Curso de Criptografia

UFAL, Maceió — Inverno 2019

Enno Nagel

Inverno 2019

Curso dado no Instituto Matemático da UFAL no semestre de Inverno de 2019. Introduz

- à histórica,
- às aplicações,
- às implementações (em `Python`), e
- à teoria (dos números) por trás

da criptográfia.

Sumário

Introdução

A *criptografia* é a arte da transformação de um texto (ou qualquer outro arquivo) compreensível em um texto incompreensível tal que só uma informação adicional secreta, a *chave*, permita desfazê-la; útil desde a antiguidade, por exemplo, para ocultar do inimigo a comunicação militar.

Resumo

O livro primeiro estabelece a diferença entre

- a antiga criptografia *simétrica* usada entre correspondentes *conhecidos* para, por exemplo,
  - cifrar a comunicação em uma rede sem fio, e
  - proteger um arquivo no disco rígido, ...

e

- a recente criptografia *assimétrica* usada entre correspondentes *desconhecidos* para, por exemplo,
  - cifrar a comunicação na internet, e
  - autenticar documentos ou transações (financeiras).

Um sobrevoo da História da criptografia apresenta os algoritmos simétricos da antiguidade que servirão como pedras angulares dos algoritmos criptográficos simétricos modernos, as *redes de permutação e substituição*, como o atual padrão AES.

Este é percorrido passo a passo para compreender a sua invulnerabilidade contra os ataques mais sofisticados para decifrar o texto cifrado por ele sem conhecimento da chave.

Entra nos anos 70 a criptografia assimétrica, indispensável na idade da internet e ubíqua no nosso dia-a-dia, por exemplo, para cifrar os mensageiros do celular, sites seguros no navegador e transações financeiras à distância. Sofre, porém, do ataque pelo homem no meio que assume as identidades dos correspondentes, o que leva à infraestrutura de teias de confiança, autoridades e certificados, visualizados por cadeados na barra de endereço no navegador.

Por trás de toda a criptografia assimétrica figura a Aritmética Modular que permite criar uma aplicação invertível (chamada de *arapuca*) cujo inverso é praticamente incomputável. Exemplos são a potenciação respectivamente a exponenciação modular que são usadas pelos dois algoritmos fundamentais da criptografia assimétrica:

- O mais antigo (dos anos 70, contudo até hoje seguro) é a *Troca de Chaves por Diffie-Hellman*, e
- o mais usado até hoje, o `RSA`

que são ambos estudados passo a passo e cujas bases teóricas são motivadas e explicadas.

Ultimamente a criptografia por *curvas elípticas finitas*, um grupo finito de pontos num reticulado no plano, tornou-se o novo padrão da criptografia assimétrica. A sua aplicação arapuca é análoga à do Diffie-Hellman, assim como os passos da cifração e assinatura, mas pela sua maior complexidade mais segura.

Elas são usadas, por exemplo, para assinar as transações na criptomoeda `Bitcoin` cujo conceito e funcionamento é desenvolvido na parte final do livro: Ao contrário de uma moeda comum na qual os negociantes confiam em um banco, uma criptomoeda mantém um livro-razão público de todas as transações. Para garantir a sua imutabilidade, ele precisa de *funções de embaralhamento* (ou *hashes*) que têm vários usos na computação, os quais são todos apresentados e comparados no penúltimo capítulo. O último capítulo examina a anatomia do livro-razão, a *Blockchain* ou, literalmente, cadeia de blocos: os seus blocos e os laços entre eles (os hashes). Em seguida responde às questões como ela

- começou,
- cresce por *mineração*, e
- é mantida consistente pela rede dos participantes.

História

Historicamente, a chave para codificar e descodificar é a mesma: a *cifração simétrica*. Nos anos 70, surgiu a *cifração assimétrica*, na qual as chaves para cifrar (*a chave pública*) e decifrar (*a chave secreta*) são diferentes. Matematicamente, ela baseia-se em uma *função alçapão*, uma função invertível que é facilmente computável, mas cujo inverso é computacionalmente inviável.

Hoje em dia os algoritmos de cifração assimétrica têm altíssimo valor comercial: Toda hora, seguram e certificam milhões de transações financeiras na internet; quanto mais seguros os algoritmos, tanto mais as transações.

Criptografia Simétrica

Distinguimos entre

- a criptografia histórica (necessariamente simétrica antes dos anos 1970) e
- criptografia simétrica moderna.

A criptografia (simétrica) histórica trata textos e dispõe de duas operações:

- permutação (das letras do alfabeto), e
- transposição (das letras do texto claro)

Por exemplo, a `Enigma` usada pela Alemanha na Segunda Guerra Mundial usava apenas o primeiro recurso, a substituição das letras do texto claro pelas letras permutadas do alfabeto.

A criptografia simétrica moderna trata dados, isto é, bites, baites ou números em geral em blocos (Cifras de Feistel como o `AES`) e itera estas duas operações,

- permutação (das letras do alfabeto), e
- transposição (das letras do texto claro)

além de aplicar uma operação matemática como a multiplicação de matrizes, para obter ótima difusão (isto é, filosoficamente, cada letra do texto cifrado depende de todas as letras da chave e do texto claro).

Criptografia Assimétrica

A primeira parte trata a questão ubíqua da *confiança*. Como a criptografia assimétrica permite obter a chave por uma fonte anônima, como garantir que seja confiável? Quer dizer, como evitar o perigo do ataque do `man-in-the-middle` em que o atacante se interpõe entre os correspondentes, assumindo a identidade de cada um, assim observando e interceptando suas mensagens?

Na segunda parte, tratamos das ideias matemáticas por trás da implantação dos algoritmos de criptografia assimétrica:

- quanto à teoria matemática, orientamo-nos ao livro Hoffstein, Pipher, e Silverman (2008) que se contenta com os conhecimentos algébricos do ensino médio;
- quanto às implementações dos algoritmos apresentados, usamos o livro de Sweigart (2013).

Introduzimos as ferramentas básicas para tratar a criptográfica assimétrica, que é matematicamente mais rica: Por trás de todos os algoritmos criptográficos é o problema da insolubilidade computacional do logaritmo discreto, o que muda a figura é o grupo sobre o qual ele opera. Os exemplos principais são

- o grupo multiplicativo de um corpo finito, o caso clássico, e
- o grupo aditivo de uma curva elíptica, o mais recente.

O Grupo Multiplicativo de um Corpo Finito. Estudamos primeiro o caso clássico, o grupo multiplicativo de um corpo finito: Revisaremos as bases matemáticas, números primos, aritmética modular e corpos finitos, para então tratar os algoritmos mais antigos (dos anos 80) baseados nelas: os algoritmos

- ElGamal, que é a implementação a mais próxima do primeiro esboço de um sistema assimétrico delineado por Diffie e Hellman (1976),
- RSA por Rivest, Shamir, e Adleman (1978), a sua implementação predominante.

O Grupo Aditivo de uma Curva Elíptica. Concluímos com o caso mais recente, o grupo aditivo de uma curva elíptica: As suas bases matemáticas são mais envolvidas, com efeito, baseiam-se na teoria dos corpos finitos.

Criptomoedas

Os meios criptográficos para implementar uma criptomoeda como o `Bitcoin` são

- a *criptografia assimétrica* para assinar as transações (mais especificamente, a criptografia assimétrica por curvas elípticas finitas), e
- as *funções de hash criptográficas* para
  - endereçar as transações, e

– dificultar a extensão da *blockchain* (em particular, a sua modificação maléfica), o livro de registro público das transações mantido pelos computadores.

Roteiro

0. O primeiro capítulo não-numerado, Introdução,

   - define a criptografia,
   - resume a sua História, dos tempos antigos aos tempos modernos, e
   - orienta o leitor sobre a divisão dos capítulos.

   O segundo capítulo não-numerado, Notações, é um glossário das noções e notações usadas.

1. Este capítulo estabelece a diferença entre a criptografia *simétrica*, usada desde a antiguidade, e a criptografia *assimétrica*, inventada nos anos 70. Em particular, destaca a importância da assinatura digital, possibilitada pela última. Mostra como quaisquer dados, sejam textos ou imagens, se codificam em arquivos digitais, isto é,

   - *sequências de bits* (o ponto de vista útil para a criptografia simétrica) ou, equivalentemente,
   - *números* (o ponto de vista útil para a criptografia assimétrica).

2. Este capítulo dá os exemplos históricos da criptografia (simétrica) dos romanos e espartanos que servirão como protótipos para algoritmos modernos.

   Discute a segurança destes algoritmos e mostra como podem ser quebrados por regularidades estatísticas apesar de um grande número de chaves.

   Desmantela a Enigma, máquina criptográfica usada pelos eixos de potências, em particular, pelos alemães durante a Segunda Guerra Mundial. Delineia como ela foi quebrada pelos aliados apesar de ser teoricamente segura.

3. Este capítulo apresenta AES, o atual algoritmo padrão da criptografia simétrica, e mostra como os algoritmos antigos servem como suas pedras angulares. Motiva porque ele é considerado imune contra os ataques mais valentes conhecidos. Explica em particular o ataque da *Criptoanálise Diferencial* e mostra a imunidade do AES contra ela.

4. Este capítulo sobre a criptografia assimétrica discute

   - o seu funcionamento,
   - as vantagens e limitações,
   - as suas manifestações no nosso dia-a-dia, como a cifração de sites seguros no navegador.

   Salienta a indispensabilidade da *assinatura digital* para assegurar toda negociação (financeira) online.

   Orienta sobre as melhores práticas para o gerenciamento das chaves públicas e privadas; em particular, como guardar as chaves privadas da forma mais segura possível.

   Introduz ao uso do o atual padrão GPG para criptografia assimétrica, e apresenta uns aplicativos, tais como mensageiros, que a usam (entre eles WhatsApp).

5. Este capítulo estimula o leitor a estudar a base de toda a teoria por trás da criptografia assimétrica, a Aritmética Modular, pela sua utilidade na criptografia assimétrica.

6. Este capítulo dá os passos do protocolo criptográfico *assimétrico* mais antigo, dos anos 70, a troca de chaves por Diffie-Hellman, e explica porque é considerado seguro até hoje.

7. Este capítulo prepara a estrada para o algoritmo assimétrico mais usado até hoje, o RSA, pela apresentação do Algoritmo de Euclides, uma iterada divisão com resto, em que ele se baseia.

8. Este capítulo dá

   - os passos do primeiro algoritmo assimétrico, o RSA, e explica porque é considerado seguro até hoje, e

    - os do algoritmo `ElGamal` que se baseia no Diffie-Hellman e serve como protótipo para muitos protocolos de assinatura como o `DSA` e `EC-DSA`.

9. Este capítulo introduz a criptografia pelas *curvas elípticas finitas*, o novo padrão da criptografia assimétrica e atualmente considerado mais eficiente. Compara as suas vantagens e inconveniências com o Diffie-Hellman ou `RSA`.

10. Este capítulo prepara as ferramentas principais para as criptomoedas baseadas na *Prova de Trabalho* (tal como o `Bitcoin`) pela introdução das *funções de embaralhamento* ou *funções hash*. Os seus valores servem como (carteiras de) identidade de arquivos. Existem vários usos delas na computação; logo, existem diferentes tipos delas com diferentes finalidades os quais apresenta todas.

11. Este capítulo explica

    - o conceito de uma criptomoeda e
    - o funcionamento da criptomoeda pioneira `Bitcoin`:
        - O que é uma cadeia de blocos,
        - de que consiste um bloco, e
        - como os `Bitcoins` são (seguramente) guardados e transferidos?

Livros de Referência

Excelentes livros de referências para acompanhar o curso são

- o clássico Menezes, Oorschot, e Vanstone (1997), gratuitamente disponível no endereço http://www.cacr.math.uwaterloo.ca/hac/, e
- o compêndio mais recente Joux (2009).

Links

Lembremo-nos de que, mesmo se o link `URL` expirou, o conteúdo é ainda disponível pelo `archive.org` no endereço https://web.archive.org/web/URL/ . Por exemplo, o plotter de funções sobre um corpo finito http://graui.de/code/ffplot/ ficará disponível sob https://web.archive.org/web/http://graui.de/code/ffplot/ .

Neste caso, a página foi arquivada no dia 5 de maio 2018; o seu estado nesta data continua a estar disponível em https://web.archive.org/web/20180505102200/http://graui.de/code/ffplot/ .

Notações

Esta secção explica a linguagem matemática básica, conjuntos e funções. Como as notações usada na matemática acadêmica diferem em uns pontos das do ensino escolar, repitamos as mais comuns:

Conjuntos

A noção fundamental da matemática é a de um *conjunto*; segundo os dicionários Aurélio, Houaiss ou Michaelis:

- qualquer coleção de seres matemáticos,
- qualquer reunião de objetos, determinados e diferenciáveis, quer esses objetos pertençam à realidade exterior, quer sejam objetos do pensamento, ou
- qualquer reunião das partes que constituem um todo.

