa pelo ponto P e intercepta apenas mais um ponto na curva, o ponto R', o resto do algoritmo é o mesmo onde R será o espelho em relação ao eixo$x$ de R', como mostrado na figura 2.14

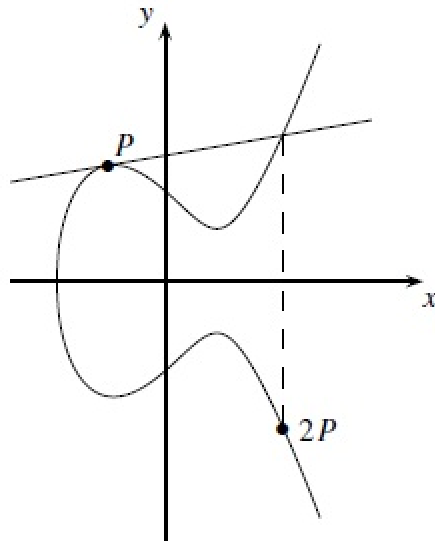

Figura 2.14: EC: dobra do ponto - retirado de [13] pg.243

Com isto, a definição de uma operação fechada está próxima. Pois já se tem o fato dela ser associativa, mas a prova disto não será descrita neste documento, porém pode ser encontrada em [13]. Faltam apenas duas definições, o elemento neutro da operação ϑ, onde $\forall P \in E$ onde E é o grupo de pontos pertencentes à curva elíptica em questão, $P + \vartheta = P$. A outra definição é a inversa da operação, onde cada ponto P possui uma inversa dentro do grupo que definiremos como $-P$ tal que $P + (-P) = \vartheta$.

Percebe-se ao fazer testes de soma de pontos que existe um caso em que a reta não atinge nenhum outro ponto na curva, são os casos em que a reta é paralela ao eixo$y$, para que esta operação permaneça fechada inclui-se ao grupo a definição de ponto no infinito, que é como diz o nome, um ponto no infinito do plano. Ao definir este ponto percebe-se que este age exatamente como o elemento neutro da operação, pois uma reta que venha deste ponto do infinito paralelamente ao eixo$y$ e atinja o ponto P, atingirá também um outro ponto na curva, seguindo o algoritmo da soma de pontos percebe-se que o espelho deste ponto com relação ao eixo$x$ é o próprio P, logo este ponto no infinito somado a P, seja qual for o P, resultará no próprio. Logo se tem que o nosso ponto no infinito é o elemento neutro ϑ.

E da mesma analise percebe-se que os casos em que a reta não intercepta o ponto espelho da soma e sim o ponto do infinito são os que P e Q são respectivamente (x, y) e $(-x, y)$, logo, é possível afirmar que a inversas de um ponto $P(x, y)$ é o ponto $-P$ cujas coordenadas são $(-x, y)$. Com isso, os requisitos para ter um grupo está completo.

É possível ver a prova geométrica das definições acima na figura 2.15

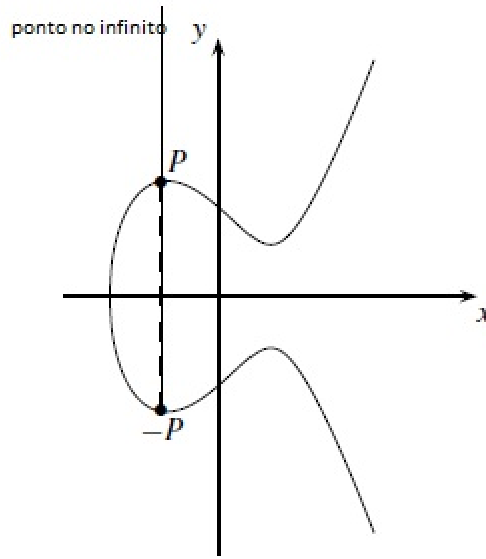

Figura 2.15: EC: Elemento Neutro - retirado de [13] pg.245 - com alterações

Entretanto, para fins criptográficos, apenas representações geométricas não são úteis, mas com a utilização de cálculos geométricos simples é possível criar expressões para cada um dos casos de adição. O processo geométrico para chegar às fórmulas não é importante para o objetivo deste documento, então, elas serão apenas expostas.

Dado R o resultado da soma e R_x e R_y as coordenadas dele, os pontos P e Q, com suas coordenadas (P_x, P_y) e (Q_x, Q_y), respectivamente, e $P, Q, R \in E$, sendo E uma Curva Elíptica descrita em um campo de Galois $GF(p)$ em que p é um primo e a e b são coeficientes da curva, primeiramente, calcula-se um valor intermediário s, como demonstrado na equação 2.21

$$\begin{gathered} s \equiv (Q_y - P_y) * (Q_x - P_x)^{-1} \pmod{p}; \forall P \neq Q \\ s \equiv (3P_x^2 + a) * (2P_y)^{-1} \pmod{p}; \forall P = Q \end{gathered} \tag{2.21}$$

Depois calcula-se as coordenadas de R, com as equações 2.22.

$$\begin{gathered} R_x \equiv s^2 - P_x - Q_x \pmod{p} \\ R_y \equiv s(P_x - R_x) - P_y \pmod{p} \end{gathered} \tag{2.22}$$

Para exemplificar será usado um exemplo apresentado em [13]. Neste exemplo será usada a Curva E definida pela equação $y^2 = x^3 + 2x + 2 \pmod{17}$, e o ponto P da curva com coordenadas $(5, 1)$. Com isso a nossa conta fica assim:

$$
\begin{gathered}
2P = P + P = (5,1) + (5,1) = (R_x, R_y) \\
s \equiv (3P_x^2 + a) * (2P_y) \pmod{p} \Rightarrow (2*1)^{-1}(3*5^2+2) \equiv 13 \pmod{17} \\
R_x \equiv s^2 - P_x - Q_x \pmod{p} \Rightarrow 13^2 - 5 - 5 \equiv 6 \pmod{17} \\
R_y \equiv s(P_x - R_x) - P_y \pmod{p} \Rightarrow 13(5-1) - 1 \equiv 3 \pmod{17} \\
2P = (5,1) + (5,1) = (6,3)
\end{gathered}
\tag{2.23}
$$

2.3.2 Definindo o Problema do Logaritmo Discreto com Curvas Elípticas

Inicialmente será definido o que é o problema do logaritmo discreto que será chamado a partir de agora pela sigla de sua tradução do inglês *DLP*.

O DLP em um campo módulo um primo p é definido em [13] da seguinte maneira.

"Dado um Grupo Finito Cíclico Z_p^* de ordem $p-1$ e um elemento primitivo $\alpha \in Z_p^*$ e outro elemento $\beta \in Z_p^*$. O DLP é o problema de determinar o inteiro $1 \leq x \leq p-1$ tal que $\alpha^x \equiv \beta \pmod{p}$".

Antes de interligar este conceito com as curvas elípticas, será abordada outra característica das curvas elípticas.

Como descrito em [13], os pontos de uma curva elíptica junto com o ponto ϑ possuem subgrupos cíclicos, e em certos casos, todos os pontos formam um grupo cíclico.

A importância desta característica é clara, vendo que sabe-se as características de um grupo cíclico, como por exemplo, que ele possui um elemento denominado gerador, a qual seus múltiplos $P, 2P, 3P, ..., (\#E)P$ geram todos os outros elementos do grupo, onde $\#E$ é o número de elementos existentes no grupo. E sabendo que existem meios de criar sistemas criptográficos a partir de grupos cíclicos, o objetivo necessário para o projeto foi alcançado.

Para exemplificar será utilizada a mesma curva e ponto gerador do exemplo anterior, o $\#E$ desta curva é 19, ou seja, um primo, o que diz que todos os elementos do grupo são geradores. Será mantido o ponto $(5,1)$ como o gerador.

Calculando os pontos $P, 2P, ..., 19P$ temos:

$$\begin{aligned}
P &= (5,1)\\
2P = P + P &= (6,3)\\
3P = 2P + P &= (10,6)\\
4P &= (3,1)\\
5P &= (9,16)\\
6P &= (16,13)\\
7P &= (0,6)\\
8P &= (13,7)\\
9P &= (7,6)\\
10P &= (7,11)\\
11P &= (13,10)\\
12P &= (0,11)\\
13P &= (16,4)\\
14P &= (9,1)\\
15P &= (3,16)\\
16P &= (10,11)\\
17P &= (6,14)\\
18P &= (5,16)\\
19P &= \vartheta
\end{aligned} \tag{2.24}$$

Perceba que o elemento $19P = \vartheta$, o elemento neutro, e que se somar P a este valor, obtendo $20P$ o resultado será o próprio P e a partir daí recomeça o ciclo.

Outro ponto importante a ser visto é o fato de que não há um método de calcular o valor de dP onde $d \in Z_p$ e $P \in E$ a não ser a soma sucessiva, assim como no mundo dos inteiros por mais robusto que sejam os métodos de calcular um exponencial, acaba sendo uma série de multiplicações.

Logo, o cálculo inverso, ou seja, dado o gerador P e outro ponto da curva Q, calcular o inteiro d que satisfaça a equação $dP = Q$, possui dificuldade semelhante ao da DLP, assim sendo considerado a DLP das curvas elípticas.

Com isso conclui-se que tendo um grupo cíclico e podendo aplicar a DLP dentro das curvas elípticas, é possível sim utilizar este novo grupo em processos criptográficos.

Além disto, por ser uma área recente dentro da matemática, poucos métodos de diminuir a complexidade da operação que gera o DLP foram estudadas, diferente da exponenciação de inteiros, que já tem amplo estudo. Assim, a dificuldade de resolver o DLP de uma exponenciação com um certo número de bits é alcançada pelas curvas elípticas com a utilização de muito menos bits, fazendo com que o armazenamento necessário para chaves e assinaturas seja menor e sua velocidade de processamento seja maior.

2.4 Segurança Computacional do Problema do Logaritmo Discreto

Por fim, para entender a importância do estudo de novas técnicas de se obter o problema do logaritmo discreto, será definido o conceito de Segurança Computacional em assinaturas digitais.

Dentro de um sistema criptográfico de assinatura digital, pode-se definir que um sistema é seguro se, dado um sistema A que assina um documento e um sistema B que o recebe, a assinatura que for verificada como sendo do sistema A só possa ter sido assinado por A, e que esta assinatura não pode ser usada em outro documento, como no ataque de aniversário descrito na seção 2.2.4.

Para atingir estes objetivos, em qualquer sistema assimétrico, ou seja, que se utiliza de um par de chaves, pública e privada, a segurança do sistema depende da premissa que apenas o dono do par de chaves possua a chave privada, e que, o resultado de suas operações não deem pista da mesma chave.

Com isto, dentro da assinatura digital, é definido como segurança computacional a dificuldade de descobrir a chave privada de um sistema se utilizando de elementos públicos da assinatura. Como o próprio sistema de assinatura, a chave pública ou a assinatura com a mensagem.

Para alcançar esta dificuldade, os esquemas de assinatura digital se baseiam em problemas matemáticos de difícil resolução, mesmo para computadores.

Os principais métodos de assinatura digital se classificam em três grupos: os que se baseiam em fatoração de números grandes, os que se baseiam no problema do logaritmo discreto no campo dos inteiros e os que se baseiam no problema do logaritmo discreto em curvas elípticas.

O tamanho mínimo que os números do protocolo devem ter para que seja computacionalmente exaustivo calcular a chave privada é definido como nível de segurança de um sistema de assinatura.

Como exemplo, o representante dos que se baseiam na fatoração de inteiros, o RSA, explicado na seção 2.2.3, tem como nível de segurança aceitável hoje, números de 2048-3072 bits. O DSA que será explicado na seção 3.3, precisa de números de 1024 bits, e o ECDSA que será explicado na seção 3.4 necessita apenas de números de 160-256 bits de tamanho de chave privada.

Capítulo 3

Protocolo ECDSA

Neste capítulo será abordado o protocolo ECDSA, que é uma variação do padrão de assinatura proposto pelo *NIST* [1], o DSA, utilizando-se de álgebra em curvas elípticas. Ambas se baseiam no problema do logaritmo discreto(PDL). Outro famoso algoritmo de assinatura baseado no mesmo problema é o sistema de assinatura de Elgamal.

Por este motivo, este capítulo será dividido da seguinte maneira: primeiramente será definido o esquema de Elgamal e como ele é utilizado em assinatura, em seguida o DSA e por fim o ECDSA. As explicações aqui são baseadas em [13].

3.1 Esquema criptográfico de Elgamal

Antes de falar da assinatura baseada em Elgamal, é necessário entender como funciona este padrão.

O esquema de cifragem de Elgamal foi proposto em 1985 por Taher Elgamal, ele se baseia no problema do logaritmo discreto(PDL), ou seja encontrar o expoente x que responde a equação $\alpha^x \equiv \beta \pmod p$ onde p é primo e $\alpha, \beta \in Z_p^*$.

3.1.1 Geração das Chaves

A geração de chaves do esquema Elgamal funciona assim:

- Selecione um primo p grande.(Acima de 1024 bits)
- Selecione um elemento primitivo $\alpha \in Z_p^*$ ou de um subgrupo de Z_p^*
- Selecione de modo aleatório um d, tal que $2 \leq d \leq p - 2$
- Calcule $\beta \equiv \alpha^d \pmod p$

Assim a chave privada *prk* gerada é o número d e a chave pública *pbk* são os valores de (p, α, β).

[1] National Institute of Standards and Technology.

3.1.2 Cifragem

A forma que o esquema de Elgamal cifra uma mensagem x utilizando o par de chaves gerado é a seguinte:

- Escolha de forma aleatória um i, tal que $2 \leq i \leq p-2$
- Calcule $k_e \equiv \alpha^i \pmod p$
- Calcule $k_m \equiv \beta^i \pmod p$
- Calcule $y \equiv x * k_m \pmod p$

A mensagem cifrada então é y a qual é mandada juntamente com chave efêmera k_e.

3.1.3 Decifragem

Ao receber o par (y, k_e) e com posse da chave privada, o método de decifragem é este:

- Calcule $k_m \equiv k_e^d \pmod p$
- Calcule $x \equiv y * k_m^{-1} \pmod p$

A prova de que o x calculado é igual a mensagem original, vem da propriedade da exponenciação que diz $(a^b)^c = (a^c)^b$, observem o sistem 3.1.

$$
\begin{aligned}
x &\equiv y * k_m^{-1} \pmod p \\
&\equiv (x * km) * (k_e^d)^{-1} \pmod p \\
&\equiv (x * (\beta^i)) * ((\alpha^i)^d)^{-1} \pmod p \\
&\equiv (x * (\alpha^d)^i) * \alpha^{-di} \pmod p \\
&\equiv x * \alpha^{di} * \alpha^{-di} \pmod p \\
x &\equiv x * 1 \pmod p
\end{aligned}
\tag{3.1}
$$

3.2 Assinatura de Elgamal

O esquema de assinatura de Elgamal, foi proposto em 1985 e tem por base o PLD. Diferente do algoritmo RSA, abordado por alto no capítulo anterior, as funções de assinatura e verificação são bem diferentes. Serão abordados então os outros dois passos do protocolo, a assinatura e a verificação, pois a geração de chaves já foi exposta na seção 3.1.1.

3.2.1 Assinatura

A assinatura de Elgamal recebe a mensagem x, e com a utilização da chave privada d, produz uma assinatura que consiste de dois números r e s. O processo é dado da seguinte forma:

- Escolhe um número aleatório k_e, tal que $2 \leq k_e \leq p-2$ e $mdc(k_e, p-1) = 1$.
- Calcula $r \equiv \alpha^{k_e} \pmod{p}$
- Calcula $s \equiv (x - d * r) * k_e^{-1} \pmod{p-1}$

É enviado então a mensagem com sua assinatura, ou seja, $(x, (r, s))$.

3.2.2 Verificação

No protocolo de assinatura de Elgamal, quando uma pessoa recebe uma mensagem assinada $(x, (r, s))$, este usa a chave pública $pbk = (p, \alpha, \beta)$ da pessoa que diz ter assinado e verifica do seguinte jeito:

- Calcule o valor de $t \equiv \beta^r * r^s \pmod{p}$
- Calcule o valor de $w \equiv \alpha^x \pmod{p}$

Caso $t = w$ a assinatura é válida, caso contrário, a assinatura é inválida.

3.2.3 Prova de Validade

Para verificar a validade do protocolo, é necessário fazer algumas manipulações algébricas. Na validação uma das contas que deve ser feita é a do sistema 3.2.

$$\begin{aligned} t &\equiv \beta^r * r^s \pmod{p} \\ &\equiv (\alpha^d)^r * (\alpha^{k_e})^s \pmod{p} \\ &\equiv \alpha^{d*r+k_e*s} \pmod{p} \end{aligned} \tag{3.2}$$

Para que seja verificada a assinatura esta conta deve ser equivalente a α^x. De acordo com o Pequeno Teorema de Fermat, $a^p \equiv a \pmod{p}$, com algumas manipulações explicadas em [13], é possível escrever que, sendo p primo, $p \equiv 1 \pmod{p-1}$, assim se tem o sistema 3.3

$$\begin{aligned} \alpha^x &\equiv \alpha^{d*r+k_e*s} \pmod{p} \\ x &\equiv d * r + k_e * s \pmod{p-1} \\ s &\equiv (x - d * r) * k_e^{-1} \pmod{p-1} \end{aligned} \tag{3.3}$$

É visto no sistema 3.3 que o resultado da manipulação algébrica é exatamente a conta que gera o número s, confirmando assim que a verificação é factível.

3.3 Algoritmo de Assinatura Digital

O esquema de assinatura DSA, proposto pelo NIST, é uma variante do esquema de Elgamal. O DSA foi comprovado mais robusto contra ataques que funcionavam contra o Elgamal e sua assinatura é menor do que o mesmo. Assim é um esquema muito mais utilizado.

Para explicar este esquema utilizaremos o primo p com tamanho de 1024 bits.

3.3.1 Geração de Chaves

Para a geração de chaves segue-se os passos abaixo:

- Gera um primo p com tamanho de 1023 a 1024 bits
- Encontre um primo q com tamanho entre 159 a 160 bits que divida $p - 1$
- Encontre um elemento α que gere um grupo cíclico de ordem q
- Escolha um d tal que $1 \leq d \leq q - 1$
- Calcule $\beta \equiv \alpha^d \pmod p$

As chave privada será o número d, e a chave pública sera o grupo de valores (p, q, α, β).

A idéia de ter dois subgrupos cíclicos é ter a segurança de um subgrupo grande de ordem $ord(p)$ com uma assinatura pequena dada pelo tamanho de q.

3.3.2 Assinatura

Assim como no esquema de Elgamal, a assinatura do DSA possui dois números, r e s, com o tamanho em bits igual ao de q, tendo a chave privada d e a mensagem x, calcula-se a assinatura da seguinte maneira:

- Escolha um número k_e de modo aleatório, tal que $1 \leq k_e \leq q - 1$
- Calcule $r \equiv (\alpha^{k_e} \pmod p) \pmod q$
- Calcule $s \equiv (h(x) + d * r) * k_e^{-1} \pmod q$

Onde $h(x)$ é um resumo criptográfico da mensagem x, neste exemplo, por precisar que o $h(x)$ não tenha mais de 160 bits, o uso tradicional é do SHA-1, já explicado na seção 2.2.5.

A assinatura é enviada junto à mensagem x na forma $(x, (r, s))$

3.3.3 Verificação

Assim que alguém recebe a mensagem assinada acima, ela, utilizando a chave pública da pessoa que diz ter assinado e verifica essa assinatura da seguinte maneira.

- Calcule $w \equiv s^{-1} \pmod{q}$
- Calcule $u_1 \equiv w * h(x) \pmod{q}$
- Calcule $u_2 \equiv w * r \pmod{q}$
- Calcule $v \equiv (\alpha^{u_1} * \beta^{u_2} \pmod{p}) \pmod{q}$

Caso $v \equiv r \pmod{q}$ a assinatura é válida, caso contrário, a assinatura é inválida.

3.3.4 Prova de Validade

A prova inicia-se com o sistema 3.4.

$$\begin{gathered} s \equiv (h(x) + d * r) * k_e^{-1} \pmod{q} \\ k_e \equiv s^{-1} * h(x) + d * r * s^{-1} \pmod{q} \\ k_e \equiv u_1 + d * u_2 \pmod{q} \end{gathered} \tag{3.4}$$

A escolha de q ser primo divisor de $p - 1$ permite que se utilize a última equação de 3.4 e a altere para $\alpha^{k_e} \pmod{p} \equiv \alpha^{u_1 + d * u_2} \pmod{p}$. Seguindo se tem o sistema 3.5

$$\begin{gathered} \alpha^{k_e} \pmod{p} \equiv \alpha^{u_1 + d * u_2} \pmod{p} \\ \alpha^{k_e} \pmod{p} \equiv \alpha^{u_1} * \beta^{u_2} \pmod{p} \\ (\alpha^{k_e} \pmod{p}) \pmod{q} \equiv (\alpha^{u_1} * \beta^{u_2} \pmod{p}) \pmod{q} \\ r \equiv v \pmod{q} \end{gathered} \tag{3.5}$$

Assim, percebe-se a partir da última equação do sistema 3.5, que a verificação de assinatura é factível.

3.4 Algoritmo de Assinatura Digital em Curvas Elípticas

Por fim, nesta seção, será abordado do algoritmo que é o objetivo deste documento. O ECDSA é um algoritmo baseado em aritmética de curvas elípticas e está fortemente

ligado ao esquema DSA. Em 1998 o esquema de assinatura ECDSA foi padronizado pela ANSI[2] e em 1999 começou a ser usado, principalmente na rede *BitCoin*.

Em particular, por ser uma matemática muito recente, ainda não se tem ataques robustos aos protocolos baseados em curvas elípticas, logo, é obtido um nível satisfatório de segurança com números entre 160-256 bits, enquanto protocolos baseados em exponenciação de números inteiros necessitam de números de 1024-3072 bits. Esta comparação pode ser encontrada em [13]. Com números menores, as contas ficam mais rápidas, o que para criptografia é ótimo.

3.4.1 Geração de Chaves

Lembrando da matemática em curvas elípticas, descrita na seção 2.3, para definir uma curva é necessário os coeficientes a e b da equação $y^2 = x^3 + a * x + b$, e um primo p para criar o grupo finito e cíclico, que possui uma ordem q e pelo menos um elemento primitivo A.

Tendo esses elementos, é gerado o par de chaves da seguinte forma:

- Escolha um número aleatório d, tal que $1 \leq d \leq q - 1$
- Calcule o ponto $B = d * A$

A chave privada será o número d, e a chave pública é o conjunto de valores formados por (p, a, b, q, A, B). Note que enquanto o DSA forma a DLP ao fazer a conta $\alpha^d \pmod p$ o ECDSA a forma com a operação multiplicativa $d * A$.

3.4.2 Assinatura

Assim como os esquemas de Elgamal, e o DSA, a assinatura gerada pelo ECDSA possui dois números r e s com mesmo tamanho em bits de q. Usando as chaves privada e pública geradas, uma mensagem x é assinada da seguinte maneira.

- Escolha um número k_e de forma aleatória, em que $0 < k_e < q$
- Calcule $R = k_e * A$ e associe o valor da coordenada x de R ao elemento r
- Calcule $s \equiv (h(x) + d * r) * k_e^{-1} \pmod q$

Onde $h(x)$ é o resumo criptográfico da mensagem. Neste documento, falaremos do ECDSA baseado em números de 256 bits, e a implementação segue este mesmo esquema. Para tal esquema a função de resumo criptográfico escolhida usualmente é o SHA-256.

A assinatura depois deste processo é enviada junto com a mensagem na forma (x,(r,s)).

3.4.3 Verificação

Ao receber um documento assinado pelo protocolo ECDSA, o indivíduo que pretende verificar esta assinatura, deve, com o auxílio da chave pública do dito assinante, fazer o seguinte passo:

[2] American National Standards Institute

- Calcule $w \equiv s^{-1} \pmod{q}$
- Calcule $u_1 \equiv w * h(x) \pmod{q}$
- Calcule $u_2 \equiv w * r \pmod{q}$
- Calcule $P = u_1 * A + u_2 * B$

A assinatura será válida se a coordenada x do ponto P, denominada x_P, seja equivalente ao valor de r módulo q. Caso contrário, a assinatura é inválida.

3.4.4 Prova de Validade

Inicia-se novamente pelo cálculo da assinatura que gera o parâmetro s. Fazendo manipulações, se tem o sistema 3.6.

$$\begin{gathered} s \equiv (h(x) + d * r) * k_e^{-1} \pmod{q} \\ k_e \equiv s^{-1} * h(x) + d * s^{-1} * r \pmod{q} \\ k_e \equiv u_1 + d * u_2 \pmod{q} \end{gathered} \tag{3.6}$$

Sabendo que A gera um grupo cíclico de ordem q, é possível escrever que $k_e * A = (u_1 + d * u_2) * A$, o restante da prova está demonstrado no sistema 3.7

$$\begin{gathered} k_e * A = (u_1 + d * u_2) * A \\ k_e * A = u_1 * A + d * u_2 * A \\ k_e * A = u_1 * A + u_2 * B \\ R = P \end{gathered} \tag{3.7}$$

Assim o $r = x_R$ e o x_P, são equivalentes, pois $R = P$. Ou seja, a verificação está correta.

Isto define o protocolo ECDSA, no próximo capítulo, será abordado como que o módulo implementado para concretizar este aprendizado foi feito.

Capítulo 4

Documentação do Módulo

IMPORTANTE

Antes de documentar a implementação, é bom deixar claro que esta não foi feita para ser usada em sistemas de segurança reais, visto que o objetivo deste módulo é concretizar o estudo em cima de assinatura digital de curva elíptica e por isso o recurso de geração de números aleatórios não é considerado seguro pela literatura, mas foi escolhido por ser de implementação e entendimento simples.

Documentação

Neste capítulo será descrito como foi a implementação dos módulos, seus algoritmos e parâmetros. Depois de explicar rapidamente o *header* [1] *basics.h*, será explicado o módulo *bigint.h* onde definiremos o tipo inteiro de 256 bits, não nativo da linguagem C++, e suas operações básicas.

Em seguida será abordado a implementação das funções do módulo *sha256.h*, onde ocorre o resumo criptográfico das mensagens. Logo após, será explicado o módulo *Algebra_op.h*, onde foi implementada a algebra de teoria dos números necessária para o protocolo ECDSA. O próximo a ser documentado é o módulo *EC_Point.h*, onde é definido a classe do ponto em uma curva elíptica. Continuando se tem a documentação do módulo *EC_op.h*, onde é implementada as operações de curvas elípticas.

Por fim será explicada a implementação do protocolo ECDSA no módulo *ECDSA.h*, assim como a interface criada para demonstrar a utilização dos módulos, implementada no arquivo *256ECDSA.cpp*.

Para uma visualização melhor da dependência entre os módulos, se tem a figura 4.1, onde a seta sai do módulo dependente e vai para o módulo a qual o primeiro tem dependência.

4.1 basics.h

Neste pequeno *header* apenas se incluem as bibliotecas que serão necessárias para os próximos módulos, tirando o *bigint.h* que também é incluído dentro da *basics.h*.

Eis o código:

[1] arquivo cuja extensão é .h, onde contém definições de tipos,estruturas e funções

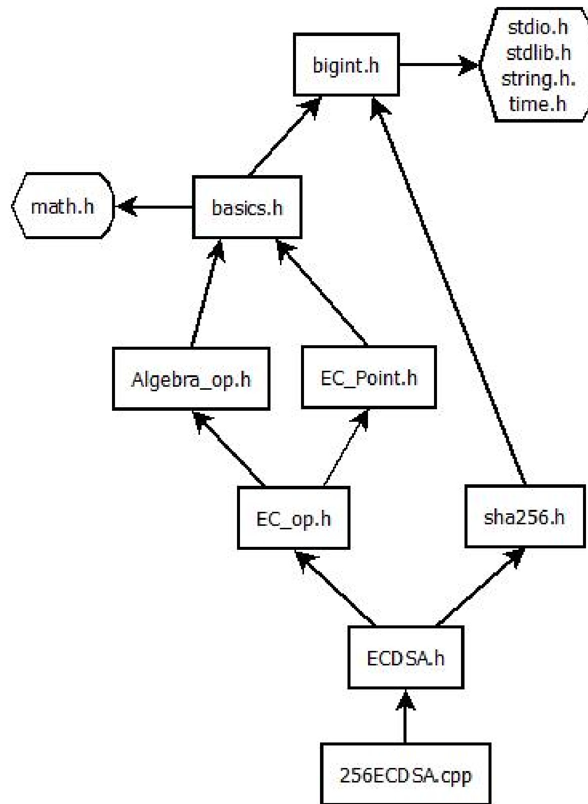

Figura 4.1: Diagrama de Comunicação

```
1 #ifndef BASICS_H_
2 #define BASICS_H_
3
4 #include <math.h>
5 #include "bigint.h"
6
7 #endif
```

4.2 bigint.h

A idéia deste módulo, como já explicado na introdução é criar uma implementação simples, sem a necessidades de bibliotecas que não estejam dentro das padrões do *GCC* [2]. Para tal, um dos primeiros problemas encontrados é a falta de um tipo que represente um número inteiro de 256 bits, necessário para o protocolo de 256 bits do ECDSA que queremos montar.

Este módulo tem como objetivo definir o tipo *bigint* e suas operações básicas, como soma, multiplicação, entre outros.

Para alcançar este objetivo este módulo foi estruturado da seguinte maneira:

Foram incluídas as bibliotecas *stdio.h,stdlib.h,string.h,time.h.*

[2] compilador padrão de C/C++ para sistema unix, também adaptado para windows na versão minGW.

Foi definido o tipo *bigint*, que representará números de até 512bits, apesar de as operações se basearem em 256 bits, operações como a multiplicação de 256 bits necessitam de um armazenamento de 512 bits.

As funções definidas neste módulo definem as operações básicas entre números inteiros necessárias para a implementação do protocolo ECDSA. Elas são:

Para entrada e saída:

- **ler_bigint** - lê um número na forma de caractere(bytes) e guarda em um *bigint*
- **ler_hex_bigint** - lê um número na forma hexadecimal e guarda em um *bigint*
- **fler_hex_bigint** - lê um número em um arquivo na forma hexadecimal e guarda em um *bigint*
- **ler_decimal** - lê um número na forma decimal e guarda em um *bigint*
- **imprime_hex_bigint** - imprime um *bigint* na tela na forma hexadecimal
- **fimprime_hex_bigint** - imprime um *bigint* em um arquivo na forma hexadecimal
- **imprime_char_bigint** - imprime um *bigint* na tela na forma de caractere(byte)
- **imprime_decimal** - imprime um *bigint* na tela na forma decimal

Para Comparação:

- **not_null** - testa se um *bigint* é diferente de zero.
- **eq_bigint** - Testa se um *bigint* é igual a outro *bigint*
- **eqi_bigint** - Testa se um *bigint* é igual a um inteiro
- **lt_bigint** - Testa se um *bigint* é menor que outro *bigint*
- **ltu_bigint** - Testa se um *bigint* é menor que outro *bigint* na representação sem sinal
- **let_bigint** - Testa se um *bigint* é menor ou igual a outro *bigint*.

Para operações lógicas:

- **append_int_bigint** - acrescenta os bits de um inteiro no lado menos significativo do *bigint* e descarta o mesmo número de bits adicionados no lado mais significativo
- **srl1_bigint** - Dá um shift lógico de 1 bit à direita em um *bigint*
- **sll1_bigint** - Dá um shift lógico de 1 bit à esquerda em um *bigint*

Para operações algébricas:

- **init_bigint** - Inicia um *bigint* com zero
- **neg_bigint** - Troca o sinal de um *bigint*
- **random_bigint** - Gera um *bigint* pseudo-aleatório

- **atrib_bigint** - Atribui o valor de um *bigint* em outro
- **atribi_bigint** - Atribui o valor de um inteiro em um *bigint*
- **add_bigint** - Soma dois *bigint*
- **addi_bigint** - Soma um *bigint* com um inteiro
- **sub_bigint** - Subtrai um *bigint* por outro
- **subi_bigint** - Subtrai um *bigint* por um inteiro
- **mult_bigint** - Multiplica dois *bigint*
- **multi_bigint** - Multiplica um *bigint* com um inteiro
- **div_mod_bigint** - Divide um *bigint* por outro, obtendo quociente e resto
- **div_mod_i_bigint** - Divide um *bigint* por um inteiro, obtendo quociente e resto
- **pow_modp_bigint** - Eleva um *bigint* por outro e este resultado é calculado módulo um terceiro *bigint*

A seguir a descrição mais detalhada de cada uma.

4.2.1 Tipo bigint

O tipo *bigint* nada mais é que uma cadeia de bytes, ele é definido da seguinte maneira:

```
typedef unsigned char bigint[TBITS+1]
```

Onde a constante TBITS está definido com 65, 1 bit para armazenar o sinal do número mantendo $2^{512} - 1$ números positivos possíveis. Percebam que é necessário apenas um número positivo máximo de $2^{256} - 1$, mas na multiplicação de 2 números deste tamanho, precisamos armazenar um número de 64 bytes.

O +1 se deve a necessidade da linguagem de um byte para demarcar o fim da *string* [3] sempre que se utiliza uma.

Antes de começar a descrever como foram implementadas as funções a seguir, é bom salientar que tais algoritmos foram retirados em sua maior parte de [14] e de [13].

4.2.2 init_bigint

Esta função tem por objetivo inicializar uma variável *bigint* com o valor 0. Recebe como parâmetro a variável a ser inicializada a.

Isto é feito em um loop que passa por cada byte da variável atribuindo o valor 0 a cada uma. A definição fica assim:

```
void init_bigint(bigint a){
        int i;
    //loop que passa por todos os bytes da variavel
        for(i=0;i<TBITS;i++){
                a[i]=0;
```

[3] cadeia de caracteres.

```
6         }
7 }
```

4.2.3 not_null

Esta função recebe uma variável *bigint a* como o parâmetro e verifica se ela é igual a 0 ou não, caso seja, a função retorna falso, caso contrário, retorna verdadeiro.

Isto é feito em um loop que passa por cada byte da variável testando se ela é igual a 0, caso encontre um retornará falso, se depois de todos os testes, não for encontrado um byte igual a 0 retorna verdadeiro. O código fica assim:

```
1  bool not_null(bigint a){
2          int i;
3      //loop que passa por todos os bytes da variavel
4          for(i=0;i<TBITS;i++){
5          //caso o byte da vez for nao nulo, retorna verdadeiro
6                  if(a[i]!=0){
7                          return true;
8                  }
9          }
10     //caso todos sejam nulos, retorna falso
11         return false;
12 }
```

4.2.4 neg_bigint

Esta função recebe uma variável *bigint* e inverte todos os bits do número. Em seguida adiciona 1 ao resultado. Método este que inverte o sinal do número representado pela forma de complemento de 2 [4]. O código fica assim:

```
1  void neg_bigint(bigint a){
2          int i;
3      //loop que passa por todos os bytes da variavel
4          for(i=0;i<TBITS;i++){
5          //inverte todos os bits de cada byte
6                  a[i] = ~a[i];
7          }
8      //adiciona 1 ao valor de a
9          addi_bigint(a,a,1);
10 }
```

A explicação da função **addi_bigint** estará mais adiante.

[4]representação de número na base 2 em que o primeiro bit além de definir uma parte do valor do número também define se este é negativo ou positivo.

4.2.5 random_bigint

Esta função como dita anteriormente apenas simula um gerador aleatório, já que o objetivo do módulo é o protocolo do ECDSA implementado de forma simples não foi utilizado um método mais robusto.

Para simular um gerador de números aleatórios foi utilizada a função nativa da biblioteca *stdlib* como base. Para usar esta função é necessário no início do programa iniciar a semente. No código é usada a função *rand()* em cada byte até o 32º. Eis o código:

```
void random_bigint(bigint a){
        int i;
    //loop que percorre os bytes do numero
        for(i=0;i<TBITS;i++){
        //os 33 mais significativos ficam em 0, para que o numero
           seja de no maximo 256 bits
                if(i<=32){
                        a[i]=0;
                }else{
        //gera um numero aleatorio para cada byte
                        a[i] = rand()%256;
                }
        }
}
```

4.2.6 ler_bigint

Esta função recebe como parâmetro a variável *bigint* que armazenará o número lido em forma de caracteres, ou seja, armazenará exatamente os caracteres lidos. Eis o código:

```
void ler_bigint(bigint s){
        int i,j;
        unsigned char c;
        i=0;
        c = getc(stdin);
    //o loop ira ler cada caracter enquanto este for diferente de
       espaco ou enter, ou se passar do tamanho maximo do tipo
        while(i<TBITS&&(c!='\n'&&c!=' ')){
                s[i]=c;
                c = getc(stdin);
                i++;
        }
    //caso o numero de caracteres forem menor que o tamnho do tipo,
        os bytes devem ser reposicionados
        if(i<TBITS){
                i = TBITS - i;
        }
    //este loop reposiciona os bytes para as posicoes menos
       significativas.
    //ao terminar os bytes restantes sao valorizados com zero.
        for(j=TBITS-1;j>=0;j--){
```

```
                if(j>=i){
                        s[j] = s[j-i];
                }else{
                        s[j] = 0;
                }
        }
}
```

Perceba que ao ler uma seqüência de caracteres como "abc" estes caracteres ficam nos bytes mais significativos da cadeia que forma o tipo bigint, por isso, depois de ler, reposicionamos estes para os bytes menos significativos. Como no exemplo:

Tabela 4.1: Tabela de Armazenamento do tipo bigint

0	1	2	...	TBITS-2	TBITS-1
a	b	c	...	0	0

depois do reposicionamento:

Tabela 4.2: Tabela de Armazenamento Retificado

0	1	...	TBITS-3	TBITS-2	TBITS-1
0	0	...	a	b	c

4.2.7 ler_hex_bigint

Esta função também é feita para ler do teclado o valor de um tipo bigint, mas o valor esperado na leitura é no formato hexadecimal. Para tal é utilizado uma manipulação da tabela ascii, de tal forma que caso o caracter lido for um algarismo é subtraído de seu valor 48, para que se tenha o valor correto de tal algarismo. No caso das letras A, B, C, D, E, F, para que seus valores fiquem iguais a $10, 11, 12, 13, 14, 15$, subtrai-se 55, por fim para as letras a, b, c, d, e, f, pelo mesmo motivo, subtrai-se 87.