Esta coleção de seres matemáticos é denotada por { seres matemáticos }. Se um ser matemático a pertence ao conjunto A, dizemos que a é em X e escrevemos $a \in \mathrm{A}$ ou $\mathrm{A} \ni a$.

Por exemplo, o conjunto

- dos *números naturais* $\mathbb{N} = \{1, 2, 3, \ldots\}$ é denotado por $\mathbb{N}$.
- dos *números inteiros* $\mathbb{Z} = \{-2, -1, 0, 1, 2, \ldots\}$ é denotado por $\mathbb{Z}$,
- dos *números racionais* $\{x/y$ para x, y em $\mathbb{Z}\}$ é denotado por $\mathbb{Q}$; por exemplo, $2/3$ é nele , e
- dos *números reais* $\{a_{\mathrm{N}}10^{\mathrm{N}} + \cdots + a_1 10 + a_0 + a_{-1}10^{-1} + \cdots\}$ para a_{N}, …, a_1, a_0, a_{-1} em { 0, 1, …, 9 } é denotado por $\mathbb{R}$; por exemplo, $\pi = 3{,}14159\ldots$ é nele.

Podemos comparar dois conjuntos A e B:

- denote $\mathrm{A} \supseteq \mathrm{B}$ que A contém (ou inclui) B, e
- denote $\mathrm{A} \subseteq \mathrm{B}$ que A é contido (ou incluso) B.

Podemos formar um novo conjunto a partir de dois conjuntos A e B:

- denote $\mathrm{A} \cup \mathrm{B}$ a *união* de A e B, o conjunto dos elementos que pertencem a A *ou* B,

- denote $A \cap B$ a *intersecção* de A e B, o conjunto dos elementos que pertencem a A *e* B;
- se $A \supseteq B$, então denote $A - B$ a *diferença* entre A e B, o conjunto dos elementos que pertencem a A mas *não* a B.

Por exemplo, os *números irracionais* são todos os números reais que não são racionais, isto é, que pertencem ao $\mathbb{R} - \mathbb{Q}$.

Denote $A \times B$ o seu *produto* cujos elementos são (a, b) para a em A e b em B, isto é,

$$A \times B = \{(a, b) \text{ para } a \text{ em A e } b \text{ em B}\}.$$

Se B = A, escrevemos A^2 em vez de $A \times A$, e A^3 em vez de $A \times A \times A$, e assim por diante. Por exemplo, $\mathbb{R}^2$ é o plano (euclidiano), e $\mathbb{R}^3$ é o espaço.

Funções

Fórmula. Uma *função* descreve a dependência entre duas quantidades. No ensino escolar, é usualmente denotada por uma equação

$$f(x) = \text{ expressão em } x$$

onde na expressão figuram

- operações aritméticas $+, -, \cdot, / \ldots$
- sobre números reais, constantes, funções algébricas como x^2 e funções especiais como $\exp(x)$, $\log(x)$, $\text{sen}(x)$, ...

Por exemplo:

1. A distância s percorrida por um objeto na queda livre depende do tempo da queda t: vale $s = g \cdot t^2$ onde $g = 9{,}8\text{m/s}^2$ é a constante da aceleração gravitacional da terra.
2. A forca de atração K entre duas massas m' e m'' depende de m', m'' e a distância r entre elas: Pela lei gravitacional de Newton $K = G(m'm'')/r^2$ onde $G = 6{,}67 \cdot 10^{-11}\text{Nm}^2\text{kg}^{-2}$ é a constante de gravidade universal de Newton.

Porém, existem outras dependências, por exemplo:

3. O preço de uma entrada ao teatro de fantoches depende do número da fila do assento: Esta dependência descreve-se por uma tabela de preços, dando o número da fila 1, 2, 3, ... e o preço correspondente de 100 Centavos, 20 Centavos, 5 Centavos, ...

Definição. Uma *função* f é uma regra que associa a cada elemento x de um conjunto X exatamente um elemento y de um conjunto Y.

Esta associação é simbolicamente expressa por $y = f(x)$. Refiramo-nos

- a x como *variável (independente)* ou *argumento*,
- a y como *variável dependente* ou *valor (da função)*,
- a X como *domínio* da função, e
- a Y como *contra-domínio* da função.

A *imagem* $\mathrm{im} f$ de f é o conjunto de todos os valores de f,

$$\mathrm{im} f = \{ \text{ todos os } f(x) \text{ para } x \text{ em X}\}.$$

Por exemplo, imsen $= [-1,1]$ e $\mathrm{im} f = \{c\}$ para a função constante $f \equiv c$. O contra-domínio contém a imagem, mas não necessariamente coincide com ela,

$$\mathrm{im} f \subseteq \mathrm{Y}.$$

Usualmente, vemos funções com X e Y $\subseteq \mathbb{R}$, isto é, *funções reais*; chamamo-nas muitas vezes simplesmente de funções. Usualmente as letras $a, b, c, \ldots, r, s, t, u, x, y, z$ denotem números reais; as letras i, j, k, l, n, m denotem números naturais (usados para enumerações). Nos nossos exemplos:

1. Aqui (a magnitude) de t em $\mathbb{R}_{\geq 0}$ e $f(t) = s \in \mathbb{R}_{\geq 0}$. Tem-se $\mathrm{X} = \mathbb{R}_{\geq 0}$ e $\mathrm{im} f = \mathbb{R}_{\geq 0}$.
2. Aqui (a magnitude) de m', m'' em $\mathbb{R}_{\geq 0}$ e K em $\mathbb{R}_{\geq 0}$. Tem-se $\mathrm{X} = (\mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0}) \times \mathbb{R}_{\geq 0}$ e $\mathrm{im} f = \mathbb{R}_{\geq 0}$.
3. Aqui $\mathrm{X} = \mathbb{N} = \{1,2,3\}$ e $\mathrm{im} f = \{100, 20, 5\}$.

Funções reais aparecem em diversas formas: além de

- uma equação,

obtemos pela substituição da variável por um número real como 0,5, 1, 2, 1000, ...

- uma tabela de valores x e y dado pela avaliação do lado direito da igualdade $f(x) = ...$, e
- um *gráfico*

    1. pela marcação dos pontos (x, y) no plano que são dados pelos valores da tabela, e
    2. pela conexão destes pontos.

Em mais detalhes:

- Na forma de uma equação como nos exemplos acima: $y = f(x)$, por exemplo, $y = x^2, \text{sen}x \ldots$ Para facilitar, fazemos o convênio seguinte: Se escrevemos apenas a regra de associação, por exemplo, $f(x) = 1/(x(x-3))$ ou $\sqrt{x}$, então o domínio é o subconjunto máximo de $\mathbb{R}$ para que esta regra seja definida (quer dizer, faça sentido). Por exemplo, nestes exemplos, $\mathrm{X} = \mathbb{R} - \{0, 3\}$ e $\mathrm{X} = \mathbb{R}_{\geq 0}$.
- Na forma de uma tabela de valores, por exemplo: Um experimento mede a tensão U de um resistor em dependência da corrente Y. A tabela tem as entradas $\mathrm{Y} = 50; 100; 150, \ldots (m\mathrm{A})$ e $\mathrm{U} = 2; 4, 6; \ldots (\mathrm{V})$. Aqui $\mathrm{X} = \{50, 100, \ldots\}$ (e $\mathrm{Y} = \mathbb{R}_{\geq 0}$). (Gostaríamos de extrapolar esta função, isto é, estender o seu domínio a $\mathbb{R}_{\geq 0}$; a este fim, parece provável que $\mathrm{U} = \mathrm{RI}$ com $\mathrm{R} = 0,04\Omega$.)
- A equação funcional $y = f(x)$ associa a cada valor x um único valor y, em símbolos, $x \mapsto y = f(x)$. Um tal par de valores (x_0, y_0) pode ser interpretado como ponto P no plano cartesiano. Para cada par de valores (x_0, y_0) obtemos exatamente um ponto. Dado um ponto P, os números reais x_0 e y_0 são chamados das *coordenadas (cartesianas)*. O conjunto de todos os pontos $(x, y = f(x))$ forma a *curva da função* (ou *gráfico*), que ilustra o percurso da função $y = f(x)$. (Por exemplo, a parábola $y = x^2$ ou uma função afim $y = ax + b$.)

  Para desenhar a curva:

  1. Enche uma tabela de valores para uns argumentos x', x'', ...
  2. Desenha os pontos dados nela, e
  3. Conecta-os para obter o percurso da curva da função.

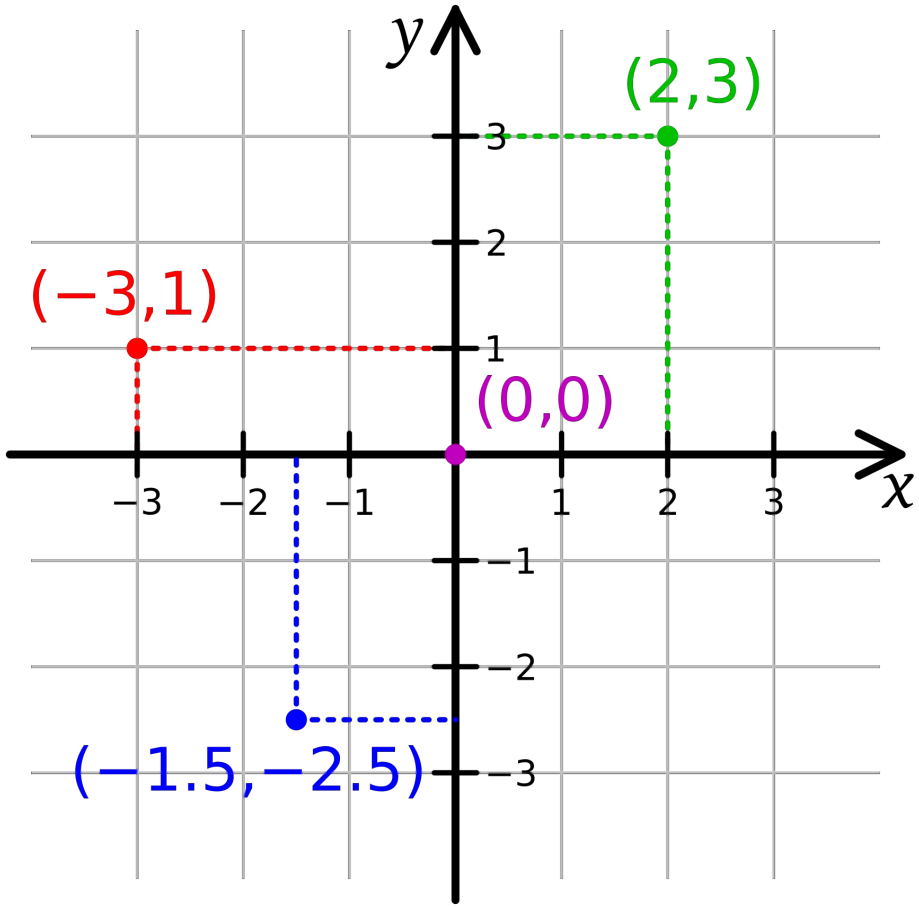


Figura 1: Coordenadas Cartesianas de um ponto nos eixos x e y

De vez em quando, para destacar que um objeto, por exemplo, uma função $h\colon \mathrm{X} \to \mathrm{Y}$, depende de outro objeto, por exemplo, de um conjunto D ou uma ênupla w, escrevemos $h(\mathrm{D})$ ou h_{D} respetivamente $h(w)$ ou h_w em vez de h.

Aplicação. No ensino acadêmico, uma *função* ou *aplicação* é denotada por

$$f\colon \mathrm{X} \to \mathrm{Y}$$

para os dois conjuntos,

- o *domínio* X, e
- o *contra-domínio* Y de f

dizemos que *manda* ou *envia* cada *argumento* x em X a um único *valor* y em Y.

Lê-se que f é uma função de X a Y, ou tem *argumentos* em X e *valores* em Y. No contexto informático, pensamos de f como um algoritmo, e referimo-nos a X e Y como *entrada* e *saída.*

Uma função manda ou envia cada *argumento* x em X a um único *valor* y em Y (ou *associa* a cada x um único y). Escrevemos

$$x \mapsto y$$

e denotemos este valor y por $f(x)$. Frequentemente, $\mathrm{X} = \mathbb{R}$ ou $\mathrm{X} = \mathbb{R} \times \cdots \times \mathbb{R}$ e $\mathrm{Y} = \mathbb{R}$ ou $\mathrm{Y} = \{\pm 1\}$. Por exemplo, a função $f\colon \mathbb{R} \to \mathbb{R}$ dada por $x \mapsto x^2$ manda 2 em $\mathbb{R}$ a $2^2 = 4$ em $\mathbb{R}$.

Função e *aplicação* são sinónimos. Contudo, a conotação é outra: Se os objetas do domínio são, por exeplo, números inteiros, conjuntos, então *aplicação* é mais comum, enquanto função é principalmente usada quando o domínio consiste de (ênuplas de) números reais.