Este valor é somado à variável, inicialmente zerada e depois de cada soma, com exceção da última, multiplica-se o resultado por 16. Assim fazendo a mudança de base de hexadecimal para binário. O código está abaixo:

```
void ler_hex_bigint(bigint s){
        unsigned char c;
        unsigned int n;
    //inicia-se a variavel com 0
        init_bigint(s);
        c = getc(stdin);
    //ler os caracteres ate que ache um espaco ou enter
        while(c!='\n'&&c!=' '){
        //multiplica o resultado por 16 para mudanca de base
                multi_bigint(s,s,16);
        //Analisa qual o caracter e faz a operacao necessaria
                if(c>47&&c<58){//algarismos
                        n = c-48;
```

```
14                  }else if(c>96&&c<103){//a,b,c,d,e,f
15                          n = c-87;
16                  }else{//A,B,C,D,E,F
17                          n = c-55;
18                  }
19          //adiciona o resultado a variavel
20                  addi_bigint(s,s,n);
21                  c = getc(stdin);
22          }
23  }
```

4.2.8 fler_hex_bigint

Esta função recebe como parâmetro uma variável tipo *bigint* "s" e um ponteiro para um arquivo. Ela funciona do mesmo jeito que a anterior, com a diferença que ao invés de receber os caracteres do teclado, ele lê de um arquivo. O código segue abaixo:

```
1   void fler_hex_bigint(bigint s,FILE *fp){
2           unsigned char c;
3           unsigned int n;
4       //inicia a variavel com o valor 0
5           init_bigint(s);
6           c = getc(fp);
7       //le caracter do arquivo enquanto este for diferente de espaco
           ou enter ou fim de arquivo
8           while(c!='\n'&&c!=' '&&!feof(fp)){
9           //multiplica o valor da variavel por 16 para mudanca de
               base
10                  multi_bigint(s,s,16);
11          //Analisa qual foi o tipo de caracter lido para transforma-
               lo adequadamente
12                  if(c>47&&c<58){//algarismo
13                          n = c-48;
14                  }else if(c>96&&c<103){//a,b,c,d,e,f
15                          n = c-87;
16                  }else{//A,B,C,D,E,F
17                          n = c-55;
18                  }
19          //adiciona valor lido a variavel
20                  addi_bigint(s,s,n);
21                  c = getc(fp);
22          }
23  }
```

4.2.9 ler_decimal

Esta última função de leitura é semelhante às anteriores, mas a entrada esperada do teclado é um número decimal. Para tal, além de apenas fazer as contas para transformar

os caracteres dos algorismos, multiplica-se a variável a cada algorismo lido por 10, já que a mudança de base agora é de decimal para binário.

Outro caso a ser analisado é se existe o símbolo "-" no início do número, o que informa que o valor é negativo, para este caso, ao final do cálculo do valor absoluto, apenas se inverte o valor com a função **neg_bigint**.

```
void ler_decimal(bigint s){
        unsigned char c;
        unsigned int n;
        bool nega=false;
    //inicia o valor da variavel com 0
        init_bigint(s);
        c = getc(stdin);
    //testa se o numero lido e negativo
        if(c=='-'){
                nega = true;
                c = getc(stdin);
        }
    //le os algorismos ate que achar um espaco ou o enter
        while(c!='\n'&&c!=' '){
        //multiplica o valor da variavel por 10 para mudanca de
           base
                multi_bigint(s,s,10);
        //transforma o caracter no valor do algarismo
                n = c-48;
        //adiciona o valor lido a variavel
                addi_bigint(s,s,n);
                c = getc(stdin);
        }
    //caso o numero lido tiver o sinal - inverte seu valor
        if(nega){
                neg_bigint(s);
        }
}
```

4.2.10 imprime_hex_bigint

Esta função recebe como parâmetro uma variável do tipo *bigint* onde está armazenado o número que será escrito na forma hexadecimal no terminal. Este é feito da seguinte maneira. Do byte mais significativo, começa-se a percorrer a cadeia enquanto não achar um byte diferente de zero. Caso todos sejam zero, imprime zero, caso contrário a partir do primeiro byte não zero, imprime o byte em forma hexadecimal. A seguir o código:

```
void imprime_hex_bigint(bigint s){
        int i;
        i=0;
    //procura o primeiro byte nao nulo.
        while(i<TBITS&&s[i]==0){
                i++;
```

```
7          }
8      //caso todos os bytes sejam zeros, imprime zero na tela
9          if(i==TBITS){
10                 printf("%02x",0);
11                 return;
12         }
13     //caso contrario, cada byte sera escrito na forma hexadecimal
14         while(i<TBITS){
15                 printf("%02x",s[i]);
16                 i++;
17         }
18 }
```

4.2.11 fimprime_hex_bigint

A idéia desta função é a mesma da anterior, a diferença está que esta recebe como parâmetro, também, um ponteiro para arquivo, e ao invés de escrever o número em hexadecimal na tela, este será escrito no arquivo a qual o ponteiro está associado.

```
1  void fimprime_hex_bigint(bigint s,FILE *fp){
2          int i;
3          i=0;
4      //procura o primeiro byte nao nulo.
5          while(i<TBITS&&s[i]==0){
6                  i++;
7          }
8      //caso todos os bytes sejam zeros, imprime zero no arquivo
9          if(i==TBITS){
10                 fprintf(fp,"%02x",0);
11                 return;
12         }
13     //caso contrario, cada byte sera escrito na forma hexadecimal
       no arquivo
14         while(i<TBITS){
15                 fprintf(fp,"%02x",s[i]);
16                 i++;
17         }
18 }
```

4.2.12 imprime_char_bigint

Esta função novamente escreve o número armazenado na variável *bigint* passada como parâmetro. Mas desta vez na forma de caracter. A implementação segue a mesma lógica, mas ao invés de imprimir na forma hexadecimal, imprime na forma de caracter.

```
1  void imprime_char_bigint(bigint s){
2          int i;
3          i=0;
4      //procura o primeiro byte nao nulo.
```

```
5	        while(i<TBITS&&s[i]==0){
6	                i++;
7	        }
8	    //caso todos os bytes sejam zeros, imprime caractere nulo na
	       tela
9	        if(i==TBITS){
10	                putc(0,stdout);
11	                return;
12	        }
13	    //caso contrario, cada byte sera escrito na forma de caractere
14	        while(i<TBITS){
15	                putc(s[i],stdout);
16	                i++;
17	        }
18	}
```

4.2.13 imprime_decimal

Esta função pega o parâmetro *bigint* passado para ela e a imprime na tela na forma decimal. Para esse código a lógica é um pouco mais extensa. Primeiramente, é avaliado se o número armazenado é negativo, caso seja, imprime o caracter ”-” na tela e transforma o número em positivo, caso contrário, não faz nada.

Para o próximo passo é necessário duas variáveis *bigint* para auxiliar. No primeiro passo dividi-se o valor do parâmetro por 10 e guarda o quociente em um dos auxiliares e o resto em outro. Imprime o algarismo relativo ao resto e se o quociente for diferente de zero, inicia-se o seguinte loop:

Dividi-se o quociente por 10 e substitui o novo quociente no lugar do antigo, armazenando o novo resto no lugar do antigo também, e novamente imprime o algarismo relativo ao resto na tela. Repete-se até que o quociente seja igual a zero.

O código fica assim:

```
1	void imprime_decimal(bigint s){
2	        bigint q,m,x;
3	        unsigned char num[80];
4	        int i;
5	    //atribui-se o valor do parametro a outra variavel para
	       preservar o original
6	        atrib_bigint(x,s);
7	    //se o numero for zero, simplesmente imprime 0
8	        if(!not_null(x)){
9	                printf("%d",0);
10	                return;
11	        }
12	    //testa se o bit mais significativo e 1(numero negativo), caso
	       seja imprime "-" na tela e inverte o sinal do numero
13	        if(s[0]&0x80){
14	                printf("-");
15	                neg_bigint(x);
16	        }
```

```
17          i=1;
18      //armazena o resto modulo 10 do numero num vetor que sera
           impresso depois
19          div_mod_i_bigint(q,m,x,10);
20          num[0] = m[TBITS-1]+48;
21      //loop que refaz a operacao acima enquanto o quociente for
           diferente de zero
22          while(not_null(q)){
23                  div_mod_i_bigint(q,m,q,10);
24                  num[i] = m[TBITS-1]+48;
25                  i++;
26          }
27          i--;
28      //imprime o vetor na ordem inversa, gerando na tela o numero
           correspondente ao armazenado
29          while(i>=0){
30                  putc(num[i],stdout);
31                  i--;
32          }
33  }
```

4.2.14 eq_bigint e eqi_bigint

As funções **eq_bigint** e **eqi_bigint** testam se os dois parâmetros passados a elas possuem valores iguais. A primeira é um teste direto entre duas variáveis do tipo bigint, a segunda testa um *bigint* com um inteiro nativo da linguagem C++. Para esta o que se faz é atribuir o valor do inteiro em uma variável do tipo *bigint* e chamar a função **eq_bigint**, agora sim, com dois bigint.

Para testar a igualdade entre dois elementos *bigint* a função simplesmente percorre cada byte da cadeia de um, comparando-o com o byte correspondente do outro, caso sejam iguais continua o loop, caso contrário, interrompe o loop retornando falso. Se depois de decorrido toda a cadeia não for encontrado par de bytes diferentes retorna verdadeiro.

```
1   bool eq_bigint(bigint a,bigint b){
2           int i;
3       //loop que percorre cada byte da cadeia
4           for(i=0;i<TBITS;i++){
5           //para cada byte testa-se se seu correspondente na outra
               cadeia e diferente ou igual
6                   if(a[i]!=b[i]){
7               //retorna falso caso encontre qualquer igual
8                           return false;
9                   }
10          }
11      //retorna verdadeiro caso todos sejam iguais
12          return true;
13  }
```

A função **eqi_bigint** não terá seu código exposto, visto que não há a necessidade, pois é simplesmente uma atribuição seguida de uma chamada da função acima.

4.2.15 lt_bigint e ltu_bigint

Ambas as funções **lt_bigint** e **ltu_bigint** recebem dois parâmetros do tipo bigint, o objetivo de ambas é testar se o primeiro parâmetro tem valor menor que o segundo. A única diferença entre elas é que no primeiro é considerado se os números são positivo ou negativo, enquanto no segundo, considera-se que são não sinalisados [5]

Para efeito prático só será mostrado o código do **lt_bigint**, visto que a outra função é idêntica, apenas tirando o teste de sinal. Como então funciona o algoritmo?

Primeiro, quando há teste de sinal, testa-se se o sinal dos dois parâmetros são diferentes, caso sejam e o primeiro tenha sinal negativo, ou seja, bit mais significativo igual a 1, a função termina retornando verdadeiro, caso seja positivo, ou seja, bit mais significativo igual a 0, retorna falso.

Caso os sinais sejam iguais, percorre-se cada posição da cadeia, do mais significativo pro menos significativo, testando se o byte do primeiro parâmetro é maior que o seu respectivo no segundo, caso seja, significa que o segundo é menor que o primeiro, então retorna falso, se o primeiro for menor que o segundo, retorna verdadeiro. Se depois de todos os bytes testados, não for achado par de bytes diferentes, retorna falso.

```
1  bool lt_bigint(bigint a,bigint b){
2          int i;
3      //testa se os sinais sao diferentes
4          if((a[0]&0x80)!=(b[0]&0x80)){
5                  if((a[0]&0x80)!=0){//a e negativo, logo menor que b
6                          return true;
7                  }else{//a e positivo, logo maior que b
8                          return false;
9                  }
10         }
11     //para sinais iguais, testa-se do byte mais significativo ate o
           menos, qual e o maior
12         for(i=0;i<TBITS;i++){
13                 if(a[i]<b[i]){//byte de a menor que o de b, a menor
                       que b
14                         return true;
15                 }else if(a[i]>b[i]){//byte de a menor que o de b, a
                       maior que b
16                         return false;
17                 }
18         }
19     //se nenhum par de bytes for diferente, a e igual a b
20         return false;
21 }
```

[5]tradução livre para unsigned, sem sinal.

4.2.16 let_bigint

Esta função testa se o primeiro dos dois parâmetros do tipo *bigint* que ela recebe é menor ou igual ao segundo parâmetro. Para tal, simplesmente é testado se este dois parâmetros mandados na mesma ordem que foram recebidos retornam verdadeiro para as funções **eq_bigint** ou **lt_bigint**. Caso sim retorna verdadeiro, senão falso. Visto a simplicidade da função, é desnecessário mostrar o código.

4.2.17 atrib_bigint e atribi_bigint

As funções **atrib_bigint** e **atribi_bigint** recebem dois parâmetros cada, a primeira recebe dois bigint's, a segunda recebe um *bigint* e um inteiro de 32 bits. Nas duas o objetivo é passar o valor do segundo parâmetro para o primeiro.

Para tal no **atrib_bigint** apenas se percorre cada byte da cadeia copiando o valor do byte do segundo parâmetro para seu respectivo no primeiro parâmetro, fica então assim:

```
1 void atrib_bigint(bigint a,bigint b){
2         int i;
3     //loop que passa byte a byte os valores da cadeia b para a
        cadeia a
4         for(i=0;i<TBITS;i++){
5                 a[i] = b[i];
6         }
7 }
```

Para a função **atribi_bigint**, onde passamos um valor de um inteiro para o *bigint* é necessário algo mais. Primeiramente, sabe-se que na arquitetura da linguagem C++, o tipo inteiro possui 32 bits, ou seja, 4 bytes. Estes que serão copiados nos 4 bytes menos significativos do parâmetro bigint.

Entretanto, é necessário analisar primeiro o sinal do inteiro, guarda-se a informação se este é positivo ou negativo, caso seja negativo, multiplica-se ele por -1, passando ele para o valor absoluto.

Depois simplesmente inicia-se o *bigint* com o valor 0. Em seguida é atribuído o valor dos 4 bytes do inteiro nos 4 bytes menos significativos do bigint.

Ao terminar este passo, analisa-se a informação guardada sobre o inteiro, a que diz se este é positivo ou negativo, caso seja negativo, inverte-se o sinal do *bigint* com a função **neg_bigint**. O código fica assim:

```
1 void atribi_bigint(bigint a, int n){
2         bool nega = false;
3     //inicia a com valor 0
4         init_bigint(a);
5     //testa se a e negativo
6         if(n<0){
7         //caso seja armazena esta informacao e transforma o numero
            em positivo
8                 nega = true;
9                 n*=-1;
10        }
```

```
11      //atribui os 4 bytes do inteiro nos 4 bytes menos
           significativos do bigint
12          a[TBITS-1] = n&0xff;
13          a[TBITS-2] = (n>>8)&0xff;
14          a[TBITS-3] = (n>>16)&0xff;
15          a[TBITS-4] = (n>>24)&0xff;
16      //caso o numero fosse inicialmente negativo, transforma o
           bigint em negativo
17          if(nega){
18                  neg_bigint(a);
19          }
20  }
```

4.2.18 append_int_bigint

Esta função recebe 2 parâmetros, o primeiro um *bigint*, o segundo um inteiro sem sinal, a idéia é concatenar os bits do *bigint* com os bits do inteiro. Pelo fato de ter o limite de tamanho do tipo *bigint* os 4 bytes mais significativos deste serão descartados. Como ilustrado na tabela 4.3, onde a é a variável *bigint* e n é a variável inteira.

Tabela 4.3: Concatenação com descarte

a[0]	a[1]	...	a[TBITS-5]	a[TBITS-4]	a[TBITS-3]	a[TBITS-2]	a[TBITS-1]
a[4]	a[5]	...	a[TBITS-1]	n[0]	n[1]	n[2]	n[3]

Perceba que os bytes $a[0], a[1], a[2], a[3]$ foram descartados.

O algoritmo para tal função funciona da seguinte forma: percorre-se o parâmetro *bigint* do byte 0, o mais significativo, até o byte $TBITS - 5$ e para cada um deles será atribuído o valor do byte que está 4 posições a frente, assim na prática move-se todos os bytes 4 posições em direção ao lado mais significativo. Depois apenas atribui-se nos 4 bytes menos significativos os bytes do parâmetro inteiro inteiro.

O código fica assim:

```
1   void append_int_bigint(bigint a, unsigned int n){
2           int i;
3           //move-se todos os bytes de a, 4 posicoes no sentido mais
               significativo
4       for(i=TBITS-32;i<TBITS-4;i++){
5                   a[i] = a[i+4];
6           }
7       //atribuimos o valor de n nos bytes menos significativos de a
8           a[TBITS-4] = (n>>24)&0xff;
9           a[TBITS-3] = (n>>16)&0xff;
10          a[TBITS-2] = (n>>8)&0xff;
11          a[TBITS-1] = (n)&0xff;
12  }
```

4.2.19 srl1_bigint e sll1_bigint

Estas funções tem como objetivo mover todos os bits do parâmetro passado para elas uma posição à direita(lado mais significativo) no caso da **srl1_bigint** ou a esquerda(lado menos significativo) no caso da **sll1_bigint**.

No primeiro caso, em que se quer mover os bits para direita, percorre-se byte a byte do *bigint*, começando do mais significativo, e para cada um é dado um shift [6] de 1 posição à direita, antes armazenando o bit que seria descartado.

Para o primeiro byte apenas opera-se o shift à direita, para os outros, logo depois de faze-lo, coloca-se o bit descartado no byte anterior no "espaço" [7] obtido após o movimento.