Formar Novas Funções.

Composição. A partir de duas funções f e g (sobre quaisquer domínios e imagens) pode ser obtida outra por *concatenação*; a *função composta* ou a *composição* $g \circ f$, dado que a imagem de f é contida no domínio de g, isto é, $\mathrm{Y}(f) \subseteq \mathrm{X}(g)$. É definida por

$$g \circ f\colon x \mapsto g(x) \mapsto f(g(x)),$$

simbolicamente, para obter $g \circ f(x) = g(f(x))$, substituímos x em $g(x)$ por $f(x)$.

Se temos duas funções,

$$f\colon \mathrm{X} \to \mathrm{Y}, \quad \text{e} \quad g\colon \mathrm{Y} \to \mathrm{Z}$$

isto é, tais que os valores de f são argumentos de g, a sua *composição* é definida por

$$x \mapsto f(x) \mapsto g(f(x)),$$

isto é, a saída de f é a entrada de g, e denotada por

$$g \circ f : \mathrm{X} \to \mathrm{Z}.$$

Inversão de funções. Por definição, uma função $f : \mathrm{X} \to \mathrm{Y}$ associa a cada argumento $x \in \mathrm{X}$ exatamente um valor $y \in \mathrm{Y}$. Frequentemente surge o problema inverso: Dado um valor y, determina o seu argumento x sob f, isto é, tal que $y = f(x)$. Se uma função f é *injetora*, isto é, $x' \neq x''$ implica $f(x') \neq f(x'')$, isto é, a argumentos diferentes são associados valores diferentes, então a cada valor $y \in \mathrm{im} f$ é associado um único argumento $x \in \mathrm{X}$. A função $g : \mathrm{im} f \to \mathrm{X}$ obtida pela associação inequívoca *inversa* $y \mapsto x$, ou $g(y) = x$ é a *função inversa* ou o *inverso* de f e denotada por $g = f^{-1}$. Ora, y é a variável *independente* e x a variável *dependente*. Em fórmulas, a função inversa g é obtida pela permutação das duas variáveis x e y na equação $y = f(x)$.

Matematicamente, a função f tem o inverso $g = f^{-1}$, se $g(f(x)) = x$ e $f(g(x)) = x$. Por exemplo, sobre $\mathbb{R}_{\geq 0}$ vale $\sqrt{(x^2)} = x = (\sqrt{x})^2$ para $f(x) = x^2, g(x) = \sqrt{x}$, e $(2x+1)/2 - 1/2 = x = 2(x/2 - 1/2) + 1$ para $f(x) = 2x + 1$ e $g(x) = x/2 - 1/2$.

Para desenhar a função inversa, poderíamos permutar as designações dos dois eixos. Porém, isto comumente não se faz. Porém, se o domínio e a imagem coincidem, o gráfico da função inversa $g = f^{-1}$ é o espelhamento do gráfico da função invertida f na diagonal dos pontos cuja coordenada x é igual à coordenada y.

Exemplos. Olhemos uns exemplos de funções invertíveis. Observamos em particular que a invertibilidade depende do domínio: Quanto menor o domínio, tanto mais provável que a função seja invertível.

- A parábola $y = x^2$ *não* é invertível sobre $\mathbb{R}$, mas só sobre $\mathbb{R}_{\geq 0}$ e $\mathbb{R}_{\leq 0}$ (pela radiciação quadrática).
- A função $y = 2x+1$ cresce de modo estritamente monótono, em particular, é invertível.
- O seno (que mede o comprimento do cateto oposto no círculo unitário) cresce de modo estritamente monótono no intervalo $[-\pi/2, \pi/2]$; logo tem um inverso, o *arco-seno* neste intervalo.

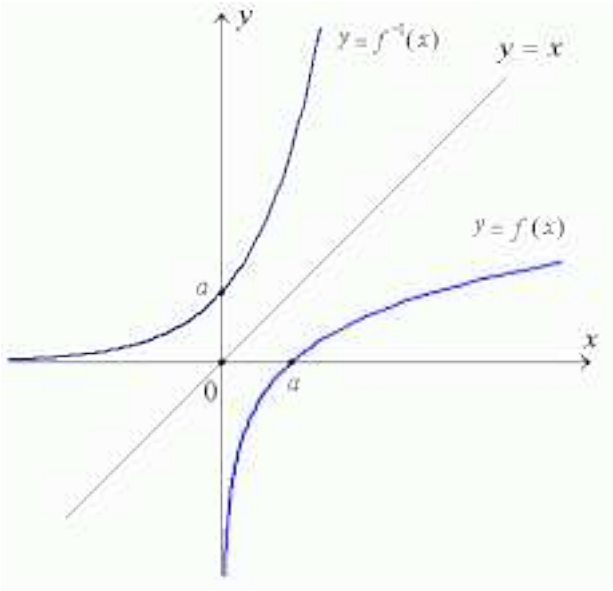


Figura 2: A função inversa

- a exponencial exp é invertível pelo logaritmo log.

Resumimos:

- Toda função estritamente monótona (crescente ou decrescente) é invertível.
- Na inversão o domínio e imagem trocam os papéis.
- Se o domínio e imagem coincidem, obtemos o gráfico da função inversa pelo espelhamento do gráfico da função na diagonal $y = x$.

Algarismos

Um número real no dia-a-dia é escrito em notação decimal

$$a_N \dots a_1 a_0, a_{-1} \dots$$

com a_N, …, a_1, a_0, a_{-1} …, em $\{0,1,\dots,9\}$. Isto é, é como soma em potências de 10 e com algarismos 0, …, 9. Por exemplo,

$$321,34 = 3 \cdot 10^2 + 2 \cdot 10 + 1 + 3 \cdot 10^{-1} + 4 \cdot 10^{-2}.$$

Em vez da *base decimal* $b = 10$, existem outras. As mais comuns na informática são

- a *base binária* $b = 2$ com os algarismos 0 e 1, e
- a *base hexadecimal* $b = 16$ com os algarismos $0, \ldots, 9$ e A, B, C, D, E e F (que correspondem a 10, 11, 12, 13, 14 e 15).

Isto é,

- no conta-quilómetro binário, o último algarismo retorna à posição inicial após dois quilómetros, o penúltimo após $2^2 = 4$ quilómetros, e assim por diante,
- no conta-quilómetro hexadecimal, o último algarismo retorna à posição inicial após 16 quilómetros, o penúltimo após $16^2 = 256$ quilómetros, e assim por diante.

Por exemplo,

$$1101 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

e

$$\text{A02F} = 10 \cdot 16^3 + 0 \cdot 16^2 + 2 \cdot 16^1 + 15 \cdot 16^0.$$

1 Criptografia

A criptografia serve a proteger dados, e consegue isto por um embaralhamento (cifração) que praticamente pode unicamente ser invertido (decifração) com uma informação adicional secreta, a chave.

Outrora, a criptografia estudou a transformação de um

texto inteligível

$\Downarrow$

texto ininteligível

tal que só uma informação adicional secreta, a *chave*, permita desfazê-la.

Hoje em dia, a criptografia estuda a transformação de

dados inteligíveis

$\Downarrow$

dados ininteligíveis

tal que só uma informação adicional secreta, a **chave**, permita desfazê-la.

dados	=	arquivo digital (de texto, imagem, som, vídeo, ...)
	=	sequência de bites (= 0, 1)
	=	sequência de baites (= 00, 01, ..., FE, FF)
	=	número (= 0, 1, 2, 3 ...)

Isto é, toda sequência de bites 1011... é uma número n pela sua expansão binária $n = 1 + 0 \cdot 2 + 12^2 + 12^3 + \cdots$ e vice-versa.

- O ponto de vista de uma sequência de bites (ou, mais exatamente, de dígitos hexadecimais cujos dezasseis símbolos 0 – 9 e A – F correspondem a um grupo de quatro bites) é preferida na criptografia simétrica cujos algoritmos transformam-nas, por exemplo, por permutação e substituição dos seus dígitos.

- O ponto de vista de um número é preferida na criptografia assimétrica (com duas chaves) cujos algoritmos operam nele por funções matemáticas como a potenciação e exponenciação.

1.1 Dados

Dados referem-se classicamente, antes da época digital, a textos, hoje em dia a qualquer arquivo digital, sejam textos, imagens, sons, e assim por diante. Nos nossos exemplos, sempre suponhamos que os dados sejam um texto. Quando assumirmos o ponto de vista matemática, os dados referem-se a um número natural, isto é, um elemento em 0, 1, 2, 3, ...

Mostramos pelos exemplos de textos e imagens como esta codificação, dos dados em (sequências de) números se faz: Recordemo-nos de que

- um *bite* é um dígito binário, isto é, uma das 2 possibilidades 0 ou 1, e
- um *baite* a sequência de 8 bites, isto é, uma das $2^8 = 256$ possibilidades entre $0 = 00000000$ e $255 = 11111111$.

Codificação de Textos. Um texto é uma sequência de símbolos. Há várias codificações que associam a cada símbolo um número; as duas mais conhecidas são

- a `ASCII`, que cobre todos os caracteres ingleses (e os símbolos de pontuação) e que envia um símbolo a um baite,
- a `UTF-8`, que inclui a `ASCII` e cobre todos os caracteres de todos os idiomas (por exemplo, do chinês, coreano, ... e além disso, por exemplo, todos os símbolos matemáticos) e que envia um símbolo a 1, 2, 3 ou até 4 baites.

Por exemplo,

$$0 \mapsto 48, ..., 9 \mapsto 57; \mathrm{A} \mapsto 65, ..., \mathrm{Z} \mapsto 90; a \mapsto 97, ..., z \mapsto 122.$$

Com efeito, o `ASCII` somente predetermina a codificação dos números $0 - 127$, isto é, todas as sequências de bites que começam com `0`. Depois do `ASCII`,para internacionalizá-la, surgiram inúmeras codificações (por exemplo,

- `ISO-8859-1` ou `ISO-Latin-1` para os alfabetos latins na Europa ocidental e

- `ISO-8859-5` para os alfabetos cirílicos usados na Europa oriental (por exemplo, na Bulgária ou Sérbia) e na Rússia.

Todas elas ocuparam o espaço dos números $128 - 255$ para acrescentar os símbolos do próprio alfabeto, isto é, todas as sequências de bites que começam com 1.

A codificação UTF-8 teve uma abordagem mais engenhosa: O número dos dígitos binários consecutivos iniciais iguais a 1 nos primeiros 3 bites, acrescentado por 1, iguale o número de baites, de 1 a 4, do símbolo codificado. Isto é, ela envia um símbolo a 1, 2, 3 ou até 4 baites, onde

$$\#\{\text{baites}\} = \#\{\text{dígitos binários consecutivos iniciais iguais a } 1\} + 1$$

Por exemplo, o símbolo ç, a letra c com uma cedilha, tem 2 baites e

$$\text{ç} \mapsto 1011001111000111.$$

Vemos que, com efeito, o número dos primeiros coeficientes consecutivos iguais a 1 é 1 (como já o segundo coeficiente é igual a 0), o que, acrescentado por 1, resulta em 2, o número de baites de ç.

Codificação de Imagens (por um *bitmap* = mapa de graus das cores primárias). Para imagens, o formato mais simples é o de um bitmap, um mapa de bites. Uma imagem

- é um conjunto de pixeis (por exemplo, 1024×768),
- cada pixel é uma cor, e
- cada cor é os seus graus de intensidade da luz das três cores primárias
  - Vermelho (= *R*ed),
  - Verde (= *G*reen) e
  - Azul (= *B*lue).

  O homem não consegue distinguir mais de 256 graus de cada cor primária,

  $\Longrightarrow$ basta codificar cada pixel por três baites.