Para o caso do **sll1_bigint** a idéia é a mesma, as alterações são: ao invés de para cada byte ser dado um shift a direita, é dado à esquerda, e percorre-se os bytes do *bigint* pelo byte menos significativo. Assim será mostrado apenas o código referente ao **srl1_bigint**, visto que as alterações do outro código são triviais.

```
1  void srl1_bigint(bigint a){
2          char out,out2,aux;
3          int i;
4          out = 0;
5      //percorre cada byte de "a" a partir do byte mais significativo
           .
6          for(i=0;i<TBITS;i++){
7                  out2 = out;
8                  out = a[i]<<7;//guarda o bit a ser descartado pelo
                      shift
9                  aux = a[i]>>1;//shift a direita de 1 posicao
10                 //a sera a concatenacao do bit descartado
                      anteriormente e ele proprio dado um shift de 1
                      posicao a direita
11         a[i] = out2|aux;
12         }
13 }
```

4.2.20 add_bigint e addi_bigint

Nas próximas funções serão implementadas as funções algébricas básicas, para estas deve-se dizer que os algoritmos usados estão descritos em [14], e especificamente os algoritmos de multiplicação e potenciação, foram retirados de [13]

A função **add_bigint** recebe 3 parâmetros, o primeiro, denominado de c, será onde será armazenado o resultado da soma, os outros dois, denominados de a e b, são os operandos da soma. Logo se tem que $c = a + b$. Para tal percorre-se byte a byte, a partir do menos significativo e soma-se os bytes correspondentes de a e de b e de uma variável denominada *carry*, inicialmente zerada. Guarda-se a soma dos bytes em uma variável auxiliar *short* [8], onde o primeiro byte será posto no byte correspondente de c e o outro

[6] movimento lógico que significa mover todos os bits de uma variável tantas posições para algum lado.

[7] o shift automaticamente bota um bit 0 no bit que não tem antecessor, dizemos aqui que este é o "espaço"obtido após o shift.

[8] inteiro de 2 bytes

será posto no *carry* para a próxima soma. O último valor de *carry* é descartado. Assim temos que o código é:

```
1  void add_bigint(bigint c,bigint a,bigint b){
2          unsigned char carry;
3          unsigned short aux;
4          int i;
5          carry = 0;
6      //percorre byte a byte a partir do menos significativo
7          for(i=TBITS-1;i>=0;i--){
8          //soma-se os bytes de a e b, alem do carry obtido na soma
               anterior
9                  aux = a[i]+b[i]+carry;
10         //o primeiro byte vai para o byte de c
11                 c[i] = aux&0xFF;
12         //o segundo byte e o proximo carry
13                 carry = aux>>8;
14         }
15 }
```

A função **addi_bigint** recebe um parâmetro c *bigint*, seguido de um parâmetro a *bigint*, e um parâmetro n *int*. Primeiramente atribui-se o valor de n em uma variável auxiliar do tipo *bigint* b e chama-se a função **add_bigint** passando como parâmetro os valores c, a, b nesta ordem.

Não há necessidade de mostrar o código dada a simplicidade deste, o algoritmo na verdade é o da função descrita anteriormente.

4.2.21 sub_bigint e subi_bigint

As funções de subtração seguem praticamente o mesmo algoritmo da adição, para entender tal semelhança é necessário saber que uma subtração e adição em byte é feita no módulo 16, ou seja, $4 - 1 = 3 \equiv 3 \pmod{16}$ ou $1 - 4 = -3 \equiv 13 \pmod{16}$, assim, se um byte tem valor 1 e outro de valor 4, a subtração do segundo pelo primeiro corresponde a 13. Isso é muito útil, pois é como se, automaticamente, o primeiro pegasse emprestado um bit de seu byte vizinho mais significativo, o que, na verdade, é o que é feito na subtração quando o algorismo da vez é menor do que o valor a ser subtraído.

Logo, segue-se o mesmo algoritmo da soma, apenas trocando as operações de $+$ por $-$, o código fica assim, sabendo que os parâmetros passados são os mesmos da adição:

```
1  void sub_bigint(bigint c,bigint a,bigint b){
2          unsigned char carry;
3          unsigned short aux;
4          int i;
5          carry = 0;
6      //percorre byte a byte a partir do menos significativo
7          for(i=TBITS-1;i>=0;i--){
8          //subtrai-se o byte b de a, alem do carry obtido na
               subtracao anterior
9                  aux = a[i]-b[i]-carry;
10         //o primeiro byte vai para o byte de c
```

```
11                  c[i] = aux&0xFF;
12          //o carry sera igual ao bit mais significativo do segundo
                byte
13                  carry = (aux>>8)&0x1;
14          }
15  }
```

Assim, é possível também trabalhar de forma análoga a função **subi_bigint** tendo como base a função **addi_bigint**, ou seja, atribui-se o valor do parâmetro inteiro para uma variável *bigint* e chama-se a função **sub_bigint**. Logo, também é desnecessário mostrar o código aqui.

4.2.22 mult_bigint e multi_bigint

Estas funções tem como objetivo multiplicar o segundo e terceiro parâmetros e colocar no primeiro. A diferença é que na **mult_bigint** todos os parâmetros são do tipo *bigint*, enquanto a **multi_bigint** tem como terceiro parâmetro um inteiro.

Basicamente o que é feito na segunda função é transpor o valor do inteiro para uma outra variável do tipo *bigint* e chama-se a primeira. Por isso só será explicado a primeira.

A primeira função tem um algoritmo mais complexo. A fim de otimizar sua operação, é utilizado um algoritmo descrito em [13] denominado soma-e-dobra [9] onde é analisado bit a bit do multiplicador, a partir do mais significativo, e é feito a seguinte operação com o parâmetro c iniciado com 0.

- caso o bit seja 0, apenas dobra-se o valor de c.
- caso o bit seja 1, dobra-se o valor de c e soma-se o valor do multiplicando.

Para entender melhor, será mostrado um exemplo:

Seja $a = 5$ o multiplicando, $b = 6$ o multiplicador e c o resultado, temos o sistema 4.1.

$$
\begin{gathered}
b = 6 \Rightarrow b = (110)_2 \\
c = 0 \\
bit_2 = 1 \Rightarrow c \leftarrow (2 * c) + a \Rightarrow c = 5 \\
bit_1 = 1 \Rightarrow c \leftarrow (2 * c) + a \Rightarrow c = 15 \\
bit_0 = 0 \Rightarrow c \leftarrow (2 * c) \Rightarrow c = 30
\end{gathered}
\tag{4.1}
$$

Ou seja, $c = a * b = 30$, exatamente como esperado. Apesar de neste exemplo parecer mais simples fazer apenas $5*6 = 30$, mas em números maiores, principalmente de 256 bits, como o projeto demanda, se tem uma boa otimização comparada ao método tradicional escolar.

A algebra modular já demonstrada anteriormente garante também que o sinal do resultado estará correto.

[9] tradução livre de double-and-sum

O código fica assim:

```
void mult_bigint(bigint c,bigint a,bigint b){
        unsigned int i,j;
        bigint aux;
    //inicia o resultado como 0
        init_bigint(aux);
    //posiciona o b no primeiro byte mais significativo nao nulo.
       Para evitar contas desnecessarias.
        for(i=0;i<TBITS&&b[i]==0;i++);
    //a partir desta posicao, percorre o b byte a byte
        for(;i<TBITS;i++){
                \\para cada byte percorremos todos os bits
        j=0x80;
                while(j!=0){
            \\para cada bit fazemos a operacao descrita no double-
               and-sum
                        add_bigint(aux,aux,aux);
                        if(b[i]&j){
                                add_bigint(aux,aux,a);
                        }
                        j=j>>1;
                }
        }
    \\atribuimos o resultado final a c
        atrib_bigint(c,aux);
}
```

4.2.23 div_mod_bigint e div_mod_i_bigint

Estas funções possuem 4 parâmetros do tipo *bigint*, no caso da função **div_mod_i_bigint**, o último parâmetro é do tipo inteiro. O primeiro parâmetro, denominado d, receberá o quociente da divisão que será feita, o segundo, denominado m, receberá o resto da divisão, o terceiro e o quarto, denominados, respectivamente, de a e b serão o dividendo e o o divisor, nesta ordem.

O algoritmo é essencialmente o mesmo aprendido na escola, mas antes de partir para divisão, é necessário tratar a questão dos sinais. Sabe-se que o sinal do quociente será positivo se os sinais do divisor e do dividendo forem iguais, e negativo caso contrário. O sinal do resto será igual ao sinal do dividendo, mas para fins criptográficos, depois de calculado, ele é substituído pelo menor valor positivo equivalente no módulo do divisor.

Dado que foram feitas as operações necessárias dos sinais, é feita a divisão com os valores absolutos de a e de b. Inicia-se com duas variáveis auxiliares do tipo *bigint* $aux1$ e $aux2$, ambas iniciadas com 0. percorre-se bit a bit, a partir do mais significativo, os bits de a, sendo este bit 1, dobra-se o valor de $aux2$ e soma-se 1 ao seu resultado, caso seja 0, apenas dobra-se o seu valor, além disso, independente de qualquer coisa, dobra-se o valor do $aux1$. Perceba que na operação binária, dobrar o valor significa dar um shift a esquerda. Depois testa-se se o valor de $aux2$ é maior ou igual à b, caso seja, tiramos o valor b de $aux2$ e acrescenta-se 1 ao valor de $aux1$. Isto é repetido em todos os bits de

a. Ao acabar se tem que $aux1$ será o quociente da divisão e o $aux2$ o resto, aplicando as operações de sinal necessárias atribui-se tais valores a d e m.

Eis o código:

```
void div_mod_bigint(bigint d,bigint m,bigint a,bigint b){
        int i,j,sinalQ=1,sinalR=1;
        bigint a1,b1,aux1,aux2;
    //para manter o valor de a o copiamos em a1
        atrib_bigint(a1,a);
    //teste de sinal de a, resto tem o mesmo sinal
        if((a[0]&0x80)!=0){
                sinalR*=-1;
                sinalQ*=-1;
                neg_bigint(a1);//a1 = valor absoluto de a
        }
    //para manter b valor de a o copiamos em b1
        atrib_bigint(b1,b);
    //teste do sinal de b, dependendo de a, o sinal do quaciente
       pode ser positivo ou negativo
        if((b[0]&0x80)!=0){
                sinalQ*=-1;
                neg_bigint(b1);//b1 = valor absoluto de b
        }
   //inicia os auxiliares com 0
        init_bigint(aux1);
        init_bigint(aux2);
    //percorre todos os bytes de a1
        for(i=0;i<TBITS;i++){
                j=0x80;
        //para cada byte, percorre todos os seus bits
                while(j!=0){
            //dobra, ou seja, da shift a esquerda, os valores de
               aux1 e aux2
                        add_bigint(aux1,aux1,aux1);
                        add_bigint(aux2,aux2,aux2);
            //teste do bit desta volta do loop
                        if(a1[i]&j){
                                addi_bigint(aux2,aux2,1);
                        }
            //caso aux2 seja maior ou igual a b1, subtrai este de
               b1 e adiciona 1 a aux1
                        if(let_bigint(b1,aux2)){
                                addi_bigint(aux1,aux1,1);
                                sub_bigint(aux2,aux2,b1);
                        }
                        j=j>>1;
                }
        }
    //associa os valores de aux1 a d e de aux2 a m
        atrib_bigint(m,aux2);
```

```
44          atrib_bigint(d,aux1);
45      //faz o tratamento de sinais necessarios
46          if(sinalQ<0){
47                  neg_bigint(d);
48          }
49          if(sinalR<0){
50                  neg_bigint(m);
51          //substitui o resto negativo pelo seu menor positivo
                equivalente
52                  add_bigint(m,m,b1);
53          }
54  }
```

Este algoritmo ainda pode parecer abstrato, então será feito um exemplo.
Seja $a = 14$ o dividendo e $b = 3$ o divisor. O sistema 4.2 fica assim:

$$
\begin{gathered}
a = 14 = (1110)_2 / b = 3 = (11)_2 \\
a_bit_3 = 1 \Rightarrow aux2 \leftarrow 2 * aux2 + 1 = (1)_2 / aux1 \leftarrow 2 * aux1 = (0)_2 \\
aux2 < b \Rightarrow nada \\
aux1 = (0)_2; aux2 = (1)_2 \\
a_bit_2 = 1 \Rightarrow aux2 \leftarrow 2 * aux2 + 1 = (11)_2 / aux1 \leftarrow 2 * aux1 = (0)_2 \\
aux2 = b \Rightarrow aux1 \leftarrow aux1 + 1 / aux2 \leftarrow aux2 - b \\
aux2 = (0)_2; aux1 = (1)_2 \\
a_bit_1 = 1 \Rightarrow aux2 \leftarrow 2 * aux2 + 1 = (1)_2 / aux1 \leftarrow 2 * aux1 = (10)_2 \\
aux2 < b \Rightarrow nada \\
aux2 = (1)_2; aux1 = (10)_2 \\
a_bit_0 = 0 \Rightarrow aux2 \leftarrow 2 * aux2 = (10)_2 / aux1 \leftarrow 2 * aux1 = (100)_2 \\
aux2 < b \Rightarrow nada \\
aux2 = (10)_2; aux1 = (100)_2
\end{gathered}
\tag{4.2}
$$

Ou seja, $aux2 = m = (10)_2 = 2$ e $aux1 = d = (100)_2 = 4$, logo $14/3$ tem com resultado 4 e resto 2. Como esperado.

4.2.24 pow_modp_bigint

Nesta função temos 4 parâmetros do tipo *bigint* o primeiro ,denotado de r, será o que receberá o resultado da conta, o segundo, denotado de b, é a base da potência, o terceiro, denotado de e, é o expoente, e o quarto, denotado de p é o módulo.

Esta função tem como objetivo calcular r, onde $r \equiv b^e \pmod{p}$.

Para tal é usado um algoritmo similar ao usado na multiplicação, e que também é descrito em [13], o nome do método é *quadrado-e-mult* [10]. Análogo ao soma-e-dobra, utiliza-se a forma binária do expoente e inicia-se o resultado como 1. Para cada bit, começando do mais significativo é feito o seguinte:

- se o bit for 0, apenas eleva o resultado ao quadrado.
- se o bit for 1, eleva o resultado ao quadrado e multiplica pela base.

Dado que a base $b = 3$, o expoente $e = 5$ e o resultado seja r, se tem o exemplo 4.3:

$$\begin{gathered} e = 5 \Rightarrow e = (101)_2 \\ r = 1 \\ bit_2 = 1 \Rightarrow e \leftarrow (e * e) * b \Rightarrow e = 3 \\ bit_1 = 0 \Rightarrow e \leftarrow (e * e) \Rightarrow e = 9 \\ bit_0 = 1 \Rightarrow e \leftarrow (e * e) * b \Rightarrow e = 243 \end{gathered} \tag{4.3}$$

Ou seja, $3^5 = 243$, como esperado, diferente da multiplicação, já com os números pequenos, percebe-se que o algoritmo é bem mais otimizado que o jeito convencional.

Outro ponto importante é que se o número tratado é de 256 bits de tamanho, uma potência daria um número excessivamente grande, assim para otimizar nossas contas é feito o cálculo do módulo p a cada multiplicação do algoritmo. O cálculo do módulo em questão é o resto devolvido pela função **div_mod_bigint** já explicada na seção 4.2.23.

O código final fica assim:

```
1  void pow_modp_bigint(bigint r,bigint b,bigint e,bigint p){
2          bigint trash,aux;
3          int i,j;
4      //inicia o resultado com 1
5          atribi_bigint(aux,1);
6      //posiciona o expoente no byte mais significativo
7          for(i=0;i<TBITS&&b[i]==0;i++);
8      //percorre byte a byte o expoente a partir da posicao
9          for(;i<TBITS;i++){
10         //cada byte e percorrido bit a bit
11                 j=0x80;
12                 while(j!=0){
13             //para cada bit se procede o algoritmo square-and-mult
14                         mult_bigint(aux,aux,aux);
15             //calcula-se o resultado modulo p a cada operacao de
                   multiplicacao
16                         div_mod_bigint(trash,aux,aux,p);
17                         if(e[i]&j){
```

[10] tradução livre de square-and-mult

```
18                                  mult_bigint(aux,aux,b);
19                                  div_mod_bigint(trash,aux,aux,p);
20                          }
21                          j=j>>1;
22                  }
23          }
24          div_mod_bigint(trash,aux,aux,p);
25      //atribui-se o resultado ao parametro r
26          atrib_bigint(r,aux);
27  }
```

Com isso encerra-se o módulo de definição e operação com números de 256 bits, denominado de *bigint*.

4.3 sha256.h

Este módulo é, especificamente, para trabalhar com a função de resumo criptográfico SHA-256.

O conceito de resumo criptográfico já foi abordado na seção 2.2.4, onde, também foi mostrada a função SHA-1. Porém explicamos que o SHA-1, além de ter tido ataques efetivos contra ele, nos dá um resumo de 160 bits, e o que queremos é uma função que nos de 256 bits de resumo. A família SHA-2, da qual o SHA-256 faz parte, possui várias funções, cada uma com um número de bits na saída.

Os sistemas ECDSA de 256 bits, em sua maioria, utilizam o SHA-256. E como a implementação que se quer gerar é um simulador de tal assinatura, faz-se necessária a implementação de tal função de resumo.

Este módulo é arquitetado da seguinte forma:

É importado o módulo *bigint.h*, pois as contas serão baseadas em inteiros de 256 bits.

Não há tipos definidos neste módulo, mas as funções que temos são:

- **rightRot** - Rotaciona Lógicamente um número n bits para a direita, sendo n um dos parâmetros da função
- **sha256** - Processa o loop principal do algoritmo do sha-256, devolvendo o resumo criptográfico gerado
- **arq_sha256** - faz o pré-processamento de uma mensagem em um arquivo e chama a função **sha256**
- **tex_sha256** - faz o pré-processamento de uma mensagem de uma string e chama a função **sha256**

Antes de começar a descrever as funções, é necessário explicar as etapas do SHA-256.

Assim como a função SHA-1 descrita na seção 2.2.5 o SHA-256 tem um pré-processamento da mensagem e em seguida um loop que irá trabalhar com blocos de tamanho pré-definido da mensagem, usando constantes pré-selecionadas e funções pré-definidas. A origem destas seleções e definições não são importantes para este documento, apenas importante frisar que não foram escolhidas aleatoriamente e existe uma matemática complexa em cada passo deste algoritmo.

Na função **sha256** será feito o loop com a mensagem já pré-processada, nas funções **arq_sha256**e **tex_sha256** parte do tratamento citado é o pré-processamento.