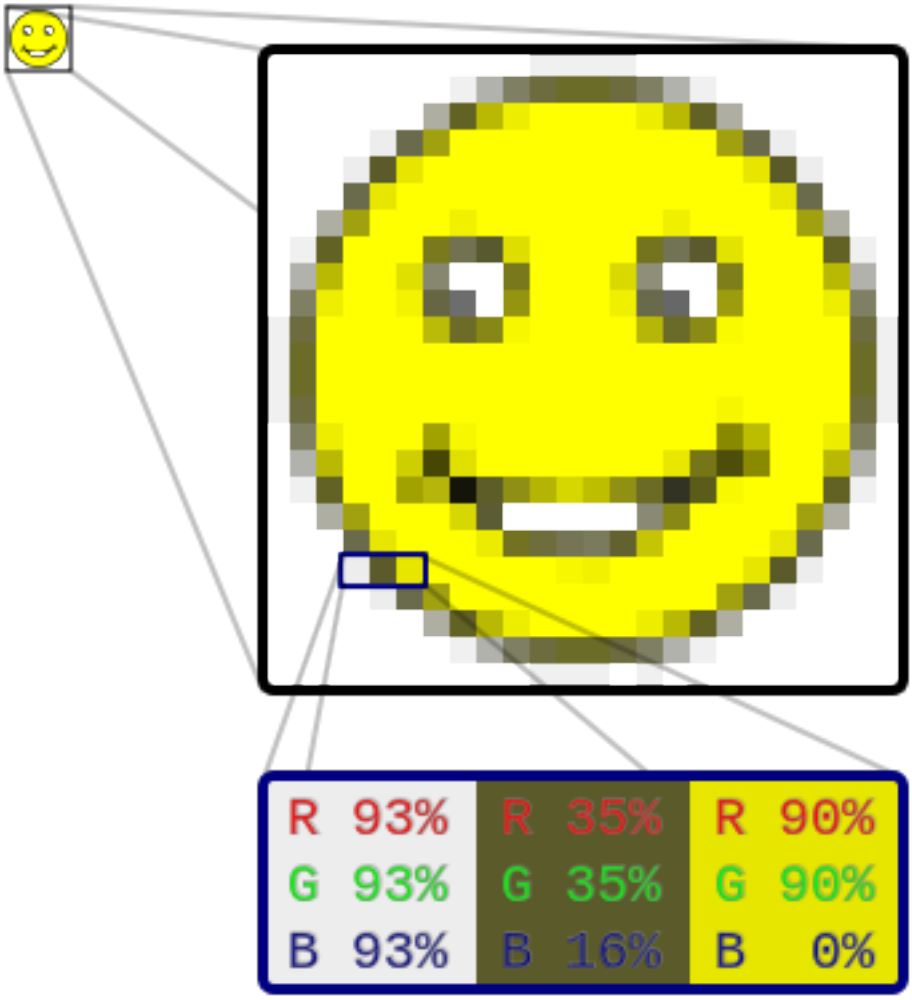


Figura 3: Mapa de graus das cores primárias (RGB = Vermelho, Verde e Azul)

1.2 Chaves: A criptografia simétrica e assimétrica

Recordemo-nos de que a *chave* é a informação adicional secreta que pode ter várias formas a qual é sobretudo uma questão da conveniência. As mais usais são a

- de um número,
- de uma sequência de letras, por exemplo,

– uma senha, ou
– uma frase secreta (com espaços).

Por exemplo, no algoritmo antigo da *Cítala* ou *bastão de Licurgo*, a chave consiste da circunferência (em letras) do bastão usado, um número. Hoje em dia, um código `PIN` (= Número de Identificação Pessoal) ou senha são omnipresentes no nosso dia-a-dia; para facilitar a decoração, para datas muito sensíveis, é encorajada a decoração de completas frases (= múltiplas palavras) secretas.

A criptografia assimétrica depende de chaves maiores e por isso as armazena em arquivos (de textos com 64 letras, chamados de `ASCII`-armor) de alguns quilobaites.

Historicamente, a chave para inverter a transformação (de dados inteligíveis em dados ininteligíveis) era tanto necessária para decifrar quanto para cifrar, a *criptografia simétrica*. Já foi usada pelo egípcios quase 2000 anos antes de Cristo.

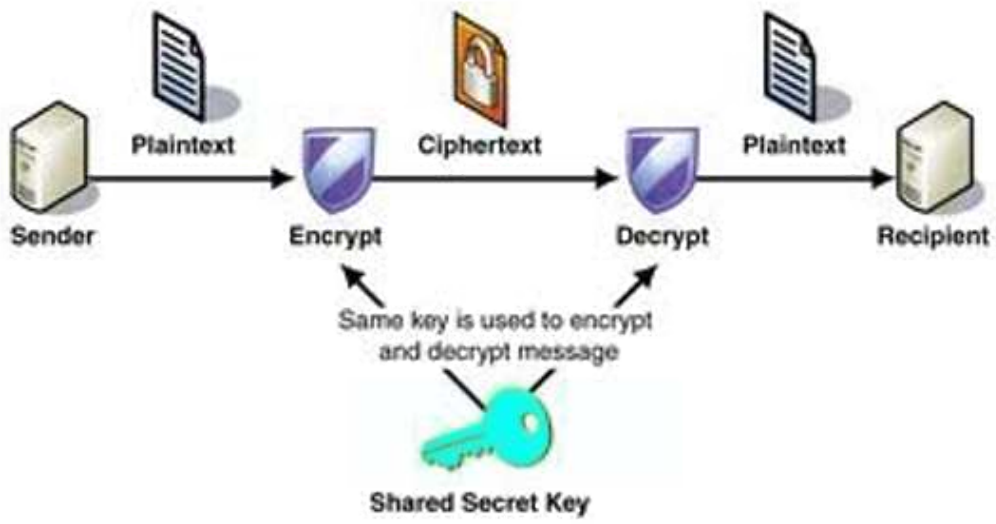


Figura 4: criptografia simétrica

Nos anos 70, surgiu a *criptografia assimétrica*, na qual as chaves para cifrar (*a chave pública*) e decifrar (*a chave privada*) são diferentes.

Com efeito, só a chave para decifrar é privada, guardada secreta, enquanto a para cifrar é pública, conhecida a todo mundo. (Porém, é possível, e muitíssimo

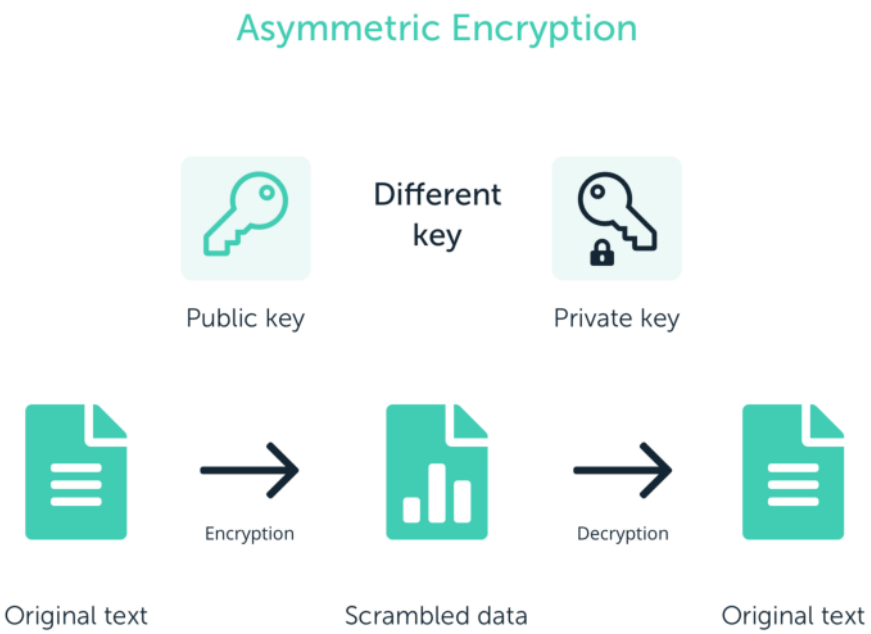


Figura 5: criptografia assimétrica

útil, que as chaves trocam os seus papéis, a chave privada cifra e a pública decifra, a *assinatura digital*, explicada mais em baixo.)

Em comparação a criptografia simétrica, a criptografia assimétrica evita o risco de comprometimento da chave para decifrar envolvido

- na troca da chave com o cifrador, e
- na posse da chave do cifrador (além do decifrador).

Em particular o primeiro ponto, que permite comunicar *seguramente* com qualquer pessoa através de um canal *inseguro*, é uma grande vantagem em comparação à criptografia simétrica! Tão grande que, enquanto antes da era da internet a criptografia assimétrica era impensável, depois a internet ficou impensável sem a criptografia assimétrica.

Tamanho da Chave. A um nível de segurança comparável, os algoritmos da criptografia simétrica são mais econômicos que os da criptografia assimétrica.

Aqui mais *econômicos* quer dizer que os algoritmos simétricos

- são mais *rápidos*, e
- usam chaves *menores*. Por exemplo, o algoritmo assimétrico RSA usa chaves de pelo menos 2048 bites, enquanto o algoritmo simétrico AES usa chaves de 128 bites. (Esta diferença será explicada em Seção 9.5.)

Esta diferença se reflete pelo gerenciamento entre uma chave assimétrica ou simétrica:

- Uma chave *assimétrica* é armazenada em um *arquivo* (por exemplo, como arquivo de texto em um disco rígido, ou como arquivo em um cartão inteligente), e este arquivo é comumente assegurada por uma frase secreta (passphrase) que o usuário decora.
- Uma chave *simétrica* é decorada como senha, isto é, guardada na memória do usuário. (Na prática, esta senha é transformado por um hash criptográfico, como o MD5 ou SHA-256 a uma sequência de byes de um comprimento determinado, por exemplo, de 32 baites. Padroniza o comprimento da chave e cifra para evitar uma leitura por outra pessoa.)

Exemplo de uma chave pública codificada em ASCII:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: SKS 1.1.6
Comment: Hostname: pgp.mit.edu

mQENBFcFAs8BCACrW3TP/ZiMRQJqWP0SEzXqm2cBZ+fyBUrvcu1fGU890pd43J diW
IreHx/sbJdW1wjABeW8xS1bM67nLW9VVHUPLi9QP3VGfmqmXqbWIB7OxizZ PTDCWm
oymm/+TlTTAZWU6Wwvmjk88QlmU941tUvBsQ1cw1cAxw+2jLCgkz8XvW npMPKKj1f
sNZ/FcPVMC6dkwHAFc7Rm4DNibJzLvD8woL0vAdUR4HhOQli9+Fpv UO0KVVhOwF0f
14EURddA4qZVyPM8e3FvxiWF0JWJuxCuiBHh5ghT/Q+OQMMOJW VTwME5nQ87vohfX
gjRbYjW8gyLRqIjt2Gc7dNgIoKE5r/PABEBAAG0I0Vubm8g TmFnZWwgPGVubm8ubm
FnZWxAdC1vbmxpbmUuZGU+iQE5BBMBAgAjBQJXBQLPAh sDBwsJCAcDAgEGFQgCCQo
LBBYCAwECHgECF4AACgkQVJDsz4ujnoR03gf9HIIc pOSI1yokf6JZjH/Oon+FDoJd
7i7B1wfMyOKmSDsTbrJqimi7s8R9hsSljYuf/s TzMbGGCoHkOfnZyGuv3HovTO90x
g5JSsCTU/DhojHqANODJ23lTZFgW0vOmxkL KpRKOPiZ+xX3z9PjDNULFCBLV4s2+m
UABKbkKJLecytE8g4WWDkV7ePHrkTZyS sAoaNvLW6f0BxGmO0L4Igf35UG2ZbzMah
m8eFu6ADr3gQKO1DaiOhQsF5oInGq ymyaZyNCMR0CohWZuGT1DLVFUOzGos1p42Tl
BqylUJGw8ll9xeQJevAMf2xdrl DyRNxyy6ojOjXMzmf32CPQpI7JN7kBDQRXBQLPA
```

```
QgAwL9sFKBf2h8KLjHkfSXD EMMVgKdifO9J6F0ubS8l4vXj+DlnJusjxa7oLMIrxp
BT35y06pohtqwKMsxGEZ Uvx9MKk1NdU5OksPyvoSSVcIm4APzC/1pSFaHUjlWp3HI
4PC5yFBo3IMqIajtu PyYOB3A8O5iIIm8ip1BvEVHTruAOz/1wIFr+xgFmglqU2XvO
coDHz442h71CD0 2iDVjQyO8Cmb87AmSc+dV4iXUalK5+GQRe21lAhIB2jVprdFvx6
VJvsF7Xf+RB 2GsZ61v4glGgYFaXLDeRYwyearjGuE2thLo9RdGu2/gfyJFwij43Lc
7kGX1/rI YHOTPJTmv5QQARAQABiQEfBBgBAgAJBQJXBQLPAhsMAAoJEFSQ7M+Lo56
EoLMH /3AMzxXON2m0rXwFVnBStaYGAC7bQilAJdgoiYASAxS2KHphpvHQ8Y6BUOHv
xG Wp4v350kZkiMGDblOe63/6mFHizFg/PTxeRDJLS7hWp1RUASd47hmBbrjFDRHu
4i1mYHrHvQy6QXSO6z88fStgYFWcer+JALtlnJCs8cQ67wMRxjZPEjUj1uGrm6 skK
Y5LBJvTCj4GN3vPVExvmLvRXPgS0pgiCYPdqbxUY97VT9jzsHXeCmmNJO8t oUnlH0
HpIcNfP76d3vdBwnAkCarnxqcCPBMZ+0lJWYUeqmlRTIBsIMRTOpTi3x fdbVBxlb+
wJ/p8VnJSURox3xoxHaJCHeU==jKED

-----END PGP PUBLIC KEY BLOCK-----
```

Para comparar, o hash `md5` (de 16 baites) da palavra “chave” em codificação hexadecimal (isto é, o alfabeto `0`, …, `9`, `A`, `B`, `C`, `D`, `E`, `F`) é