4.3.1 rightRot

Esta função é bem simples, mas não há tal função nas bibliotecas padrões do GCC. Ela simplesmente rotaciona os bits de um número no sentido do bit menos significativo. Na verdade não passa de um shift, onde o bit que seria descartado é reposicionado no "espaço deixado pela operação shift". O código fica assim:

```
unsigned int rightRot(unsigned int x,unsigned int n){
        unsigned int y;
    //y sera a concatenacao dos n bits que seriam descartados no
       shift com o proprio x dado um shift de n bits
        y = (x>>n)|(x<<(32-n));
        return y;
}
```

4.3.2 arq_sha256

Esta função recebe como parâmetro um ponteiro para um arquivo e um *bigint* onde será armazenado o resumo criptográfico do arquivo. Primeiramente, lê-se o arquivo já iniciando o pré-processamento.

O pré-processamento independente da origem da mensagem funciona da seguinte maneira:

- coloca-se um bit 1 no final da mensagem.
- em seguida coloca-se k bits 0 na frente da mensagem, onde k é o menor número positivo que faça $k + 1 + l \equiv 448 \pmod{512}$, onde l é o tamanho original da mensagem.
- acrescente a frente de tudo isso 64 bits que contenham a forma binária do tamanho da mensagem.

Como exemplo será usada a mensagem "abc". Se tem então o apresentado na tabela 4.4

Tabela 4.4: Pre-processamento do SHA-256

abc	1+k(0)	tamanho(64bits)
01100001 01100010 01100011	10...0	0...011000

Assim que o este processo denominado de *padding* está feito a mensagem é dividida em blocos de 512 bits cada para iniciar o loop.

Como dito está função faz o pré-processamento enquanto lê os bytes do arquivo. Isto é feito da seguinte maneira: primeiro calcula-se o tamanho do arquivo, depois calcula-se qual será o tamanho da mensagem pré-processada. Aloca-se um buffer deste tamanho e se insere byte a byte a mensagem do arquivo no buffer. então fazem-se as adições descritas

acima no pré-processamento. A parte de dividir em blocos de 512 bits fica por parte da função **sha256**.

O código fica assim:

```
void arq_sha256(FILE *fp,bigint h_x){
        unsigned long long int tamanho,tpand,bits,i;
        unsigned char* buffer;
        //obter tamanho do arquivo
        fseek(fp,0,SEEK_END);
        tamanho = ftell(fp);
        rewind(fp);
    //calcular tamanho apos pre-processo
        tpand = tamanho;
        if(tpand%64==56){
                tpand++;
        }
        while(tpand%56!=0){
                tpand++;
        }
        tpand+=8;
    //aloca o buffer com tamanho exato necessario
        buffer = (unsigned char*) malloc (sizeof(unsigned char)*(
           tpand));
    //inseri mensagem
        fread(buffer,1,tamanho,fp);
    //inseri o 1 e os primeiros sete 0s
        buffer[tamanho] = 0x80;
    //insere o resto dos k 0s
        for(i=tamanho+1;i<tpand-8;i++){
                buffer[i] = 0x00;
        }
    //inseri os 64bits que contem o tamanho original do arquivo
        bits = tamanho*8;
        buffer[tpand-8] = bits>>56;
        buffer[tpand-7] = (bits>>48)&0xff;
        buffer[tpand-6] = (bits>>40)&0xff;
        buffer[tpand-5] = (bits>>32)&0xff;
        buffer[tpand-4] = (bits>>24)&0xff;
        buffer[tpand-3] = (bits>>16)&0xff;
        buffer[tpand-2] = (bits>>8)&0xff;
        buffer[tpand-1] = bits&0xff;
    //envia mensagem pre-processada para a funcao de hash
        sha256(buffer,tpand,h_x);
}
```

4.3.3 tex_sha256

Esta função é similar a anterior, só que ao invés de receber como parâmetro um ponteiro de arquivo, recebe um vetor de caracteres. é utilizada a função **strlen** da *string.h* para

obter o tamanho do arquivo, é feito os mesmos cálculos e aloca-se o buffer da mesma maneira. Insere-se byte a byte o que está no vetor para o buffer, completando com os bytes necessários, da mesma forma feita na função anterior.

Assim, conclui-se desnecessário a apresentação do código, visto que não muda muito comparado ao anterior.

4.3.4 sha256

Como dito anteriormente o algoritmo do SHA-256 é feito de fórmulas e constantes pré-selecionadas, e que a matemática desta seleção é muito complexa, além de fugir do foco deste documento. Então apenas serão apresentadas as fórmulas e constantes.

O loop é feito n vezes, onde $n = tamanho/512$, em outras palavras é o número de blocos de 512 bits obtidos da mensagem pré-processada. Em cada loop calcula-se um vetor de inteiros sem sinais $w[64]$ de 64 posições. O cálculo deste vetor é dado pela regra mostrada em 4.4, onde $\circ$ é concatenação, $RotR(x, n)$ é rotação à direita de x em n bits e $ShiftR(x, n)$ é shift à direita de x em n bits.

$$
\begin{aligned}
W[j] &\leftarrow M[4*j] \circ M[4*j+1] \circ M[4*j+2] \circ M[4*j+3]; \forall 0 \leq j \leq 15 \\
S_0^{(j)} &\leftarrow RotR(W[j-15], 7) \wedge RotR(W[j-15], 18) \wedge ShiftR(W[j-15], 3); \forall 16 \leq j \leq 63 \\
S_1^{(j)} &\leftarrow RotR(W[j-2], 17) \wedge RotR(W[j-2], 19) \wedge ShiftR(W[j-2], 10); \forall 16 \leq j \leq 63 \\
W[j] &\leftarrow W[j-16] + S_0^{(j)} + W[j-17] + S_1^{(j)}; \forall 16 \leq j \leq 63
\end{aligned}
\tag{4.4}
$$

Inicia-se, então, as variáveis a, b, c, d, e, f, g, h com os valores de $h0, h1, h2, h3, h4, h5, h6, h7$ respectivamente. Os primeiros valores de $h0, ..., h7$ são mostrados abaixo:

- $h0 = 0x6a09e667$
- $h1 = 0xbb67ae85$
- $h2 = 0x3c6ef372$
- $h3 = 0xa54ff53a$
- $h4 = 0x510e527f$
- $h5 = 0x9b05688c$
- $h6 = 0x1f83d9ab$
- $h7 = 0x5be0cd19$

Por fim repete-se 64 vezes as seguintes contas, usando a mesma nomenclatura das equações 4.4, além de um vetor de constantes inteiras denominado $k[]$ que terão seus valores apresentados junto com o código.

- $s1 \leftarrow RightR(e, 6) \wedge RightR(e, 11) \wedge RightR(e, 25)$

- $ch \leftarrow (e\&f) \wedge (\neg e\&g)$
- $t1 \leftarrow h + s1 + ch + k[j] + W[j]$
- $s0 \leftarrow RightR(a, 2) \wedge RightR(a, 13) \wedge RightR(a, 21)$
- $maj \leftarrow (a\&b) \wedge (a\&c) \wedge (b\&c)$
- $t2 \leftarrow s0 + maj$
- $h \leftarrow g$
- $g \leftarrow f$
- $f \leftarrow e$
- $e \leftarrow d + t1$
- $d \leftarrow c$
- $c \leftarrow b$
- $b \leftarrow a$
- $a \leftarrow t1 + t2$

Depois disso os valores de $a, ..., h$ são somados respectivamente nos valores de $h0, ..., h7$, e se repete tudo, desde a criação do vetor W para os outros blocos.

No final o resumo criptográfico é a concatenação de $h0, ..., h7$. Um número de 256 bits, que a função atribui ao parâmetro h_x e escreve em um arquivo denominado de *h_x.txt*.

O código é este:

```
void sha256(unsigned char *buffer,long long int tpand,bigint h_x){
        unsigned long long int i;
        //iniciando variaveis
        unsigned int h0,h1,h2,h3,h4,h5,h6,h7;
        h0 = 0x6a09e667;
        h1 = 0xbb67ae85;
        h2 = 0x3c6ef372;
        h3 = 0xa54ff53a;
        h4 = 0x510e527f;
        h5 = 0x9b05688c;
        h6 = 0x1f83d9ab;
        h7 = 0x5be0cd19;
        unsigned int k[64] = {0x428a2f98, 0x71374491, 0xb5c0fbcf,
                          0xe9b5dba5, 0x3956c25b, 0x59f111f1,
                          0x923f82a4, 0xab1c5ed5, 0xd807aa98,
                          0x12835b01, 0x243185be, 0x550c7dc3,
                          0x72be5d74, 0x80deb1fe, 0x9bdc06a7,
                          0xc19bf174, 0xe49b69c1, 0xefbe4786,
                          0x0fc19dc6, 0x240ca1cc, 0x2de92c6f,
                          0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
                          0x983e5152, 0xa831c66d, 0xb00327c8,
                          0xbf597fc7, 0xc6e00bf3, 0xd5a79147,
```

```
23                              0x06ca6351, 0x14292967, 0x27b70a85,
24                              0x2e1b2138, 0x4d2c6dfc, 0x53380d13,
25                              0x650a7354, 0x766a0abb, 0x81c2c92e,
26                              0x92722c85, 0xa2bfe8a1, 0xa81a664b,
27                              0xc24b8b70, 0xc76c51a3, 0xd192e819,
28                              0xd6990624, 0xf40e3585, 0x106aa070,
29                              0x19a4c116, 0x1e376c08, 0x2748774c,
30                              0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a,
31                              0x5b9cca4f, 0x682e6ff3, 0x748f82ee,
32                              0x78a5636f, 0x84c87814, 0x8cc70208,
33                              0x90befffa, 0xa4506ceb, 0xbef9a3f7,
34                              0xc67178f2};
35          unsigned long long int numW,j;
36          unsigned int w[64],s0,s1,S1,S0,ch,maj,temp1,temp2;
37          unsigned int a,b,c,d,e,f,g,h;
38      //numW e o numero de blocos de 512 bits da mensagem pre-
           processada
39          numW = tpand/64;
40      //loop roda uma vez para cada bloco
41          for(i=0;i<numW;i++){
42          //criacao do vetor W
43                  for(j=0;j<16;j++){
44                          w[j] = (buffer[64*i+4*j]<<24)+(buffer[64*i
                               +4*j+1]<<16)+(buffer[64*i+4*j+2]<<8)+
                               buffer[64*i+4*j+3];
45                  }
46                  for(;j<64;j++){
47                          s0 = (rightRot(w[j-15],7))^(rightRot(w[j
                               -15],18))^(w[j-15]>>3);
48                          s1 = (rightRot(w[j-2],17))^(rightRot(w[j
                               -2],19))^(w[j-2]>>10);
49                          w[j] = w[j-16]+s0+w[j-7]+s1;
50                  }
51          //inicializacao do a,...,h
52                  a = h0;
53                  b = h1;
54                  c = h2;
55                  d = h3;
56                  e = h4;
57                  f = h5;
58                  g = h6;
59                  h = h7;
60          //processamento do bloco
61                  for(j=0;j<64;j++){
62                          S1 = (rightRot(e,6))^(rightRot(e,11))^(
                               rightRot(e,25));
63                          ch = (e&f)^((~e)&g);
64                          temp1 = h+S1+ch+k[j]+w[j];
```

```
65                          S0 = (rightRot(a,2))^(rightRot(a,13))^(
                               rightRot(a,22));
66                          maj = (a&b)^(a&c)^(b&c);
67                          temp2 = S0+maj;
68                          h = g;
69                          g = f;
70                          f = e;
71                          e = d+temp1;
72                          d = c;
73                          c = b;
74                          b = a;
75                          a = temp1+temp2;
76                  }
77          //adiciona o resultado do processo do bloco
78                  h0 += a;
79                  h1 += b;
80                  h2 += c;
81                  h3 += d;
82                  h4 += e;
83                  h5 += f;
84                  h6 += g;
85                  h7 += h;
86          }
87      //terminado o processo, imprime o resultado em arquivo e
           atribui o mesmo resultado no parametro h_x
88          FILE *fp2;
89          init_bigint(h_x);
90          append_int_bigint(h_x,h0);
91          append_int_bigint(h_x,h1);
92          append_int_bigint(h_x,h2);
93          append_int_bigint(h_x,h3);
94          append_int_bigint(h_x,h4);
95          append_int_bigint(h_x,h5);
96          append_int_bigint(h_x,h6);
97          append_int_bigint(h_x,h7);
98          fp2 = fopen("h_x.txt","w");
99          fimprime_hex_bigint(h_x,fp2);
100         fclose(fp2);
101 }
```

4.4 Algebra_op.h

Nesta seção será abordada classe estática *Algebra_op* na qual estão definidas as operações de teoria dos números necessárias para trabalhar com o protocolo ECDSA que funcionam para o tipo criado nestes módulos, o *bigint*.

Este módulo é arquitetado da seguinte forma:

É importado o módulo *basics.h*, o qual importa o *bigint.h*.

Neste módulo, como dito, é definida a classe estática *Algebra_op*, esta não possui atributos, e seus métodos são:

- **gcd** - Calcula o mdc entre dois *bigint*
- **EEA** - Calcula o mdc entre dois *bigint* e as constantes que formam sua combinação linear
- **inverse_eea** - Calcula a inversa multiplicativa de um *bigint* módulo outro

No início será abordado o método **EEA** onde será implementado o algoritmo extendido de Euclides, logo após será explicado o método **gcd**, e por fim, o método **inverse_eea**.

4.4.1 EEA

Na seção 2.1.2, houve a explicação de como funciona o algoritmo extendido de Euclides, neste algoritmo calcula-se o maior divisor comum entre dois números a e b ao mesmo tempo que calcula-se as constantes s e t que formam uma combinação linear tal que $a * s + b * t = mdc(a, b)$.

Este método recebe como parâmetros 5 variáveis do tipo *bigint*, o primeiro, denominado de *eea* receberá o valor do mdc, o segundo e o terceiro, denominados respectivamente de a e b, são os números que serão usados para calcular o mdc, por fim os dois últimos, denominados s e t, respectivamente, receberam os valores das constantes da combinação linear.

O pseudo-código que descreve o passo a passo do algoritmo e esta em [13] é:

Algorithm 2 Algoritmo Extendido de Euclides

1: Dado: a,b
2: Resultado: mdc,s,t
3: $s_0 \leftarrow 1$
4: $t_0 \leftarrow 0$
5: $s_1 \leftarrow 0$
6: $t_1 \leftarrow 1$
7: $i \leftarrow 1$
8: **repeat**
9: $\quad i \leftarrow i + 1$
10: $\quad r_i \leftarrow r_{i-2} \pmod{r_{i-1}}$
11: $\quad q_{i-1} \leftarrow (r_{i-2} - r_i)/r_{i-1}$
12: $\quad s_i \leftarrow s_{i-2} - q_{i-1} * s_{i-1}$
13: $\quad t_i \leftarrow t_{i-2} - q_{i-1} * t_{i-1}$
14: **until** $r_i \neq 0$
15: $mdc \leftarrow r_{i-1}$
16: $s \leftarrow s_{i-1}$
17: $t \leftarrow t_{i-1}$

O código gerado é este:

```
void Algebra_op::EEA(bigint eea,bigint a,bigint b,bigint s,bigint t
   ){
        bigint si,si_1,si_2;
        bigint ti,ti_1,ti_2;
        bigint ri,ri_1,ri_2;
        bigint q,trash;
    //atribuicao dos valores iniciais necessarios para o algoritmo
        atribi_bigint(si_1,1);
        atribi_bigint(ti_1,0);
        atribi_bigint(si,0);
        atribi_bigint(ti,1);
        atrib_bigint(ri_1,a);
        atrib_bigint(ri,b);
    //recursao do algoritmo de euclides
        do{
                atrib_bigint(si_2,si_1);
                atrib_bigint(si_1,si);
                atrib_bigint(ti_2,ti_1);
                atrib_bigint(ti_1,ti);
                atrib_bigint(ri_2,ri_1);
                atrib_bigint(ri_1,ri);
                div_mod_bigint(trash,ri,ri_2,ri_1);
                sub_bigint(q,ri_2,ri);
                div_mod_bigint(q,trash,q,ri_1);
                mult_bigint(trash,q,si_1);
                sub_bigint(si,si_2,trash);
                mult_bigint(trash,q,ti_1);
                sub_bigint(ti,ti_2,trash);
        }while(not_null(ri));
    //atribuicao dos valores de s t e eea
        atrib_bigint(s,si_1);
        atrib_bigint(t,ti_1);
        atrib_bigint(eea,ri_1);
}
```

4.4.2 gcd

Este método recebe 3 parâmetros do tipo *bigint*, o primeiro receberá o mdc entre os outros dois parâmetros. Para tal simplesmente é necessário chamar o método **EEA** desconsiderando o que voltar pelos parâmetros s e t, mas pela implementação feita, é necessário que ao chamar, seja colocado o menor parâmetro ao qual se quer o mdc na terceira posição e o maior na segunda, como mostra o código.

```
void Algebra_op::gcd(bigint gcd,bigint a, bigint b){
        bigint tmp1,tmp2;
    //testa qual o menor entre a e b e chama da maneira correta
        if(lt_bigint(b,a)){
                Algebra_op::EEA(gcd,a,b,tmp1,tmp2);
        }else{
```

```
                Algebra_op::EEA(gcd,b,a,tmp1,tmp2);
        }
}
```

4.4.3 inverse_eea

Neste método, assim como o anterior, só é necessário chamar o método **EEA**, mas desta vez deve-se armazenar o que voltar do parâmetro referente a inversa do número que se quer. Caso você queira a inversa de a módulo b, a inversa será o s, se for o contrário, a inversa será o t.

Os parâmetros deste método são 3 do tipo *bigint*, o primeiro, *inv*, armazenará a inversa do segundo, a, módulo o terceiro, n.

Para isso primeiro é testado se $a \leq n$, caso seja, passa-se para a variável af o valor de a, senão, passa-se o valor de $a \pmod{n}$. Então chama-se a função **EEA**. Caso o mdc retornado seja diferente de 1, a função retorna como inversa o valor 0, que será tratado como erro. Caso contrário é testado se a inversa é negativa, se for, apenas substitui-se o valor pelo menor positivo equivalente módulo n.

O código fica assim:

```
void Algebra_op::inverse_eea(bigint inv,bigint a,bigint n){
        bigint af;
        bigint gcd,t,trash;
    //testa se a e menor que n.
        if(lt_bigint(a,n)){//caso seja utilizamos o proprio valor
           de a
                atrib_bigint(af,a);
        }else{//senao, usamos o modulo n do valor de a
                div_mod_bigint(trash,af,a,n);
        }
    //chamada do metodo EEA
        Algebra_op::EEA(gcd,n,af,t,inv);
    //teste se existe inversa
        if(!eqi_bigint(gcd,1)){
                init_bigint(inv);
                return;
        }
    //testa se a inversa esta na forma negativa e transforma caso
       esteja.
        atribi_bigint(trash,0);
        if(lt_bigint(inv,trash)){
                add_bigint(inv,inv,n);
        }
}
```

4.5 EC_Point.h

Este módulo é arquitetado da seguinte forma:

É incluído o módulo *basics.h*.

É definida a classe *EC_Point*, esta classe representa um ponto de uma curva elíptica, seus atributos são os pontos x e y, as constantes que definem a curva a e b e o primo na qual o campo finito está definido p.

Para cada atributo existe um par de métodos **set** e **get**, que respectivamente, atribuem e devolvem o valor de tal atributo. Os outros métodos são:

- **EC_Point** - Construtor, existem três diferentes, um constrói o ponto todo com valores zero, outro recebe o valor de cada atributo em valores inteiros e constrói o ponto caso estes valores representem uma curva válida, e o recebe o valores *bigint* e age de maneira semelhante ao segundo.
- **atribuir** - Copia os atributos de um outro ponto para este que chama o método.

4.5.1 Construtores

Como dito, existem três construtores, no primeiro, quando não há parâmetros, o que ocorre é que todos os atributos são atribuídos com o valor 0. Nos outros dois em que a única diferença é o tipo do atributo o que é feito é atribuir os valores passados pelo parâmetro em seus respectivos atributos. Porém, caso os atributos não definam corretamente o ponto e a curva onde ele se encontra todos os atributos se tornam zero.

No caso do construtor que recebe 5 inteiros, primeiro estes são atribuídos a tipos *bigint* e depois com estes novos, prossegue o mesmo procedimento do construtor que recebe 5 parâmetros *bigint*. Como são praticamente iguais o código que será mostrado será apenas do construtor que recebe *bigint*.

```
1  EC_Point::EC_Point(bigint x,bigint y,bigint a,bigint b,bigint p){
2          bigint aux1,aux2,trash;
3      //associa os parametros a,b,p nos seus respectivos atributos
4          atrib_bigint(this->a,a);
5          atrib_bigint(this->b,b);
6          atrib_bigint(this->p,p);
7      //conta necessaria para testar se o ponto pertence a curva
8          mult_bigint(aux1,x,x);
9          div_mod_bigint(trash,aux1,aux1,p);
10         mult_bigint(aux1,aux1,x);
11         div_mod_bigint(trash,aux1,aux1,p);
12         mult_bigint(aux2,x,a);
13         div_mod_bigint(trash,aux2,aux2,p);
14         add_bigint(aux2,aux2,b);
15         add_bigint(aux1,aux1,aux2);
16         div_mod_bigint(trash,aux1,aux1,this->p);
17         mult_bigint(aux2,y,y);
18         div_mod_bigint(trash,aux2,aux2,this->p);
19     //caso o ponto esteja atribui os valores x,y nos atributos
           correspondentes
20         if(eq_bigint(aux1,aux2)){
21                 atrib_bigint(this->x,x);
22                 atrib_bigint(this->y,y);
```

```
        }else{//caso contrario, todos os atributos sao zerados
                init_bigint(this->a);
                init_bigint(this->b);
                init_bigint(this->x);
                init_bigint(this->y);
                init_bigint(this->p);
        }
}
```

4.5.2 atribuir

Este método é bem simples, um objeto que chama este método, recebe como parâmetro outro ponto definido em uma curva, então o objeto simplesmente se torna uma cópia do ponto passado como parâmetro. Isto é feito com simples atribuição, como mostrado no código:

```
void EC_Point::atribuir(EC_Point P){
        bigint aux;
    //um a um copia o valor dos atributos de P para o objeto
        P.get_x(aux);
        atrib_bigint(this->x,aux);
        P.get_y(aux);
        atrib_bigint(this->y,aux);
        P.get_a(aux);
        atrib_bigint(this->a,aux);
        P.get_b(aux);
        atrib_bigint(this->b,aux);
        P.get_p(aux);
        atrib_bigint(this->p,aux);
}
```

4.5.3 Métodos get e set

Os métodos *get* e *set* são nativos da orientação a objeto, são meios de ter acesso aos atributos privados de um objeto, não é diferente neste caso. Cada atributo tem seu par de *get* e *set*, como exemplo será exposto o código referente ao atributo a:

```
void set_a(bigint a){
        atrib_bigint(this->a,a);
}
void get_a(bigint a){
        atrib_bigint(a,this->a);
}
```

Como visto, no **set_a** você passa um valor *bigint* como parâmetro e este será associado ao atributo a do objeto. No **get_a** você manda uma variável tipo *bigint* como parâmetro e ela retornará com o valor do atributo a do objeto.

Esta é a classe *EC_Point*, a seguir teremos a classe *EC_op* onde as operações descritas na seção 2.3.1 serão implementadas.

4.6 EC_op.h

A arquitetura deste módulo é a seguinte:

É incluído o módulo *EC_Point.h*, o qual possui a classe *EC_Point*, que define um ponto numa curva elíptica.

É definida a classe estática *EC_op*, esta classe contém as três operações em curvas elípticas descritas na seção 2.3, são elas a soma entre dois pontos distintos, a soma de um ponto com si mesmo, e a multiplicação de um ponto por um número.

A classe não possui atributos, mas seus métodos são:

- **add** - Soma dois objetos *EC_Point*
- **dobrar** - Dobra um objeto *EC_Point*
- **mult** - multiplica um inteiro e um objeto *EC_Point*

4.6.1 add

Este método recebe 2 objetos da classe *EC_Point* como parâmetros, e retorna um objeto R que será a soma deles.

Primeiro é testado se o atributo p de um dos pontos é 0, caso seja, na implementação é tratado como o ponto no infinito, que é o elemento neutro da operação de soma, logo o resultado será exatamente o outro ponto.

Em seguida é testado se os atributos x de cada ponto são iguais, se sim, há duas possibilidades. Caso os y dos pontos também forem iguais, então o resultado é uma dobra de um ponto só, assim, é chamada o método **dobrar**, se os y forem diferentes então a soma dos dois dará um ponto no infinito, logo o resultado será um objeto com todos os atributos zerados.

Por fim, caso tanto os atributos x como os y forem diferentes, é feito as contas descritas na seção 2.3.1. O código é este:

```
1  EC_Point EC_op::add(EC_Point P,EC_Point Q){
2          bigint s,x,y,p,aux1,aux2,aux3,trash;
3      //testando se nenhum dos pontos e ponto neutro
4          P.get_p(p);
5          if(!not_null(p)){
6                  return Q;
7          }
8          Q.get_p(p);
9          if(!not_null(p)){
10                 return P;
11         }
12     //testando se os x deles sao iguais
13         P.get_x(aux1);
14         Q.get_x(aux2);
15         if(eq_bigint(aux1,aux2)){
16         //caso sejam testa os y
17                 P.get_y(aux1);
18                 Q.get_y(aux2);
```

```
19                  if(eq_bigint(aux1,aux2)){
20              //caso tambem seja P e igual a Q logo P+Q e igual a 2P
21                          return dobrar(P);
22                  }
23          //caso os y deles diferem o resultado sera um ponto no
               infinito
24                  EC_Point R;
25                  return R;
26          }
27      //caso tanto x quanto y deles forem diferentes continua a conta
28          Q.get_y(aux1);
29          P.get_y(aux2);
30      //isto e feito para que a subtracao ja esteja no residuo modulo
            p padrao
31          if(lt_bigint(aux1,aux2)){
32                  add_bigint(aux1,aux1,p);
33          }
34          sub_bigint(aux1,aux1,aux2);
35          Q.get_x(aux2);
36          P.get_x(aux3);
37      //isto e feito para que a subtracao ja esteja no residuo modulo
            p padrao
38          if(lt_bigint(aux2,aux3)){
39                  add_bigint(aux2,aux2,p);
40          }
41          sub_bigint(aux2,aux2,aux3);
42      //continua o calculo de s
43          Algebra_op::inverse_eea(aux2,aux2,p);
44          mult_bigint(aux1,aux1,aux2);
45          div_mod_bigint(trash,s,aux1,p);
46      //Inicio do calculo de x
47          mult_bigint(aux1,s,s);
48          div_mod_bigint(trash,aux1,aux1,p);
49          P.get_x(aux2);
50          if(lt_bigint(aux1,aux2)){
51                  add_bigint(aux1,aux1,p);
52          }
53          sub_bigint(aux1,aux1,aux2);
54          Q.get_x(aux2);
55          if(lt_bigint(aux1,aux2)){
56                  add_bigint(aux1,aux1,p);
57          }
58          sub_bigint(x,aux1,aux2);
59      //final do calculo de x
60      //inicio do calculo de y
61          P.get_x(aux1);
62          if(lt_bigint(aux1,x)){
63                  add_bigint(aux1,aux1,p);
64          }
```

```
        sub_bigint(aux1,aux1,x);
        mult_bigint(aux1,aux1,s);
        div_mod_bigint(trash,aux1,aux1,p);
        P.get_y(aux2);
        if(lt_bigint(aux1,aux2)){
                add_bigint(aux1,aux1,p);
        }
        sub_bigint(y,aux1,aux2);
    //fim do calculo de y
    //define o ponto resultado no objeto R
        P.get_a(aux1);
        P.get_b(aux2);
        EC_Point R(x,y,aux1,aux2,p);
    //retorna R
        return R;
}
```

4.6.2 dobrar

Este método recebe 1 objeto da classe *EC_Point* e retorna outro objeto R que será o resultado da conta $2 * P$, onde P é o parâmetro.

Assim como o anterior, este algoritmo já foi explicado na seção 2.3.1. Mas existem pontos a serem explicados.

Primeiro, é testado se o P é um ponto do infinito, ou seja, se o atributo p dele é 0. Caso seja o resultado também será um ponto no infinito.

Depois é testado se o y de P é 0. caso seja, como visto anteriormente seu dobro também dará num ponto no infinito. Se nada disso ocorrer apenas é feito a matemática de curva elíptica descrita na seção 2.3.1 para dobrar um ponto.

O código fica assim:

```
EC_Point EC_op::dobrar(EC_Point P){
        bigint x,y,s,aux1,aux2,p,trash;
    //testa se P nao e ponto no infinito
        P.get_p(p);
        if(!not_null(p)){
                return P;
        }
    //testa se y de P nao e 0
        P.get_y(aux1);
        if(!not_null(aux1)){
                EC_Point R;
                return R;
        }
    //caso os testes anteriores sejam falsos continua a conta
    //inicio do calculo de s
        P.get_x(aux1);
        mult_bigint(aux1,aux1,aux1);
        div_mod_bigint(trash,aux1,aux1,p);
        multi_bigint(aux1,aux1,3);
```

```
20          P.get_a(aux2);
21          add_bigint(aux1,aux1,aux2);
22          P.get_y(aux2);
23          multi_bigint(aux2,aux2,2);
24          div_mod_bigint(trash,aux2,aux2,p);
25          Algebra_op::inverse_eea(aux2,aux2,p);
26          mult_bigint(aux1,aux1,aux2);
27          div_mod_bigint(trash,s,aux1,p);
28      //fim do calculo de s
29      //inicio do calculo de x
30          mult_bigint(aux1,s,s);
31          div_mod_bigint(trash,aux1,aux1,p);
32          P.get_x(aux2);
33          sll1_bigint(aux2);
34          div_mod_bigint(trash,aux2,aux2,p);
35          if(lt_bigint(aux1,aux2)){
36                  add_bigint(aux1,aux1,p);
37          }
38          sub_bigint(x,aux1,aux2);
39      //fim do calculo de x
40      //inicio do calculo de y
41          P.get_x(aux1);
42          P.get_y(aux2);
43          if(lt_bigint(aux1,x)){
44                  add_bigint(aux1,aux1,p);
45          }
46          sub_bigint(aux1,aux1,x);
47          mult_bigint(aux1,aux1,s);
48          div_mod_bigint(trash,aux1,aux1,p);
49          if(lt_bigint(aux1,aux2)){
50                  add_bigint(aux1,aux1,p);
51          }
52          sub_bigint(y,aux1,aux2);
53      //fim do calculo de y
54      //cria-se o objeto R com os parametros calculados
55          P.get_a(aux1);
56          P.get_b(aux2);
57          EC_Point R(x,y,aux1,aux2,p);
58      //retorna R
59          return R;
60  }
```

4.6.3 mult

Este método recebe como parâmetro 1 objeto do tipo *EC_Point*, denominado P, e um do tipo *bigint*, denominado num, e retorna o objeto resultado da conta $num * P$. Para este método, o algoritmo usado é semelhante ao usado na multiplicação de *bigint*, o double-and-sum, descrito na seção 4.2.22. Mas primeiro, é necessário testar se o ponto P

é um ponto no infinito e se o num é nulo. Caso uma das coisas aconteça o resultado é um ponto no infinito.

Caso contrário, é possível continuar com o algoritmo double-and-sum, o código fica assim:

```
EC_Point EC_op::mult(EC_Point P,bigint num){
        unsigned int i,j;
        bigint aux1;
        EC_Point R;
    //testa se o ponto P e ponto no infinito ou se num e igual 0
        P.get_p(aux1);
        if(!not_null(num)||!not_null(aux1)){
        //caso seja retorna um ponto no infinito
                return R;
        }
    //comeca o algoritmo double-and-sum.
    //posiciona o byte mais significativo nao nulo de num
        for(i=0;num[i]==0;i++);
    //da posicao atual ate o final, percorre de byte a byte o num
        for(;i<TBITS;i++){
        //para cada byte percorre todos os bits
                j=0x80;
                while(j!=0){
            //se o bit for 0, somente dobra
            //se o bit for 1 dobra e soma
                        R.atribuir(dobrar(R));
                        if(num[i]&j){
                                R.atribuir(add(R,P));
                        }
                        j=j>>1;
                }
        }
    //retorna o resultado
        return R;
}
```

4.7 ECDSA.h

Este módulo é arquitetado da seguinte forma:

São incluídos os módulos *EC_op.h*, que possui as operações da matemática de curvas elípticas, e *sha256.h*, que possui a função de resumo criptográfico SHA-256.

É definida a classe estática *ECDSA*, a classe principal deste projeto, onde é implementado cada processo do protocolo ECDSA. Como visto no capítulo anterior o protocolo consiste de três funções, geração de chaves, geração de assinatura e verificação de assinatura, os métodos descritos nesta classe implementam cada uma destas funções.

Neste módulo, além da classe é definido dois registros [11], são eles:

[11] em inglês conhecido como struct, tipo de variável composta por outras variáveis.

- *pub_key* - define os elementos de uma chave pública.
- *signature* - define os elementos de uma assinatura digital.

A classe *ECDSA* não possui atributos, mas seus métodos são:

- **key_generator** - Gera o par de chave, pública e privada.
- **sign** - assina um conjunto de bytes
- **verify** - verifica uma assinatura, retornando verdadeiro caso válida e falso caso contrário

Primeiro será descrito os registros, depois será abordado os métodos da classe, começando do método **key_generator**, em seguida do **sign** e por fim o método **verify**.

4.7.1 Registros pub_key e signature

Os registros *pub_key* e *signature* representam, respectivamente, uma chave pública e uma assinatura dentro do protocolo do ECDSA. Como visto no capítulo anterior, neste protocolo, a chave privada é um número de 256 bits, logo será representada simplesmente por uma variável do tipo *bigint*, a chave pública, por sua vez, possui todas as informações da curva, além dos pontos que são utilizados na verificação, apesar de já ter sido mostrado, vale a pena recordar o conteúdo da chave pública, são estes:

- O módulo p do campo finito.
- O coeficiente a da curva elíptica
- O coeficiente b da curva elíptica
- A ordem q da curva
- O ponto gerador A do grupo cíclico
- O ponto $B = d * A$, onde d é a chave privada

Porém, como visto, entre os atributos da classe *EC_Point* estão os valores p, a, b da curva. Logo, o registro que representará as chaves públicas terá como variáveis: duas do tipo *EC_Point*, que representaram os pontos A e B, e um do tipo *bigint*, que representará a ordem do grupo cíclico. Assim se tem a seguinte montagem:

```
typedef struct _pub_key{
        bigint order;//ordem do grupo ciclico
        EC_Point A;//ponto gerador do grupo
        EC_Point B;//ponto B=d*A
}pub_key;
```

O registro *signature*, por sua vez, representa uma assinatura, que é formada por dois números de 256 bits, um denominado de r e outro de s. Logo, as variáveis que formam o registro são duas do tipo *bigint*, uma representando r e outra representando o s. A montagem fica assim:

```
1 typedef struct _signature{
2         bigint r;
3         bigint s;
4 }signature;
```

4.7.2 key_generator

O protocolo referente a este método, assim como os dois posteriores, foram explicados na seção 3.4, por isso, o que será descrito nestas seções será puramente a implementação.

O método **key_generator** recebe como parâmetro um *bigint*, denominado *prk*, que armazenará a chave privada, uma variável *pub_key*, denominada de *pbk* onde será armazenada o chave pública, um objeto da classe *EC_Point*, denominado de G, que é o ponto gerador do grupo cíclico, e um outro do tipo *bigint*, denominado *order*, que é a ordem do grupo cíclico.

Inicialmente o método gera um número aleatório com a função **random_bigint** e atribui o resultado módulo a ordem *order* no *prk*. Em seguida, faz-se a conta de $B = prk * G$ para gerar o ponto na curva elíptica B, atribui os valores de $G, B, order$ na variável registro *pbk* e termina o método.

O código fica assim:

```
1 void ECDSA::key_generator(bigint prk,pub_key *pbk,EC_Point G,bigint
      order){
2         bigint temp;
3         EC_Point B;
4     //gera um numero aleatorio de 256 bits
5         random_bigint(prk);
6     //calcula o modulo do numero mod order
7         div_mod_bigint(temp,prk,prk,order);
8     //calcula B=order*G
9         B.atribuir(EC_op::mult(G,prk));
10    //atribui os valores necessarios no pbk
11        atrib_bigint(pbk->order,order);
12        pbk->A.atribuir(G);
13        pbk->B.atribuir(B);
14 }
```

4.7.3 sign

O método **sign** recebe como parâmetros uma variável *signature*, denominada *sign*, onde será armazenada a assinatura gerada, dois do tipo *bigint*, denominados de *h_x* e *prk*, onde estão, respectivamente, o resumo criptográfico da mensagem e a chave privada, e por fim um objeto *EC_Point*, denominado G, que é o gerador do grupo cíclico, e um *bigint*, denominado *order*, que é a ordem do grupo cíclico.

Inicialmente, gera-se um número aleatório e o armazena na variável ke, faz-se a redução de $ke \pmod{order}$, em seguida calcula-se o ponto $K = ke * G$, dentro da curva elíptica. Associa-se a coordenada x do ponto K à variável r do registro *sign*, depois calcula-se s do registro *sign* com a fórmula $s \equiv (h_x - d * r) * ke^{-1} \pmod{order}$, onde ke^{-1} é a inversa de ke módulo *order*.