```
3fc75801d131d80c89186ce5d064dc2ba3.
```

Considerações Práticas.

- Para aproveitar ambas as vantagens da criptografia simétrica e assimétrica, comunica-se frequentemente (por exemplo, ao navegar a um site seguro [pelo protocolo `TSL`])
  - primeiro com chaves assimétricas (para trocar uma chave simétrica), e
  - depois com uma chave simétrica.
- Na criptografia assimétrica, a comunicação pessoal da chave pública, pelo grande tamanho da chave pública, é inviável para o ser humano, Por isso, comumente
  - obtém-se a chave pública eletronicamente (e-mail, site, …), e
  - verifica-se por uma soma de verificação criptográfica (por exemplo, soletrando-a no telefone).

- Mesmo se uma chave assimétrica é desnecessária para um usuário cifrar por exemplo o seu disco rígido, as vezes revela-se na prática vantajosa: Atualmente só chaves assimétricas podem ser armazenadas em um cartão inteligente. Para decifrar o disco rígido cada vez que se liga o notebook,
  - em vez do usuário entrar a sua frase secreta (comprida),
  - é mais prático entrar o seu cartão inteligente no leitor.

  Assim, o usuário é mais inclinado a desligar (e assim cifrar) o seu notebook quando não o usa (quer dizer, quando está em maior risco).

Assinatura Digital. A chave pública e privada podem inverter os seus papéis: Assim só o dono da chave privada pode cifrar o texto, e todo mundo (que conhece a chave pública) pode decifrá-lo.

O que parece fútil é com efeito um uso (economicamente) importantíssimo da criptografia assimétrica. Ele chama-se *assinatura digital*, e serve para identificar a fonte de dados na internet: Com efeito, como só o dono da chave privada pode cifrar o texto tal que a chave pública o decifre, todo mundo que conhece a chave pública pode verificar que o texto cifrado provém do dono da chave privada.

1.3 Segurança

A criptografia assimétrica usa métodos matemáticos, mais exatamente aritmética modular, para cifrar. A segurança (= a dificuldade da decifração) da criptografia assimétrica é baseada em problemas matemáticos reconhecidos dificílimos há séculos.

A criptografia simétrica (como as funções de hash) usa métodos de cifrarão mais artesanais, que, de aparência, visam maximizar a difusão e confusão, principalmente por substituição e permutação (vide Seção 1.3). A segurança da criptografia simétrica é simplesmente baseada na renitência de falhar diante ataques de criptógrafos durante anos. Isto é, não é satisfatório de um ponto de vista transcendente, mais pelo menos prático. (A grande maioria das ciências, por exemplo, a medicina, confia quase exclusivamente nas estatísticas como fonte de conhecimento.)

Critérios para Segurança de um Algoritmo Criptográfico. O *princípio de Kerckhoff* postula que o

- algoritmo seja **público**.

Enquanto o conhecimento da chave compromete uma única cifração, o conhecimento do algoritmo compromete todas as cifrações. Um algoritmo público garante a dificuldade da decifração depender só do conhecimento da *chave*, mas não do do *algoritmo*. Quanto mais usado, tanto mais provável que o algoritmo será conhecido. Para ele ser útil, necessita ser seguro mesmo sendo público.

As *metas de Shannon* da

- *Confusão* respectivamente
- *Difusão*

desejam **ofuscar** a **relação** entre o texto cifrado e

- a **chave** respectivamente
- o **texto claro**

Idealmente, quando uma letra da chave respectivamente do texto claro muda, a metade do texto cifrado mude, isto é, cada letra do texto cifrado mude com uma probabilidade de *50%*. Enquanto a saída da cifração, o texto cifrado, depende deterministicamente da entrada, do texto claro, e da chave, o algoritmo visa *ofuscar* esta relação no sentido de torná-la *tão complicada*, entrelaçada, embaralhada *quanto possível*: cada letra da saída, do texto cifrado, depende de cada letra da entrada, do texto claro, e da chave.

Uma boa confusão ou difusão dificulta ataques estatísticos sobre

- ou muitas *chaves* cifrando o mesmo texto claro,
- ou muitos *textos claros* cifrados pela mesma chave.

Segurança Perfeita. Textos claros geralmente não ocorrem com a mesma probabilidade. Ela depende, por exemplo, do idioma, do jargão ou do protocolo usado.

Moralmente, um método de criptografia é *perfeitamente seguro* se um texto cifrado gerado com ele não permite tirar conclusões sobre o texto claro correspondente.

Isto é, a probabilidade que um texto claro e uma chave resultaram no texto cifrado é a mesma para todos os textos claros e todas as chaves.

Denote p um texto claro e $\mathcal{P}(p)$ a sua probabilidade.

Um método de criptografia é chamado *perfeitamente seguro* se, para todo texto claro, a sua probabilidade é (estocasticamente) independente de qualquer texto cifrado. Em fórmulas, para todo texto claro p e todo texto cifrado c, temos $\mathcal{P}(p|c) = \mathcal{P}(p)$.

Se um atacante interceptar um texto cifrado c, então ele terá nenhuma vantagem, isto é, a sua probabilidade de obter o texto claro é a mesma como se ele não conhecesse c.

Teorema de Shannon. Em 1949, Shannon provou o seguinte teorema, que explica sob quais condições um método de criptografia é perfeitamente seguro:

Seja finita o número de chaves e de texto claros e $\mathcal{P}(p) > 0$ para todo texto claro p. O método de criptografia é *perfeitamente seguro*, se

- a distribuição de probabilidade no espaço das chaves é a distribuição uniforme e
- para cada texto claro $p \in \mathcal{P}$ e todo texto cifrado c existe uma única chave k para obter c a partir de p.

Isto é, desvios estatísticos tendem a enfraquecer o criptossistema. Em particular, é importante usar um gerador de números totalmente aleatório para as chaves.

One-time pad. Um criptossistema perfeitamente seguro é o one-time pad em que uma chave (do mesmo tamanho que o do texto claro) é adicionada (bite a bite) ao texto claro (vide). Vide Seção 3.1.

Usar um tal criptossistema perfeitamente seguro é geralmente complicado na prática. Para aplicações em tempo real, como na Internet, eles são pouco usados.

Segurança Provada. Como segurança perfeita é inviável, a segurança é demonstrada

- ou por demonstração da resistência contra os ataques conhecidos,
- ou por redução à dificuldade computacional de um problema matemático (chamada de *segurança comprovada*).

Embora existam criptossistemas simétricos comprovadamente seguros (tal como o gerador pseudo-aleatório de Blum-Blum-Shub cuja segurança é redutível ao problema da computação do Resíduo Quadrático), os algoritmos mais eficientes e mais usados, tal como o AES, comprovam a sua resistência apenas contra ataques conhecidos, tais como as da criptoanálise diferencial ou linear.

Segurança Formalmente Provada. Os problemas usados pela criptografia assimétrica são todos NP: Todo algoritmo criptográfico cifra ou decifra uma mensagem *com* a chave em tempo polinomial no tamanho da chave em bites. Por outro lado, todos os algoritmos conhecidos para cifrar ou decifrar *sem* a chave levam tempo exponencial no tamanho da chave em bites.

Por enquanto, a conjectura sendo irresolvida, existem teoricamente algoritmos polinomiais para cifrar ou decifrar *sem* a chave em tempo polinomial; contudo, praticamente, após décadas de esforços contínuas da comunidade de criptógrafos em vão, parece improvável.

Exemplo. O exemplo inicial de um tal criptossistema comprovadamente seguro foi em 1982 o de Goldwasser e Micali para a segurança semântica pela redução ao problema do Resíduo Quadrático.

Dados x e N um produto de dois primos, é difícil determinar se x é quadrático módulo N (isto é, se existe y tal que $x = y^2 \bmod N$ ou não) se, e tão-somente se, o assim-chamado *símbolo de Jacobi* para x é +1 e se os fatores primos de N são desconhecidos.

O criptossistema de Goldwasser-Micali consiste em:

- um algoritmo de *geração de chaves* que produz
  - a chave privada como dois primos p e q, e

  - a chave pública $N = pq$ e um número x que é quadrático nem módulo p, nem módulo q (tal que o símbolo de Jacobi de x para N é 1, e para ambos os seus fatores p e q é -1). Por exemplo, se $p, q \equiv 3 \mod 4$, então $x = N - 1$ serve.

- um algoritmo de *cifração probabilístico*: Se $m = (m_1, m_2, \ldots)$ são as bites da mensagem clara, então são gerados números $y_1, y_2, \ldots$ indivisíveis por p e q e a mensagem cifrada é $M = (M_1, M_2, \ldots)$ com $M_1 = y_1^2 x^{m_1}$, $M_1 = y_1^2 x^{m_1}$, $\ldots$
- um algoritmo de *decifração determinístico*: Se $M = (M_1, M_2, \ldots)$ é a mensagem cifrada, então $m_1 = 0$ se, e tão-somente se, M_1 é quadrático módulo N, $\ldots$ o que é rapidamente determinado pelo conhecimento de ambos os fatores p e q de N.

Logo em seguida, em 1985, surgiu criptossistema o algoritmo ElGamal, que foi provado semanticamente seguro pela redução ao problema de Diffie-Hellman Decisório. Isto é, a segurança semântica (`IND-CPA`) do criptossistema ElGamal é provada sob a hipótese da dificuldade do problema de Diffie-Hellman Decisório que é uma variação do (dificílimo) problema da computação do logaritmo discreto (na aritmética modular); vide Seção 6.2 para mais detalhes. Isto é, prova-se que ganhar o jogo do `IND-CPA` é pelo menos tão difícil quanto o problema de Diffie-Hellman suposições de segurança com um fator polinomial.

Paradoxo. *Cautela:* A segurança teórica permanece uma idealização insuficiente para a realidade: Por exemplo, Ajtai e Dwork apresentaram em 1997 um criptossistema e o provaram teoricamente seguro pela redução a um problema difícil. Um ano depois, em 1998, este foi quebrado por Phong Nguyen e Jacques Stern.

O paradoxo é que *comprovado* não significa *verdadeiro:* um sistema comprovadamente seguro não é necessariamente verdadeiramente seguro, porque a prova se faz em um modelo formal que supõe

- certo funcionamento, atacante e objetivo de segurança e
- dificuldade do problema ao qual a prova é reduzido.

Por exemplo,

- O criptossistema implementado difere do criptossistemas formal.

- Um atacante pode ser mais forte do que aquele assumido.
- Um objetivo parcial já é suficiente para o atacante: Se, por exemplo, o objetivo de segurança é que o atacante não derive o inteiro texto claro do texto cifrado, então possivelmente já lhe baste derivar um trecho do texto claro.
- Improvável, mas teoricamente possível, é que o problema matemático subjacente é mais fácil do que suposto.

Além disto, a prova pode estar errada! Apesar desta incerteza, as provas de segurança são um critério (teoricamente necessário, mas praticamente insuficiente) útil para a segurança de um criptossistema.

Cenários de Ataques. O que exatamente significa segurança? A ideia que o atacante não pode derivar o texto original do texto cifrado é insuficiente, porque não significa que um atacante consiga outras informações relevantes sobre o conteúdo do texto claro (tais como trechos dele).

Mas até esta condição que não consiga informações sobre o texto claro é insuficiente nalgumas circunstâncias: Se

- o método de criptografia de chave pública for determinístico (ou seja, se a mesma entrada sempre retornar a mesma saída, como é o caso da criptografia RSA de livro didático, por exemplo) e
- o atacante pode limitar o número de possíveis textos claros (ele sabe, por exemplo que a mensagem cifrada é "sim" ou "não"),

então ele pode cifrar todos esses possíveis textos claros com a chave pública e comparar o texto cifrado com os seus textos cifrados obtidos.

Portanto, num método de criptografia assimétrica, sempre deve ser assumido que o atacante conheça a chave pública. Assim, ele pode cifrar qualquer texto claro de sua escolha e compará-lo com o texto cifrado para aprender algo sobre o texto claro desconhecido. Esse ataque é chamado de ataque de texto claro selecionado (Chosen Plaintext Attack = CPA).

Demarquemos a noção da segurança pela resistência contra vários cenários de ataques que ordenamos pelos recursos dos quais o atacante dispõe:

(Apenas) Texto Cifrado (ou CPO = ciphertext-only attack). O atacante tem o texto cifrado de várias mensagens, todas cifradas pelo mesmo algoritmo. A tarefa do atacante é encontrar tantas mensagens claras quanto possíveis, ou melhor ainda, encontrar a chave ou chaves que foram usadas, o que permitiria decifrar outras mensagens cifradas com essas mesmas chaves.

Texto Claro Provável. O atacante tem o texto cifrado e supõe que o texto claro contenha certas palavras ou até frases inteiras (o *crib*). Por exemplo, a Enigma, máquina criptográfica usada pelos eixos de potências na Segunda Guerra Mundial, foi quebrada pela repetitividade das palavras nas mensagens; muitas comunicaram, por exemplo, diariamente o boletim meteorológico e o anunciaram por tais palavras (Wetterbericht em alemão) no início de cada tal mensagem.

Texto Claro Conhecido (ou KPO = known plaintext attack). O atacante tem o texto cifrado e o texto claro correspondente. A tarefa é encontrar as chaves usadas ou um algoritmo que decifra outros textos cifrados pelas mesmas chaves. (A criptoanálise linear situa-se neste cenário.) Um exemplo recente é um ataque de 2006 ao protocolo Wired Equivalent Privacy (WEP) para cifrar uma rede local sem fio que explora a previsibilidade de partes das mensagens cifradas, os cabeçalhos do protocolo 802.11.

Texto Claro Joeirado ou Adaptativo (CPA = chosen plaintext attack). O atacante tem o texto cifrado e o texto claro correspondente, e pode escolher os textos claros; assim que o atacante possa deliberadamente variar, ou *adaptar*, o texto claro e analisar as alterações resultantes no texto cifrado. (A criptoanálise diferencial situa-se neste cenário.)

Este é o cenário mínimo para criptografia assimétrica! Como a chave de criptografia é pública, o atacante pode cifrar mensagens à vontade. Logo, se o atacante pode reduzir o número de textos claros possíveis, por exemplo, ele sabe que são ou "Sim" ou "Não", então ele pode cifrar todos os textos claros possíveis pela chave pública e compará-los com o texto cifrado interceptado. Por exemplo, o algoritmo modelar RSA (em Seção 8.1) sofre deste ataque, e nas suas implementações é preciso de preencher o texto claro por dados aleatórios para robustecê-lo contra este ataque CPA.

Texto Cifrado Seletivo (CCA = chosen ciphertext attack). No ataque de texto cifrado *selecionado* o atacante seleciona diferentes textos cifrados (exceto o texto cifrado a ser decifrado) que serão decifrados para ele derivar a chave. Por exemplo, o atacante dispõe de um dispositivo que não é desmontável e cifra automaticamente.

No ataque de texto cifrado *adaptativo* os textos cifrados (exceto o texto cifrado a ser decifrado) são *adaptados* dependendo do texto obtido após cada decifração. Poucos ataques práticos se situam neste cenário, mas ele é importante para provas de segurança: Se a resistência contra os ataques neste cenário é comprovada, então a resistência contra todo ataque realístico de texto cifrado selecionado pode ser admitida.

Segurança Semântica. O termo "segurança polinomial" apareceu em 1984 em Goldwasser e Micali (1984).

Segurança Semântica IND-CPA. Os autores, Goldwasser e Micali, demonstraram subsequentemente que a segurança semântica é equivalente a IND-CPA, indistinguibilidade (IND) do texto cifrado sob texto claro selecionado (= Chosen Plain-text Attack):

Em um método de criptografia assimétrica, o atacante conhece a chave pública. Assim, ele pode cifrar qualquer texto claro de sua escolha e compará-lo com o texto cifrado para aprender algo sobre o texto claro desconhecido. Esse ataque é chamado de ataque de texto claro selecionado (CPA = Chosen Plain-text Attack). Um método de criptografia é seguro contra IND-CPA (*indistinguibilidade do texto cifrado para textos claros selecionados*), se nenhum atacante consegue distinguir o qual entre dois textos claros, que ele selecionou antes, corresponde ao texto cifrado, que recebeu depois.

Passos. O criptossistema é **indistinguível sob ataque de texto claro selecionado** se todo adversário de tempo polinomial probabilístico tiver apenas uma "vantagem" insignificante sobre adivinhação aleatória:

O jogo IND-CPA em quatro passos com restrição de tempo de execução polinomial (no comprimento [em bites] da chave k) para os cálculos de atacante (ao criar os dois textos claros, no segundo passo, e ao escolher o texto claro que corresponde ao texto cifrado, no quarto passo).

1. É criada um par de chaves, uma secreta e outra pública, ambas com k bites. O atacante recebe a chave pública.
2. O atacante cria dois textos claros de tamanho igual M_0 e M_1.
3. A máquina criptográfica
    1. escolhe aleatoriamente um bit b em $\{0,1\}$
    2. cifra M_b, e
    3. passa o texto cifrado ao atacante.
4. O atacante escolhe um bite b' em $\{0,1\}$.

Um atacante que escolhe o bite b' na quarta etapa aleatoriamente está com probabilidade $1/2$ correto. Um método de criptografia é chamado seguro contra IND-CPA se nenhum atacante tiver uma probabilidade de sucesso significantemente maior que $1/2$. Isto é, se $\mathcal{P}(b = b') - 1/2$ é *insignificante*. O adversário tem uma "vantagem" *insignificante*, se vence com probabilidade $\geq 1/2 + \epsilon(k)$, onde ϵ é uma função *insignificante* em k, isto é, para toda função polinomial (diferente de zero) p existe k_0 tal que $\epsilon(k) < \frac{1}{p(k)}$ para todo $k > k_0$.

Uma diferença insignificante deve ser permitida porque é o atacante facilmente aumenta a sua probabilidade de sucesso acima de $1/2$ por adivinhar uma chave secreta e tentar decifrar o texto cifrado com ela.

Observação. Embora o jogo acima é formulado para um criptossistema assimétrico, pode ser adaptado ao caso simétrico, substituindo a cifração com a chave pública por um *oráculo criptográfico* (= uma função cujo funcionamento é totalmente desconhecido) que retém a chave secreta e cifra textos claros arbitrários a pedido do adversário.

Algoritmos Semanticamente Seguros. Algoritmos de criptografia semântica segura incluem El Gamal e Goldwasser-Micali porque a sua segurança semântica pode ser reduzida à solução dalgum problema matemático difícil, isto é, irresolúvel em tempo polinomial; nestes exemplos, o problema Diffie-Hellman decisório e o Problema do Resíduo Quadrático. Outros algoritmos semanticamente inseguros, como RSA, podem ser tornados semanticamente seguros por preenchimentos criptográficos aleatórios como o OAEP (= Optimal Asymmetric Encryption Padding).