O código, então, fica assim:

```
void ECDSA::sign(signature *sign,bigint h_x,bigint prk,EC_Point G,
   bigint order){
        bigint ke,temp,t2;
        EC_Point K;
    //gera o numero aleatorio ke
        random_bigint(ke);
    //reduz ke modulo order
        div_mod_bigint(temp,ke,ke,order);
    //calcula o r da assinatura, sendo x do ponto order*G
        K.atribuir(EC_op::mult(G,ke));
        K.get_x(sign->r);
    //calcula o s da assinatura
        Algebra_op::inverse_eea(temp,ke,order);
        mult_bigint(sign->s,prk,sign->r);
        div_mod_bigint(t2,sign->s,sign->s,order);
        add_bigint(sign->s,sign->s,h_x);
        div_mod_bigint(t2,sign->s,sign->s,order);
        mult_bigint(t2,sign->s,temp);
        div_mod_bigint(t2,sign->s,t2,order);
}
```

4.7.4 verify

O método **verify** recebe como parâmetros, um *bigint*, denominado h_x, que terá o resumo criptográfico da mensagem, um do tipo *pub_key*, denominado *pbk*, que terá a chave pública do assinante, e a assinatura *sign* do tipo *signature*. E retorna verdadeiro se a assinatura for compatível com a chave pública e falso caso contrário.

Primeiro, calcula-se a inversa de s módulo *order*, sabendo que este faz parte da chave pública *pbk*, e armazena na variável w. Em seguida calcula-se as variáveis $u1$ e $u2$ com as fórmulas $u1 \equiv w * h_x \pmod{order}$ e $u2 \equiv w * r \pmod{order}$. É computado então o ponto P na curva elíptica, em que $P = u1 * A + u2 * B$, onde A é o gerador do grupo cíclico e B o ponto calculado na geração de chaves com a fórmula $B = d * A$, em que d é a chave privada, ambos A e B são parte de *pbk*.

Caso a coordenada x do ponto P for equivalente a r módulo *order*, o método retorna verdadeiro, caso contrário, retorna falso.

O código fica assim:

```
bool ECDSA::verify(bigint h_x,pub_key pbk,signature sign){
        bigint u1,u2,w,t1,temp;
        EC_Point uA,uB,P;
    //calculo de w
        Algebra_op::inverse_eea(w,sign.s,pbk.order);
    //calculo de u1
        mult_bigint(u1,w,h_x);
        div_mod_bigint(temp,u1,u1,pbk.order);
    //calculo de u2
        mult_bigint(u2,w,sign.r);
```

```
11          div_mod_bigint(temp,u2,u2,pbk.order);
12      //calculo de ponto P=u1*A+u2*B
13          uA.atribuir(EC_op::mult(pbk.A,u1));
14          uB.atribuir(EC_op::mult(pbk.B,u2));
15          P.atribuir(EC_op::add(uA,uB));
16          P.get_x(t1);
17      //testa se x de P e igual a r
18          if(eq_bigint(t1,sign.r)){
19                  return true;//caso seja, assinatura valida
20          }
21          return false;//caso nao, assinatura invalida
22  }
```

E assim foi concluída a descrição dos módulos produzidos para este projeto. Na próxima seção será descrito uma interface que foi feita para servir de exemplo do uso destes módulos.

4.8 Interface de exemplo - 256ECDSA.cpp

Esta interface foi construída com o objetivo de demonstrar a utilização dos módulos descritos acima, assim como provar seu bom funcionamento.

A arquitetura desta interface é a seguinte:

É incluído o módulo *ECDSA.h*, que possui todos os protocolos da assinatura ECDSA.

É definida a função *main*, que irá iniciar o programa e as seguintes outras funções:

- **ler_parametros** - lê os parâmetros da curva de um arquivo
- **trocar_parametros** - troca os parâmetros da curva
- **gerar_chaves** - gera um par de chaves
- **inserir_chaves** - insere e testa um par de chaves
- **sha256_arquivo** - calcula o sha-256 de um arquivo
- **sha256_texto** - calcula o sha-256 de uma string
- **assinar** - assina um arquivo
- **assinar_texto** - assina uma string
- **verificar** - verifica um arquivo assinado
- **verificar_texto** - verifica um assinatura proveniente de uma string

4.8.1 Escolha de parâmetros e Menu

A interface inicia com a escolha dos parâmetros que definiram a curva elíptica. Para esta interface, foram escolhidos dois grupos de parâmetros muito utilizados por sistemas no mundo, ambos foram definidos pela Certicom [12], são denominadas de *Secp256k1* e

[12] Uma subsidiária integral da Blackberry

Secp256r1, da qual o primeiro é bem conhecido por ser utilizado na rede *BitCoin* [11]. Estes parâmetros possuem o primo p, os coeficientes a e b que descrevem a curva elíptica, o ponto gerador do grupo cíclico na curva G, a ordem do grupo cíclico n e um co-fator h que é definido como número de elementos total na curva elíptica definida por a, b, p, dividido pela ordem do grupo cíclico n. O ideal é que $h < 4$.

No menu há a opção de trocar de parâmetro. É importante saber que os parâmetros da curva, relevantes para o protocolo, são enviados como parte da chave pública. O código que atribui os parâmetros escolhidos, que estão presentes em arquivo, nas variáveis do programa é este:

```
1  //funcao para escolher o parametro a ser usado
2  void trocar_parametro(EC_Point *G,bigint ord){
3          bool remain=true;
4          int opcao;
5          FILE *fp;
6      //gera o menu de parametros
7          while(remain){
8                  printf("Escolha qual parametro usar:\n");
9                  printf("1-Secp256k1\n2-Secp256r1\n\nOpcao: ");
10                 scanf("%d",&opcao);
11                 switch(opcao){
12                         case 1://escolha secp256k1
13                                 fp = fopen("./parameters/Secp256k1.
                                      txt","r");
14                                 ler_parametros(G,ord,fp);
15                                 fclose(fp);
16                                 remain = false;
17                                 break;
18                         case 2://escolha secp256r1
19                                 fp = fopen("./parameters/Secp256r1.
                                      txt","r");
20                                 ler_parametros(G,ord,fp);
21                                 fclose(fp);
22                                 remain = false;
23                                 break;
24                         default:
25                                 break;
26                 }
27                 system(CLEAR_CMD);
28         }
29 }
30 //funcao para ler o parametro de um arquivo
31 void ler_parametros(EC_Point *G,bigint ord,FILE *fp){
32         bigint x,y,a,b,p;
33     //leitura do p
34         while('='!=getc(fp));
35         fler_hex_bigint(p,fp);
36         G->set_p(p);
37     //leitura do a
```

```
38          while('='!=getc(fp));
39          fler_hex_bigint(a,fp);
40          G->set_a(a);
41      //leitura do b
42          while('='!=getc(fp));
43          fler_hex_bigint(b,fp);
44          G->set_b(b);
45      //leitura do x de G
46          while('='!=getc(fp));
47          fler_hex_bigint(x,fp);
48          G->set_x(x);
49      //leitura do y de G
50          while('='!=getc(fp));
51          fler_hex_bigint(y,fp);
52          G->set_y(y);
53      //leitura da ordem
54          while('='!=getc(fp));
55          fler_hex_bigint(ord,fp);
56  }
```

A interface se apresenta como na figura 4.2.

Figura 4.2: Interface: Menu de Parâmetros

O exemplo de arquivo onde estão os parâmetros está na figura ??

Depois de escolhido o parâmetro, entra-se no menu principal, no qual há as opções de ação que englobam, geração de chaves, inserção de chaves, cálculo de resumo criptográfico a partir de SHA-256 de um texto do teclado, ou de um arquivo, assinar um arquivo ou um texto do teclado, verificação de uma assinatura de um texto do teclado ou de um arquivo e troca de parâmetros da curva.

O código que gera o menu não é importante para o objetivo deste documento, as funções que fazem cada uma das funções será explicada posteriormente.

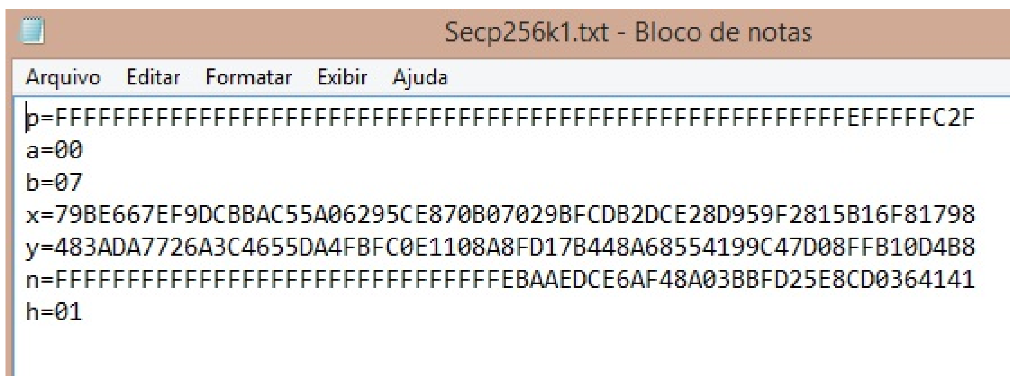

Figura 4.3: Arquivo de Parâmetros

A interface do menu é esta demonstrada na figura 4.4

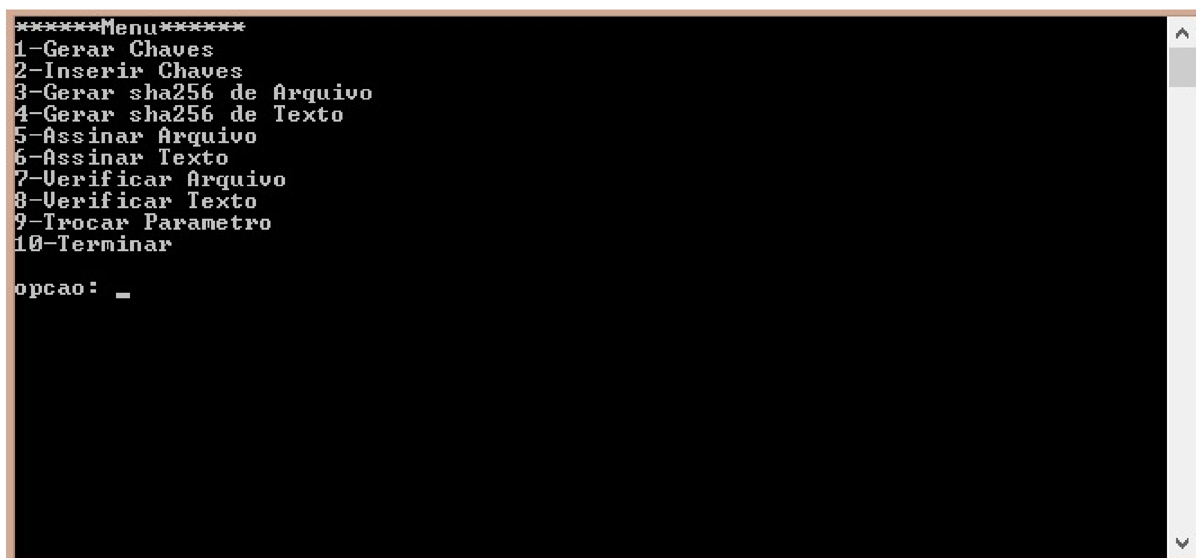

Figura 4.4: Interface: Menu Principal

4.8.2 Gerar ou Inserir Chaves

Na opção de gerar chaves o usuário escreve o nome de quem será o dono da chave, então é chamado o método **key_generator** para gerar as chaves que posteriormente serão guardadas em arquivo, a chave privada no arquivo <nome>_prk, e a pública na <nome>_pbk, como nas figuras 4.5 e 4.6

O código desta função é esta abaixo:

```
1 void gerar_chaves(EC_Point G,bigint ord){
2         FILE *fp;
3         bigint prk,x,y,temp;
4         pub_key pbk;
5         char nome[30],path_pbk[50],path_prk[50];
```

```
Insira o Nome do Dono da Chave
exemplo
```

```
Gerando Chaves ...
```

```
Gerando Chaves ...
Chaves Geradas
```

Figura 4.5: Interface: Gerando Chave

```
exemplo_pbk.txt - Bloco de notas
Arquivo Editar Formatar Exibir Ajuda
bx=0d00f7b242d1b6bd17b30674376d1b55199ec0c9b5436d4451f40f30805def3a
by=99ef11ace864ebddc1a4e2b268b1c03ae2222179dbbe488ea4a3f34ced79cd88
p=fffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f
a=00
b=07
q=fffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141
ax=79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
ay=483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
```

```
exemplo_prk.txt - Bloco de notas
Arquivo Editar Formatar Exibir Ajuda
643fca0434bfbc16866d057fbfabd3c90c82517da3df12f44cfcb36a201cbc54
```

Figura 4.6: Chaves Geradas

```
6       //prepara local onde sera salvo as chaves
7           strcpy(path_pbk,"./Keys/");
8           strcpy(path_prk,"./Keys/");
9       //leitura do nome
10          printf("Insira o Nome do Dono da Chave\n");
11          scanf("%s",nome);
12          getchar();
13          strcat(path_pbk,nome);
14          strcat(path_prk,nome);
15          strcat(path_pbk,"_pbk.txt");
16          strcat(path_prk,"_prk.txt");
17          system(CLEAR_CMD);
18          printf("Gerando Chaves ...\n");
19      //chamada do metodo que gera a chave
20          ECDSA::key_generator(prk,&pbk,G,ord);
21          pbk.B.get_x(x);
22          pbk.B.get_y(y);
23      //escrevendo o arquivo da chave privada
24          fp = fopen(path_prk,"w");
25          fimprime_hex_bigint(prk,fp);
26          fclose(fp);
27      //escrevendo o arquivo da chave publica
28          fp = fopen(path_pbk,"w");
29          fprintf(fp,"bx=");fimprime_hex_bigint(x,fp);fprintf(fp,"\n"
               );
30          fprintf(fp,"by=");fimprime_hex_bigint(y,fp);fprintf(fp,"\n"
               );
31          G.get_p(temp);
32          fprintf(fp,"p=");fimprime_hex_bigint(temp,fp);fprintf(fp,"\
               n");
33          G.get_a(temp);
34          fprintf(fp,"a=");fimprime_hex_bigint(temp,fp);fprintf(fp,"\
               n");
35          G.get_b(temp);
36          fprintf(fp,"b=");fimprime_hex_bigint(temp,fp);fprintf(fp,"\
               n");
37          fprintf(fp,"q=");fimprime_hex_bigint(ord,fp);fprintf(fp,"\n
               ");
38          G.get_x(temp);
39          fprintf(fp,"ax=");fimprime_hex_bigint(temp,fp);fprintf(fp,"
               \n");
40          G.get_y(temp);
41          fprintf(fp,"ay=");fimprime_hex_bigint(temp,fp);fprintf(fp,"
               \n");
42          fclose(fp);
43          printf("Chaves Geradas\n");
44          getchar();
45  }
```

A função de inserir chaves, não gera um par de chaves, e sim, lê as chaves pública e

privada e testa se elas formam um par, se formarem elas são escritas em arquivo, se não é dada uma mensagem e nada acontece. O parte do código que faz o teste está abaixo:

```
    //le as chaves
    printf("Insira a chave privada em hexadecimal\n");
        ler_hex_bigint(prk);
        printf("Insira a coordenada X do ponto B da chave publica em hexadecimal\n");
        ler_hex_bigint(x);
        printf("Insira a coordenada Y do ponto B da chave publica em hexadecimal\n");
        ler_hex_bigint(y);
        system(CLEAR_CMD);
        printf("Testando Chaves ...\n");
    //testa compatibilidade
        B.atribuir(EC_op::mult(G,prk));
        B.get_x(temp);
        B.get_y(t2);
        if(!(eq_bigint(x,temp)&&eq_bigint(y,t2))){
                //caso nao seja compativel
        printf("Chaves nao correspondem.\n");
                getchar();
                return;
        }
    //caso seja, continua como no codigo de geracao
```

Um exemplo da interface funcionando se encontra nas figuras 4.7 e 4.8

```
Insira o Nome do Dono da Chave
exemplo
Insira a chave privada em hexadecimal
2
Insira a coordenada X do ponto B da chave publica em hexadecimal
3
Insira a coordenada Y do ponto B da chave publica em hexadecimal
4_

Testando Chaves ...
Chaves nao correspondem.
```

Figura 4.7: Chaves Não Inseridas

4.8.3 SHA-256 de arquivo ou texto

Estas duas funções fazem basicamente a mesma coisa, a diferença é que, para arquivo, a função abre o arquivo e envia o ponteiro para a função **arq_sha256** já abordada anteriormente, para o texto enviamos a cadeia de caracteres que a armazena para a função **tex_sha256**. Eis o código:

```
//hash de arquivo
void sha256_arquivo(){
```

```
Insira o Nome do Dono da Chave
exemplo
Insira a chave privada em hexadecimal
643fca0434bfbc16866d057fbfabd3c90c82517da3df12f44cfcb36a201cbc54
Insira a coordenada X do ponto B da chave publica em hexadecimal
0d00f7b242d1b6bd17b30674376d1b55199ec0c9b5436d4451f40f30805def3a
Insira a coordenada Y do ponto B da chave publica em hexadecimal
99ef11ace864ebddc1a4e2b268b1c03ae2222179dbbe488ea4a3f34ced79cd88

Testando Chaves ...
Chaves Inseridas com Sucesso
```

Figura 4.8: Chaves Inseridas

```c
        FILE *fp;
        char arq[100];
        bigint h;
    //le o nome do arquivo
        printf("Insira o nome do arquivo:\n");
        scanf("%s",arq);
        getchar();
        fp = fopen(arq,"rb");
    //testa se o arquivo existe
        if(fp==NULL){
                printf("arquivo nao encontrado.\n");
                return;
        }
    //chama a funcao de resumo
        arq_sha256(fp,h);
        fclose(fp);
    //imprime resultado na tela
        printf("sha256(arquivo) = ");imprime_hex_bigint(h);printf("\n");
        getchar();
}
//hash de texto
void sha256_texto(){
        char texto[1024];
        bigint h;
    //le o texto do teclado
        printf("Insira o texto:\n");
        scanf("%[^\n]s",texto);
        getchar();
        //chama a funcao de resumo
        tex_sha256(texto,h);
    //imprime o hash na tela
        printf("sha256(texto) = ");imprime_hex_bigint(h);printf("\n");
        getchar();
}
```

A figura 4.9 mostra um exemplo de ambas em ação.

```
Insira o nome do arquivo:
novo.txt
sha256(arquivo) = ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015
ad

Insira o texto:
exemplo
sha256(texto) = c0fc3c713f09a43384ac08f7d91fca430dcbc6466fff9284ce4571bdc2c8f9f9
```

Figura 4.9: Interface do SHA-256

4.8.4 Assinar arquivo ou texto

As funções que geram assinatura de um arquivo ou de um texto se utilizam do método **sign** da classe *ECDSA*. A diferença é, que quando é um arquivo, testa-se se ele existe e caso sim chama a função **arq_sha256** para calcular seu resumo, caso contrário, usa-se a função **tex_sha256** para calcular o resumo do que foi lido do teclado, mas em ambos lê-se o nome de quem irá assinar, e verifica-se se há chave referente a tal assinante. Caso não exista, a função termina e nada acontece, caso exista o arquivo ou texto é assinado, criando um arquivo com tal assinatura.