Exemplo. O método de criptografia Elgamal é IND-CPA-seguro sob a hipótese de que o problema de *Diffie Hellman Decisório* seja difícil. Para provar a segurança, construamos a partir de um

- *vencedor* A do jogo IND-CPA para El Gamal (isto é, se lhe é dada a cifração de uma de duas mensagens claras sob a chave pública, então ele identifica com probabilidade $1/2 + \epsilon$ à qual (das duas mensagens claras) ela corresponde),
- um decididor S para DDH, isto é, dada uma base g e expoentes x, y e z em $\{0, \ldots, p-1\}$, decide em tempo polinomial se $g^z = g^{xy} \bmod p$ ou não,

como se segue:

1. S simula a geração das chaves e dá g^x como chave pública a A. Porém, S não sabe a chave secreta correspondente, sem A o saber.
2. A produz duas mensagens m_0 e m_1.
3. S simula a cifração: Ele escolhe aleatoriamente um bite b e define a cifração por $g^y, g^z m_b$.
4. A decide se esta cifração corresponde a m_0 ou a m_1.

- Se $g^z = g^{xy}$, então a cifração é indistinguível de uma cifração normal e A ganha com probabilidade $1/2 + \epsilon$.
- Se g^z é aleatório, então A só adivinha e ganha com probabilidade $1/2$.

A estratégia de S é, portanto, optar por $g^z = g^{xy}$ se, e tão-somente se, A está correto. Logo, a probabilidade que S esteja correto é $1/2 + \epsilon/2$.

Segurança Semântica IND-CCA.

Recordemo-nos de

- que um método de criptografia é seguro contra IND-CPA (*indistinguibilidade de texto cifrado para textos claros selecionados*), se nenhum atacante consegue distinguir o qual entre dois textos claros, que ele selecionou antes, corresponde ao texto cifrado, que recebeu depois, e
- que no ataque de texto cifrado *selecionado* (ou, mais exatamente, *adaptativo*) o atacante seleciona textos cifrados (exceto o texto cifrado a ser decifrado) que serão decifrados para ele derivar a chave.

Um método de criptografia é seguro contra IND-CCA (indistinguibilidade de texto cifrado para textos *cifrados* selecionados), se o atacante, no segundo e quarto passo do jogo de IND-CPA, pode mandar decifrar qualquer texto cifrado (exceto aquele a ser decifrado), e, mesmo assim, não consegue distinguir o qual entre dois textos claros corresponde ao texto cifrado:

1. O oráculo cria uma chave secreta.
2. O atacante manda decifrar quaisquer textos cifrados (exceto aquele a ser decifrado), calcula e produz dois textos claros de tamanho igual M_0 e M_1.
3. O oráculo
    - extrai um bit aleatório b em $\{0,1\}$
    - cifra M_b, e
    - passa o cifrado ao atacante.
4. O atacante manda decifrar quaisquer textos cifrados, calcula e escolhe um bite b' em $\{0,1\}$.

Exemplo. O ataque de Bleichenbacher de PKCS#1 de 1998 contra a variante de RSA seguro contra IND-CPA (mas, justamente, não contra IND-CCA).

Bellare e Namprempre mostraram em 2000 para uma cifra simétrica que se

- a cifra resiste contra IND-CPA, e
- a função unidirecional resiste (não é *forjável*) contra um ataque de mensagens selecionadas,

então a cifra com Encrypt-then-MAC resiste contra um ataque IND-CCA.

Maleabilidade. Um sistema criptográfico é *maleável* se é possível transformar *toda* mensagem cifrada M de uma mensagem m, sem nenhum conhecimento sobre m, em uma mensagem cifrada $N = f(M)$ de *alguma* mensagem n por uma função conhecida f tal que a relação entre m e n seja conhecida.

Exemplo. Todos os criptossistemas comuns são maleáveis; veremos, entre outros, que o criptossistema ElGamal é maleável.

Como consequência prática, uma cifra maleável não permita verificar a *autenticidade* de uma mensagem, isto é, se foi alterada entre envio e receção. Logo,

é mais suscetível a um ataque de Man-in-the-middle, um homem (no meio) entre dois correspondentes que interceta e forja as suas mensagens trocadas.

Se a maleabilidade é indesejada (para garantir a autenticidade da mensagem cifrada), então se anexa à mensagem um valor de uma função *unidirecional*, tal como um hash criptográfico, chamado de *Código de Autenticação de Mensagem* (MAC) da mensagem cifrada. (Chamado de Encrypt-then-MAC e usado, por exemplo, em VPNs, redes privadas virtuais. O hash *não* cifrado da mensagem clara, MAC-and-Encrypt é usado no protocolo SSH e o hash cifrado da mensagem clara no protocolo TLS que segura o protocolo HTTPS.) Embora um atacante possa alterar a mensagem cifrada M, não consegue alterar o seu MAC.

Observação. Existem cifras assimétricas não-maleáveis; a primeira tal cifra foi a de Cramer-Shoup.

Algoritmos Maleáveis.

- O one-time pad que

  – cifra o texto claro por uma adição XOR com a chave, e
  – decifra o texto cifrado da mesma maneira.

  é maleável porque um bite no texto cifrado é invertido se, e tão-somente se, o bit correspondente no texto claro é invertido.

  Se o atacante sabe decifrar um trecho do texto cifrado, então ele pode alterá-lo para corresponder a qualquer outro texto do mesmo comprimento. Isto é: Se m seja o texto claro, k a chave secreta e $\mathrm{E}(m) = m \oplus k$ o texto cifrado, então o texto cifrado de $m \oplus t$ para qualquer t é $\mathrm{E}(m) \oplus t = (m \oplus t) \oplus k = \mathrm{E}(m \oplus t)$.

- No sistema criptográfico RSA, um texto claro m é cifrado por $\mathrm{E}(m) = m^e \bmod n$ onde e é a chave pública módulo n. Se $\mathrm{M} = \mathrm{E}(m)$ é um texto cifrado de um texto claro m, então o texto cifrado de mt para qualquer t é

  $$\mathrm{M} \cdot t^e = \mathrm{E}(m) \cdot t^e \bmod n = (mt)^e \bmod n = \mathrm{E}(mt).$$

  Por isso, quando implementado, o texto claro é primeiro *preenchido aleatoriamente* por um protocolo como OAEP ou o PKCS1.

- No sistema criptográfico `ElGamal`, um texto claro m é cifrado pelo par $\mathrm{E}(m) = (g^b, m\mathrm{A}^b)$, onde $\mathrm{A} = g^a$ é a chave pública para a base g. Logo, se o par (g^b, c) é o texto cifrado de m, então, para qualquer t, o par $(g^b, t \cdot c)$, é o texto cifrado de tm.

CrypTool

É recomendado experimentar com o aplicativo `CrypTool` que

- apresenta mais de 300 algoritmos criptográficos tais como os históricos (simétricos)
  - de César
  - da cítala, e
  - da `Enigma`,

  mas também os modernos simétricos e assimétricos, como
  - `AES`, e
  - `RSA`

e

- dá ao usuário as ferramentas estatísticas para tentar decifrar textos exemplares.

Além disto, explica a matemática por trás destes algoritmos, por exemplo, o Algoritmo de Euclides, por animações.

Tem duas versões diferentes, `CrypTool 1` e `CrypTool 2`. Trabalhamos de preferência com o `CrypTool 1`.

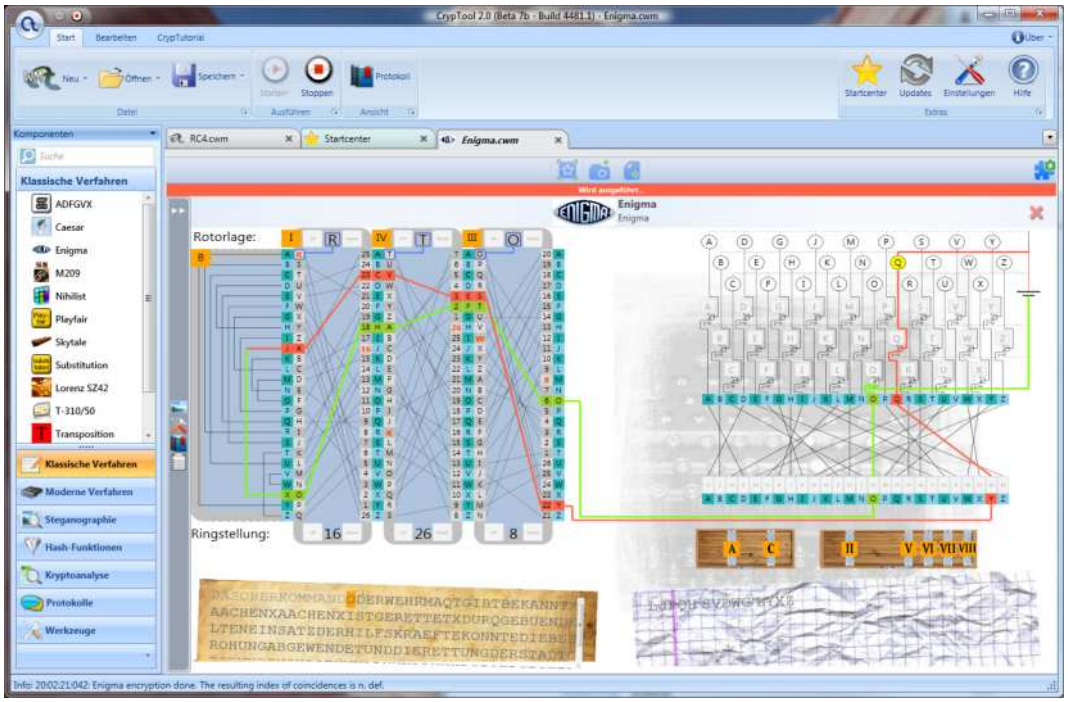


Figura 6: Cifrar com a Enigma no `CrypTool 2`

2 Criptografia Simétrica Antiga

Apresentamos uns exemplos antigos de algoritmos criptográficos sobre textos: Historicamente, a criptografia estuda a transformação de um

texto inteligível

$$\Downarrow$$

texto ininteligível

tal que só uma informação adicional secreta, a *chave*, permita desfazê-la.

Os algoritmos protótipos históricos são

- a substituição por César, que avança toda letra no texto claro pela mesma distância no alfabeto (a de Agosto era 1 enquanto a de Caesar 3), e
- a permutação do texto claro pelo bastão de Licurgo onde a fita é enrolada verticalmente mas o texto escrito sobre ela horizontalmente.

Veremos que mesmo com a presença de muitas chaves um algoritmo, como o dado pela permutação arbitraria do alfabeto que tem quase 2^{80} chaves, pode ser facilmente quebrado porque preserva regularidades como a da frequência das letras.

Como critério suficiente para uma boa segurança existe o da boa difusão por Shannon: Idealmente, se uma letra do texto claro muda, então a metade das letras do texto cifrado muda.

Veremos em Seção 3 como os algoritmos modernos, as redes da substituição e permutação, juntam e iteram os dois algoritmos protótipos complementares para atingir a meta da boa difusão por Shannon.

2.1 Substituição (= permutação das letras do Alfabeto)

Por traslado das letras do alfabeto. Este método foi usado por (Júlio) César (100 – 44 B.C.) e por Augusto (César) (63 – 14 B.C.): Fixa-se uma distancia d entre letras em ordem alfabética, isto é, um número entre 0 e 25, e traslada-se (para frente) cada letra do alfabeto (latino) por esta distância d.

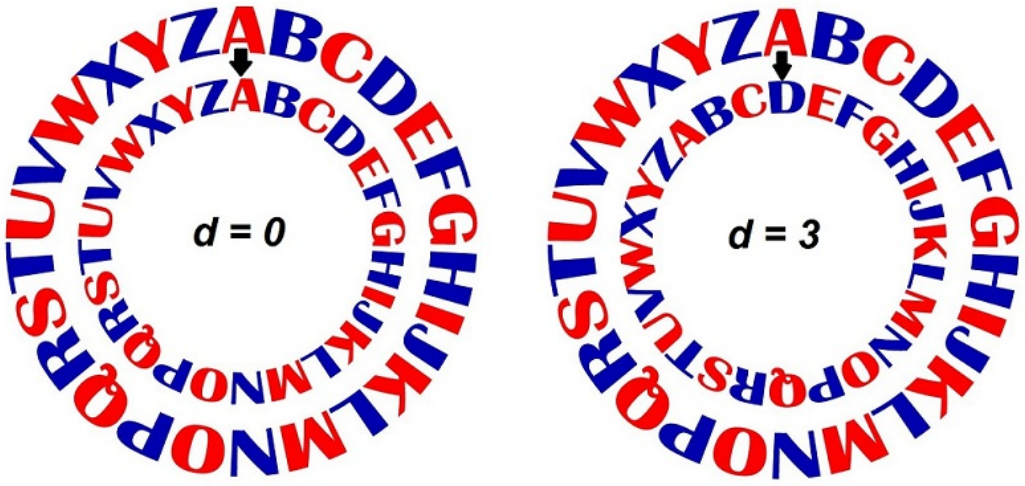


Figura 7: Imaginemos que as letras do alfabeto formem uma roda

Suponhamos que as letras sejam arranjadas em um anel; assim que o traslado de uma letra no fim do alfabeto resulte em uma letra no início do alfabeto.

Por exemplo, se $d = 3$, então

$$\mathrm{A} \mapsto \mathrm{D}, \mathrm{B} \mapsto \mathrm{E}, \ldots, \mathrm{W} \mapsto \mathrm{Z}, \mathrm{X} \mapsto \mathrm{A}, \ldots, \mathrm{Z} \mapsto \mathrm{C}.$$

Existem 26 chaves (incluindo a chave trivial $d = 0$).

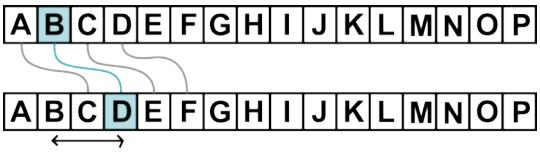


Figura 8: O César desloca cada letra do alfabeto pela a mesma distância