Para textos o arquivo terá o nome de ”temp_<assinante>_sign.txt”, caso seja arquivo o nome será ”<arquivo>_<assinante>_sign.txt”. A parte do código de importância para este documento é esta:

```
1  //assinatura de arquivo
2  void assinar(EC_Point G,bigint ord){
3          //...(codigo)...
4      //ler chave privada do assinante
5          fler_hex_bigint(prk,fp);
6          fclose(fp);
7      //...(codigo)...
8      //calculo de hash do arquivo
9          arq_sha256(fp,h_x);
10         fclose(fp);
11     //assinando
12         ECDSA::sign(&sign,h_x,prk,G,ord);
13     //criacao do arquivo de assinatura
14         strcpy(path,"./signatures/");
15         strcat(path,m);
16         strcat(path,"_");
17         strcat(path,nome);
18         strcat(path,"_sign.txt");
19         fp = fopen(path,"w");
20         fimprime_hex_bigint(sign.r,fp);fprintf(fp,"\n");
21         fimprime_hex_bigint(sign.s,fp);fprintf(fp,"\n");
```

```
22          fclose(fp);
23          printf("\nAssinado\n");
24          getchar();
25  }
26  //assinatura de texto
27  void assinar_texto(EC_Point G,bigint ord){
28          //ler chave privada
29          fler_hex_bigint(prk,fp);
30          fclose(fp);
31      //leitura do texto
32          printf("Insira o texto a ser assinado.\n");
33          scanf("%[^\n]s",m);
34          getchar();
35          system(CLEAR_CMD);
36          printf("Assinando ...");
37      //hash do texto
38          tex_sha256(m,h_x);
39      //assinando texto
40          ECDSA::sign(&sign,h_x,prk,G,ord);
41      //escrevendo texto com assinatura
42          strcpy(path,"./signatures/temp_");
43          strcat(path,nome);
44          strcat(path,"_sign.txt");
45          fp = fopen(path,"w");
46          fimprime_hex_bigint(sign.r,fp);fprintf(fp,"\n");
47          fimprime_hex_bigint(sign.s,fp);fprintf(fp,"\n");
48          fprintf(fp,"%s",m);
49          fclose(fp);
50          printf("\nAssinado\n");
51          getchar();
52  }
```

Os exemplos da interface para estas funções estão nas figuras 4.10 e 4.11.

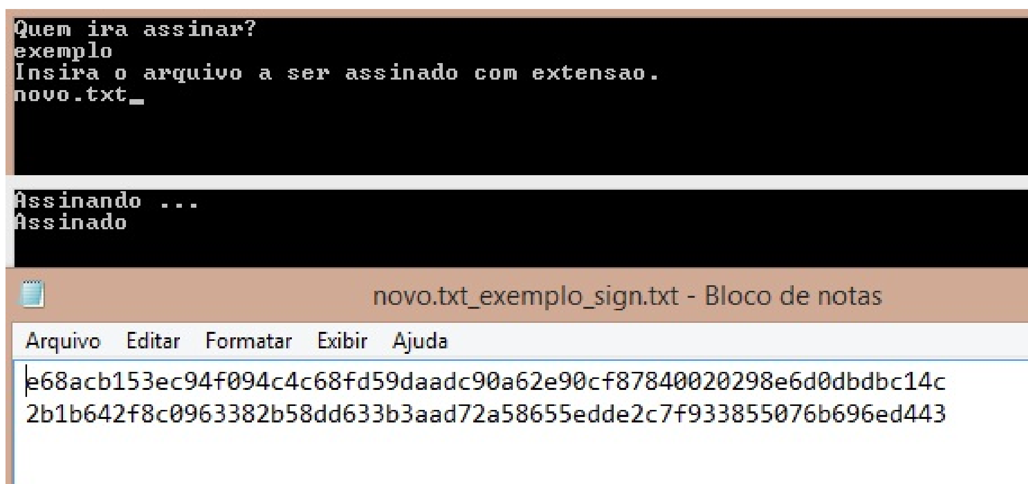

Figura 4.10: Assinatura de Arquivo

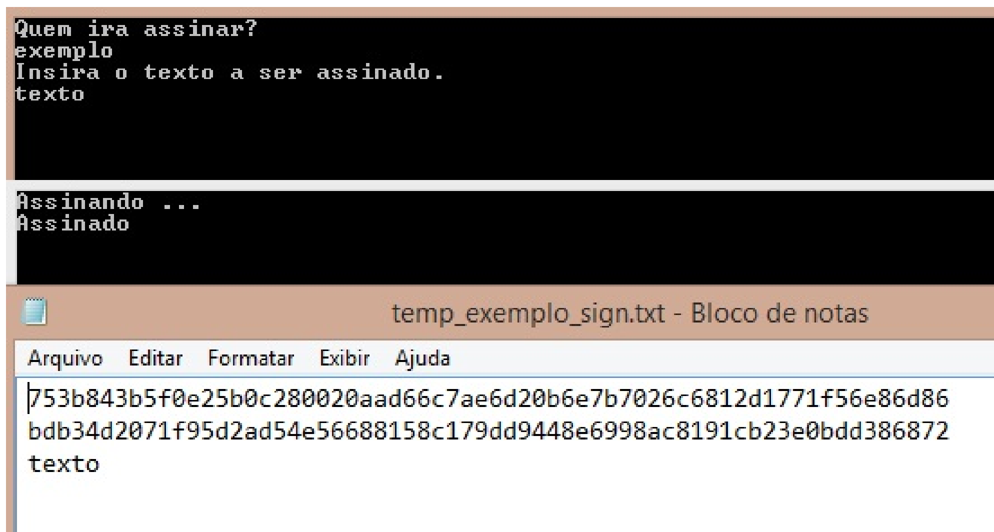

Figura 4.11: Assinatura de Texto

Perceba que no caso da assinatura de texto, o mesmo é escrito no arquivo da assinatura logo depois dela.

4.8.5 Verificar Assinatura de arquivo ou texto

A função de verificação de arquivo primeiro lê o nome do arquivo a ser verificado e em seguida lê o dono da assinatura, procura os arquivos correspondentes a assinatura e à chave pública. Caso ambas existam é feito o cálculo do resumo criptográfico do arquivo e depois testa-se se a assinatura em questão está de acordo com a chave pública, se tiver imprime que a assinatura é autêntica, caso contrário imprime que a assinatura não é autêntica.

Eis a parte do código que importa para o objetivo do complemento:

```
void verificar(EC_Point G,bigint ord){
        //...(codigo)...
    //calculo do hash do arquivo
        arq_sha256(fp,h_x);
        fclose(fp);
    //...(codigo)...
    //leitura da chave publica
        fp = fopen(path,"r");
        if(fp==NULL){
                printf("Chave nao encontrada!\n");
                getchar();
                return;
        }
        while('='!=getc(fp));
        fler_hex_bigint(x,fp);
        while('='!=getc(fp));
        fler_hex_bigint(y,fp);
        fclose(fp);
```

```
19      //atribuindo a chave no registro pub_key
20          atrib_bigint(pbk.order,ord);
21          pbk.A.atribuir(G);
22          pbk.B.atribuir(G);
23          pbk.B.set_x(x);
24          pbk.B.set_y(y);
25      //abrindo lendo assinatura
26          strcpy(path,"./signatures/");
27          strcat(path,m);
28          strcat(path,"_");
29          strcat(path,nome);
30          strcat(path,"_sign.txt");
31          fp = fopen(path,"r");
32          if(fp==NULL){
33                  printf("Assinatura nao encontrada!\n");
34                  getchar();
35                  return;
36          }
37          fler_hex_bigint(sign.r,fp);
38          fler_hex_bigint(sign.s,fp);
39          fclose(fp);
40          system(CLEAR_CMD);
41          printf("Autenticando ...");
42      //autenticando assinatura
43          if(ECDSA::verify(h_x,pbk,sign)){
44                  printf("\nAssinatura Autentica\n");//assinatura
                        confere
45          }else{
46                  printf("\nAssinatura nao autentica\n");//assinatura
                         nao confere
47          }
48          getchar();
49  }
```

Na verificação de assinatura de texto, devemos lembrar que o arquivo onde está a assinatura também possui o texto. Para este só é lido o dono da assinatura, caso tenha o arquivo com sua chave pública e o arquivo ”temp_<nome>_sign.txt”, é testada a assinatura. Caso contrário, não é feito nada. Depois é calculado o resumo criptográfico do texto que está junto com a assinatura. A partir deste ponto fica idêntico à função anterior.

A parte relevante do código está abaixo:

```
1   //verificando assinatura de texto
2   void verificar_texto(EC_Point G,bigint ord){
3       //...(codigo)...
4       //lendo a chave publica
5           fp = fopen(path,"r");
6           if(fp==NULL){
7                   printf("Chave nao encontrada!\n");
8                   getchar();
```

```
9                   return;
10          }
11          while('='!=getc(fp));
12          fler_hex_bigint(x,fp);
13          while('='!=getc(fp));
14          fler_hex_bigint(y,fp);
15          fclose(fp);
16      //atribuindo valores a variavel pbk
17          atrib_bigint(pbk.order,ord);
18          pbk.A.atribuir(G);
19          pbk.B.atribuir(G);
20          pbk.B.set_x(x);
21          pbk.B.set_y(y);
22      //abrindo e lendo arquivo de assinatura
23          strcpy(path,"./signatures/temp_");
24          strcat(path,nome);
25          strcat(path,"_sign.txt");
26          fp = fopen(path,"r");
27          if(fp==NULL){
28                  printf("Assinatura nao encontrada!\n");
29                  getchar();
30                  return;
31          }
32          fler_hex_bigint(sign.r,fp);
33          fler_hex_bigint(sign.s,fp);
34          i=0;
35      //lendo texto que foi assinado
36          while(!feof(fp)){
37                  fscanf(fp,"%c",&m[i]);
38                  i++;
39          }
40          m[i]='\0';
41      //calcula o hash do texto
42          tex_sha256(m,h_x);
43          fclose(fp);
44          system(CLEAR_CMD);
45          printf("Autenticando ...");
46      //verificando assinatura
47          if(ECDSA::verify(h_x,pbk,sign)){//caso seja autentico
48                  printf("\nAssinatura Autentica\n");
49          }else{//caso nao seja autentico
50                  printf("\nAssinatura nao autentica\n");
51          }
52          getchar();
53  }
```

O exemplo de interface da verificação de arquivo está na figura 4.12 e o da verificação de texto na figura 4.13, para este foi feita manualmente uma mudança no arquivo de assinatura para poder ver o caso de assinatura inválida.

```
insira arquivo a ser verificado com extensao.
novo.txt
insira o nome do Dono da Assinatura.
exemplo

Autenticando ...
Assinatura Autentica
_
```

Figura 4.12: Verificação de Assinatura

```
insira o nome do Dono da Assinatura.
exemplo_

Autenticando ...
Assinatura nao autentica
_
```

Figura 4.13: Verificação de Texto

As últimas opções são a troca de parâmetro, que já foi abordada, e sair do programa. Com isso encerra-se a documentação das implementação dos módulos.

Capítulo 5

Conclusão

O objetivo deste projeto foi fazer um estudo do que engloba a assinatura digital, de modo específico, os esquemas que trabalham com o problema do logaritmo discreto, e principalmente o esquema baseado em curvas elípticas denominado de ECDSA, quais as suas características que o tornam superior em robustez e velocidade comparado a outros métodos utilizados para assinatura digital.

Com este estudo, desejava-se alcançar um conhecimento necessário para poder explicar o porque da crescente adesão da criptografia de curvas elípticas, em especial a assinatura digital, por sistemas de segurança de todo mundo, além de poder implementar um módulo que simula-se o protocolo ECDSA.

Depois de todo estudo e a concretização deste com os módulos implementados, os quais documentamos aqui, percebemos que a matemática que envolve as curvas elípticas é muito útil para a criptografia, principalmente para a cifragem e a autenticação. Desde sua primeira utilização em assinaturas digitais em 1985 até sua concretização no cenário em 1998, as assinaturas baseadas em curvas elípticas vem se mostrando robusta em comparação a esquemas já conhecidos.

Mesmo hoje, a matemática de curvas elípticas foi muito pouco vista, em comparação com as outras usadas na criptografia, e com isso, suas vulnerabilidades ainda estão sendo descobertas. Como a robustez de um protocolo de segurança é provada com o tempo e com os ataques que são propostos contra eles, ainda não há muitas provas de quão robusto é o protocolo baseado em curvas elípticas, ainda mais em 1985, quando o primeiro protocolo foi proposto. Dado isto, justifica-se a demora para sua aceitação no mundo da criptografia. Porém depois de quase 15 anos sem que um ataque tenha sido efetivo, este esquema começou a ser aceito, e cada vez mais sistemas tem migrado para o protocolo baseado em curvas elípticas. E hoje, depois de 30 anos de sua primeira utilização, os protocolos baseados em curvas elípticas, sendo bem construídos, ainda se mostram robustos.

Ao término deste projeto, foi concluído que ainda é difícil se aprofundar em tal assunto, dada a pouca literatura que existe sobre o tema, e principalmente dentro do Brasil, onde a própria criptografia possui poucos estudos. Apesar disso, o estudo trouxe um bom conhecimento, que foi comprovado com a criação dos módulos descritos neste documento.

Dos objetivos propostos, percebe-se que o conhecimento pretendido foi alcançado, por meio de livros e artigos, em sua maior parte, estrangeiros, e com este conhecimento, conclui-se que o fato de cada vez mais as curvas elípticas estarem sendo usadas por sistemas de segurança se deve à sua matemática relativamente nova, e a possibilidade

de descrever com ela o problema do logaritmo discreto com números bem menores que os outros protocolos baseados em números inteiros. Com números menores, os cálculos são mais rápidos e o espaço necessário para chaves e assinaturas são menores. E como dito, até hoje, os ataques testados contra os esquemas baseados em curvas elípticas não se mostraram efetivos.

Quanto ao módulo, digo que este alcançou seu objetivo, que era uma implementação simples, que apenas se utiliza de bibliotecas padrões, para que, quem a use tenha um contato com um exemplo de protocolo que se utiliza de curvas elípticas, o ECDSA.

Como dito, a maior dificuldade deste projeto foi obter material de estudo, e principalmente, adquirir este conhecimento em pouco tempo, visto que no início deste projeto, o conhecimento deste que escreve era mínimo na área. Outra dificuldade foi a necessidade de implementar, além do protocolo, ferramentas que o padrão GCC não possui, em especial a limitação deste padrão para tamanho máximo de inteiros. Nele o tamanho máximo de um número é de 64 bits, e a implementação demandava números de 256 bits, para isto, foi necessário a criação do tipo *bigint*, e a complexidade da implementação de suas operações básicas foram um grande problema.

Como trabalhos futuros, proponho uma otimização nos módulos criados, já que estes não eram o foco deste projeto, principalmente nos cálculos básicos como o cálculo do módulo. Também proponho o aproveitamento do módulo para implementação de outros protocolos que são definidos dentro das curvas elípticas. Outra proposta é, dado a profundidade de conceitos matemáticos obtidos neste projeto, estudar possíveis vulnerabilidades da matemática de curvas elípticas.

Referências

[1] Adi Akavia, Shafi Goldwasser, and Carmit Hazay. Distributed public key schemes secure against continual leakage. In *PODC*, pages 155–164, 2012.

[2] Mihir Bellare and Viet Tung Hoang. Adaptive witness encryption and asymmetric password-based cryptography. Cryptology ePrint Archive, Report 2013/704, 2013. http://eprint.iacr.org/.

[3] Joppe W. Bos, J. Alex Halderman, Nadia Heninger, Jonathan Moore, Michael Naehrig, and Eric Wustrow. Elliptic curve cryptography in practice. 2013. http://eprint.iacr.org/.

[4] Certicon. Sec2:recommended elliptic curve domain parameters. http://perso.univ-rennes1.fr/sylvain.duquesne/master/standards/sec2_final.pdf.

[5] Niels Ferguson and Bruce Schneier. *Practical Cryptography*. John Wiley & Sons, Inc., New York, NY, USA, 1 edition, 2003.

[6] Praveen Gauravaram and Lars R. Knudsen. Cryptographic hash functions. In Stavroulakis and Stamp [17], pages 59–79.

[7] Ghassan Karame, Elli Androulaki, and Srdjan Capkun. Two bitcoins at the price of one? double-spending attacks on fast payments in bitcoin. *IACR Cryptology ePrint Archive*, 2012.

[8] Ann Hibner Koblitz, Neal Koblitz, and Alfred Menezes. Elliptic curve cryptography: The serpentine course of a paradigm shift. 2008. http://eprint.iacr.org/. 1

[9] Neal Koblitz. *A course in number theory and cryptography*. Springer-Verlag, New York, 2nd edition, 1994. 24

[10] Victor S. Miller. Use of elliptic curves in cryptography. In *Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings*, pages 417–426, 1985. 1

[11] Satoshi Nakamoto. Bitcoin: A peer-to-peer eletronic cash system, Maio 1999. 2, 81

[12] NIST. Descriptions of sha-256, sha-384, and sha-512. http://csrc.nist.gov/groups/STM/cavp/documents/shs/sha256-384-512.pdf.

[13] Christof Paar, Jan Pelzl, and Bart Preneel. *Understanding cryptography : a textbook for students and practitioners*. Springer, Berlin, Heidelberg, 2010. viii, 6, 24, 25, 26, 27, 28, 31, 33, 36, 41, 53, 55, 59, 67

[14] David A. Patterson and John L. Hennessy. *Computer Organization and Design - The Hardware / Software Interface (Revised 4th Edition)*. The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press, 2012. 41, 53

[15] Wouter Penard and Tim van Werkhoven. On the secure hash algorithm family. http://www.staff.science.uu.nl/~werkh108/docs/study/Y5_07_08/infocry/project/Cryp08.pdf.

[16] Pedro Resende. Notas de aula de segurança de dados. http://www.cic.unb.br/docentes/pedro/segdados_files/. 11

[17] Peter P. Stavroulakis and Mark Stamp, editors. *Handbook of Information and Communication Security*. Springer, 2010. 96

[18] Douglas Robert Stinson. *Cryptography : theory and practice*. Discrete mathematics and its applications. Chapman & Hall/CRC, Boca Raton, 2006. 9, 10, 17, 21