Para decifrar, traslada-se cada letra pela distância $-d$, isto é, d posições para trás. Se as letras do alfabeto forma uma roda, então as letras estão trasladadas

- na cifração no sentido horário, e
- na decifração no sentido anti-horário.

Pela ciclicidade (ou circularidade) da formação das letras, observamos que um traslado de d posições no sentido anti-horário iguala a um de $26 - d$ posições no sentido horário.

Questão: Para qual d a cifração é auto-inversa, isto é, a cifração iguala a decifração?

Por permutação das letras do alfabeto. Em vez de substituirmos cada letra por outra trasladada pela mesma distância fixada d, substituamos cada letra por outra letra arbitrária, por exemplo:

A	B	...	Y	Z
↓	↓	...	↓	↓
E	Z	...	G	A

Para podermos desfazer a cifração, é necessária que nunca duas letras sejam enviada a mesma letra! Isto é, permutamos as letras. Assim obtemos $26 \cdot 25 \cdots 1 = 26! > 10^{26}$ chaves ($\approx$ o número de senhas que podem ser formados com 80 bites).

2.2 Permutação (= permutação das letras do texto claro)

A *cítala* ou o *bastão de Licurgo* (= legislador de Esparta cerca de 800 AC) é um bastão com o qual os espartanos cifravam como segue:

1. enrolar o bastão com um pano estreito,
2. escrever neste pano horizontalmente, isto é, no sentido da borda maior, e
3. desenrolar o pano do bastão.

As letras assim transpostas no pano podiam unicamente ser decifradas por um bastão com a mesma **circunferência** (e suficientemente comprido) pela mesma maneira como o texto foi cifrado:

1. enrolar o bastão com o pano,
2. ler este pano horizontalmente, isto é, no sentido da borda maior.

Figura 9: A cítala enrolada com um cinto de couro

Aqui, a chave é o número dado pelo **número das letras que cabem nesta circunferência**.

Por exemplo, se o bastão tem uma circunferência de 2 letras (e um comprimento de 3 letras), as duas linhas

l	u	a
m	e	l

tornam-se as três linhas

l	m
u	e
a	l

que são concatenadas (para não revelarem nem a circunferência, nem o comprimento) à linha

L	M	U	E	A	L

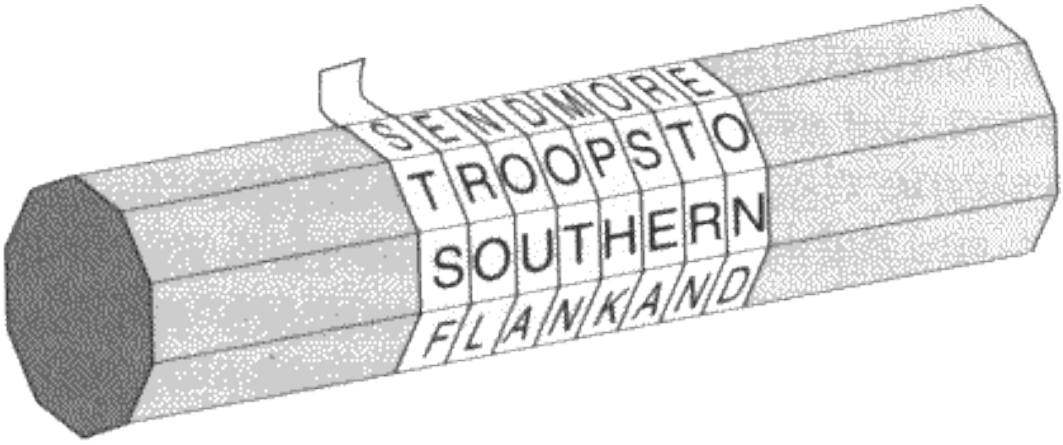


Figura 10: A cítala cifrando uma ordem militar em inglês.

2.3 Segurança dos Exemplos Históricos

Aplicamos os critérios para segurança em Seção 1.3 aos exemplos históricos.

Cifração de César. A *Substituição* (das letras do alfabeto por traslado) usada por César

$$A \mapsto D, B \mapsto E, C \mapsto F, \ldots, W \mapsto Z, X \mapsto A, \ldots, Z \mapsto C.$$

viola todas as qualidades desejáveis, principalmente o *princípio de Kerckhoff*, que o algoritmo seja público:

Uma vez o método for conhecido, considerando a pequena quantidade de *25* chaves, o texto cifrado cede em pouco tempo a um **ataque de força bruta**, uma busca exaustiva que prova cada chave.

Substituição por permutação arbitrária das letras do Alfabeto. A *Substituição* (das letras do alfabeto por permutação arbitrária)

A	B	…	Y	Z
↓	↓	…	↓	↓
E	Z	…	G	A

tem $26 \cdot 25 \cdots 1 = 26! > 10^{26}$ chaves, por isso um ataque de força bruta é computacionalmente inviável.

Mas ele viola as metas da *difusão e confusão*. Se a chave (= permutação do alfabeto) troca a letra α pela letra β, então há

- má *confusão* porque a substituição de β na chave implica unicamente a substituição de cada letra β no texto cifrado,
- má *difusão* porque a substituição de uma letra α no texto claro implica unicamente a substituição da letra correspondente β no texto cifrado.

Com efeito, o algoritmo permite ataques estatísticos sobre a frequência de

- letras,
- bigramas (= pares de letras) e

- trigramas (= triplos de letras).

em português. Por exemplo,

- a letra mais frequente em português é 'a'
- o bigrama mais frequente em português é 'de'
- o trigrama mais frequente em português é 'que'

$\Longrightarrow$ Substituindo

- a letra mais frequente do *texto cifrado* à letra mais frequente *em português* (= 'a'),
- o bigrama mais frequente do *texto cifrado* ao bigrama mais frequente *em português* (= 'de'), ...
- o trigrama mais frequente do *texto cifrado* ao bigrama mais frequente *em português* (= 'que'), ...

é um ponto de partida propício para decifrar o texto: Quanto mais texto cifrado, tanto mais provável que esta substituição coincide com a dada pela da chave para estas letras.

Por exemplo, o texto

GWQP VQX Q CQPQ XU GWQP G NQX VQX

considerando que as três letras mais comuns do português são, nesta ordem, AEO e que Q, G e X aparecem respectivamente 6, 4 e 4 vezes, obtemos

E*E* *AO A *A*A OU E*E* *AO *AO.

Suponhamos que $*$AO = VAO e que E$*$E$*$ = ELES. Em particular, V corresponde a V e P a S. Logo

ELES VAO A *ASA O* ELES *AO VAO

para estarmos levados ao chute

ELES VAO A CASA OU ELES NAO VAO.

Este exemplo é muito artificial pela sua brevidade que dificilmente permita formar uma frase que tenha no mesmo tempo um conteúdo típico e frequências de letras típicas. Como exercício, caro leitor, construa uma tal frase!

A cítala. A cítala viola

- o *princípio de Kerckhoff*, que o algoritmo seja público.

Com efeito, o valor máximo da circunferência é $< n/2$ onde n = o número das letras do texto cifrado. Por isso, um ataque de força bruta, é viável.

Ela tem

- má *difusão* porque a substituição de uma letra α no texto claro implica unicamente a substituição da mesma letra α no texto cifrado.

Com efeito, o algoritmo permite ataques estatísticos sobre a frequência de

- bigramas (= pares de letras)
- trigramas (= triplos de letras), e
- assim por diante.

Por exemplo, propício seria a escolha da circunferência como o número n que maximiza a frequência do bigrama 'de' entre as cadeias de letras nas posições $1, 1+n, 1+2n, \ldots$, $2, 2+n, 2+2n, \ldots$.

Por exemplo, se olhamos

ADABESACA.

observamos que d e e são distanciados por três letras, o que nos leva ao chute que

circunferência = 3 letras,

e à decifração

ABA DE CASA.

2.4 A Enigma

A enigma substitui o alfabeto, isto é, permuta as letras do alfabeta latim, mas cada letra por outro alfabeto; é uma *substituição poli-alfabética.*

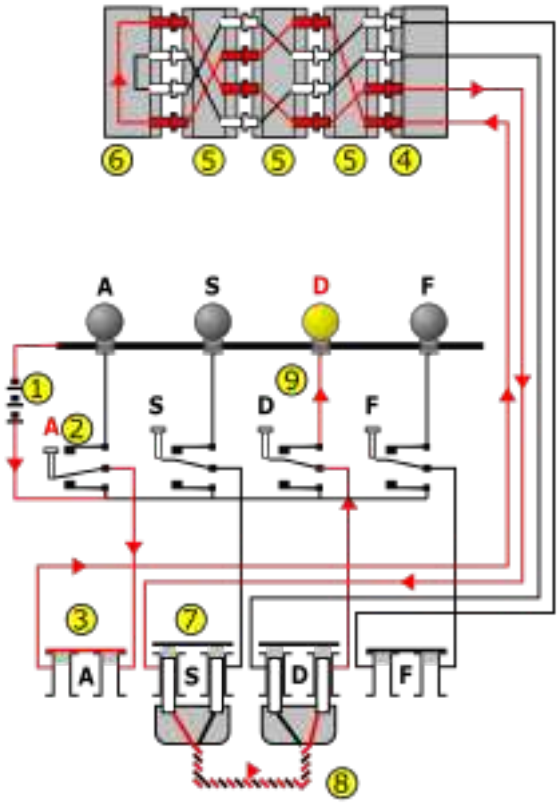


Figura 11: O esquema elétrico

Construção. O esquema elétrico, constituído por:

- Bateria (1),
- Teclado (2),
- O painel de ligações (3, 7) com cabo (8),
- o jogo dos cilindros (5) com o de entrada (4) e o de retorno (6), e
- o painel das lampadas (9)

Ao teclar uma letra, a corrente entra

1. pelo painel de ligações,
2. no cilindro de entrada,
3. no jogo dos cilindros,
4. no cilindro de retorno, e
5. percorre todo o caminho outra vez, no sentido inverso,

6. para finalmente ascender a lampada da letra cifrada.

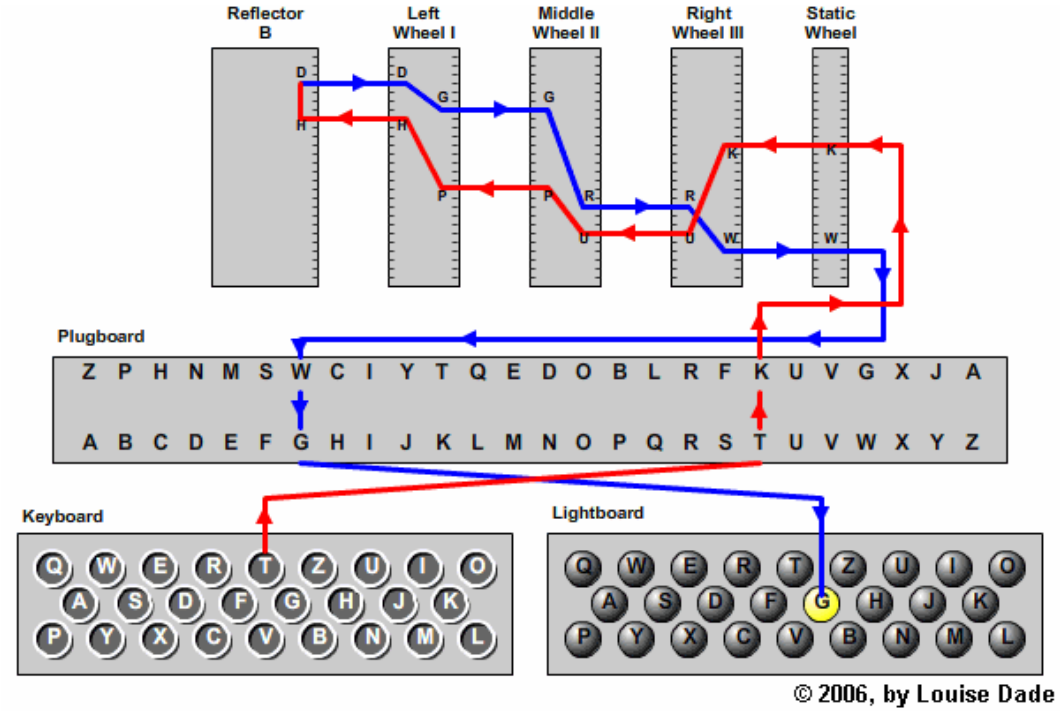


Figura 12: O percurso da corrente nos cabos da Enigma ao teclar uma letra

Cada cilindro consiste em

- um cabeamento interno,
- um anel
- um rotor.

Ao teclar uma letra, o rotor à direita (o rotor *rápido*) avança uma posição (antes da corrente entrar nos cilindros). Figura 13 mostra a substituição da letra `A`

- ao primeiro, e
- ao segundo teclar.

Em posições inicialmente determinadas pela posição do anel do rotor rápido e do rotor no meio, avançam também, uma posição,

- primeiro o rotor no meio (o rotor *médio*) e
- depois o à esquerda (o rotor *lento*)

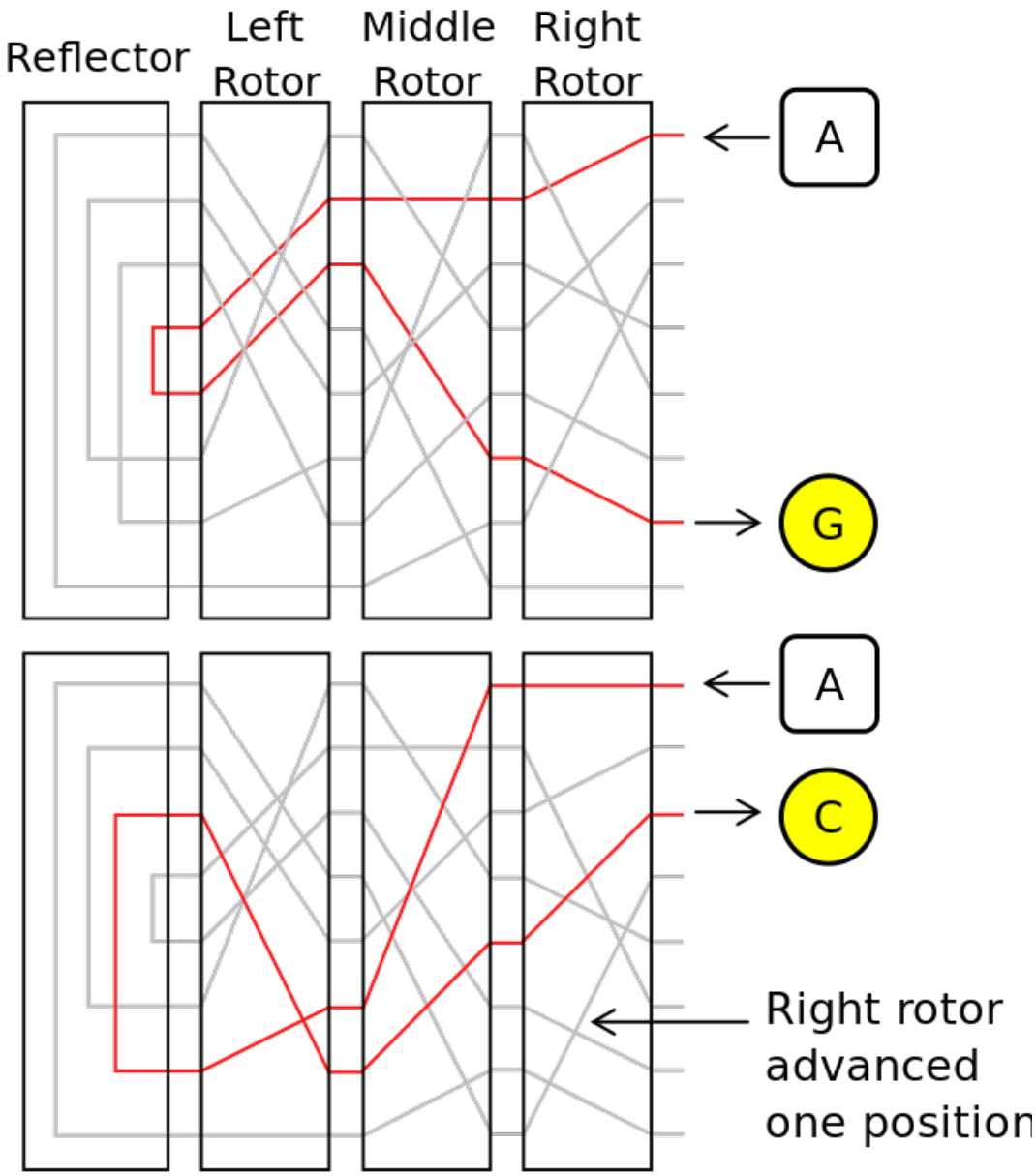


Figura 13: A Enigma em ação

Após este primeiro avanço (ou do rotor médio, ou do rotor lento),

- o rotor médio avança (uma posição) após uma rotação completa do rotor rápido, isto é, 26 avanços, e
- o rotor lento avança (uma posição) após uma rotação completa do rotor médio, isto é, 26 avanços.

Isto é, os rotores comportam-se como os de um taxímetro ou *conta-quilômetros*, com a diferença que

- os da Enigma têm 26 "algarismos" (correspondentes às letras do alfabeto) e
- os do velocímetro 10 algarismos

(além do detalhe que o rotor médio avança mais uma vez após o avanço do rotor lento). Esta simulação da Enigma online mostra a cifração da Engima em ação.

Cada *cilindro* permuta o alfabeto inteiro por um cabeamento interno. Este cabeamento é fixo, e não pode ser mudado pelo usuário. Por dentro, há uma verdadeira salada de cabos:

O anel tem o entalhe que desencadeia o avanço do rotor seguinte. Por isso, a posição do anel determina o ponto em que o *vai-um*, o avanço do rotor seguinte, tem lugar. Como o rotor lento não tem sucessor, de fato só as posições do rotor médio e rápido têm efeito criptográfico.

O *cilindro de retorno* garante que a substituição seja auto-inversa, isto é, a cifração iguale à decifração. Também, um defeito criptográfico, implicou que uma letra **nunca** foi cifrada **a si mesma** para evitar um curto-circuito!

O *cilindro de entrada* igualmente permuta o alfabeto, mas só existia na versão comercial da Enigma.

Na versão militar, não fazia nada, isto é, a substituição é a identidade. Isto foi intuído por *Marian Rejweski*, criptógrafo polonês (pelo vício dos alemães na ordem) antes da Segunda Guerra Mundial. Assim conseguiu inferir o cabeamento dos cilindros da Enigma à distância, enquanto os franceses e britânicos estavam perdidos.

O *painel de ligações* troca uns pares de letras, na prática, dez. Por exemplo, na foto, `A` e `J`, e `S` e `O`.

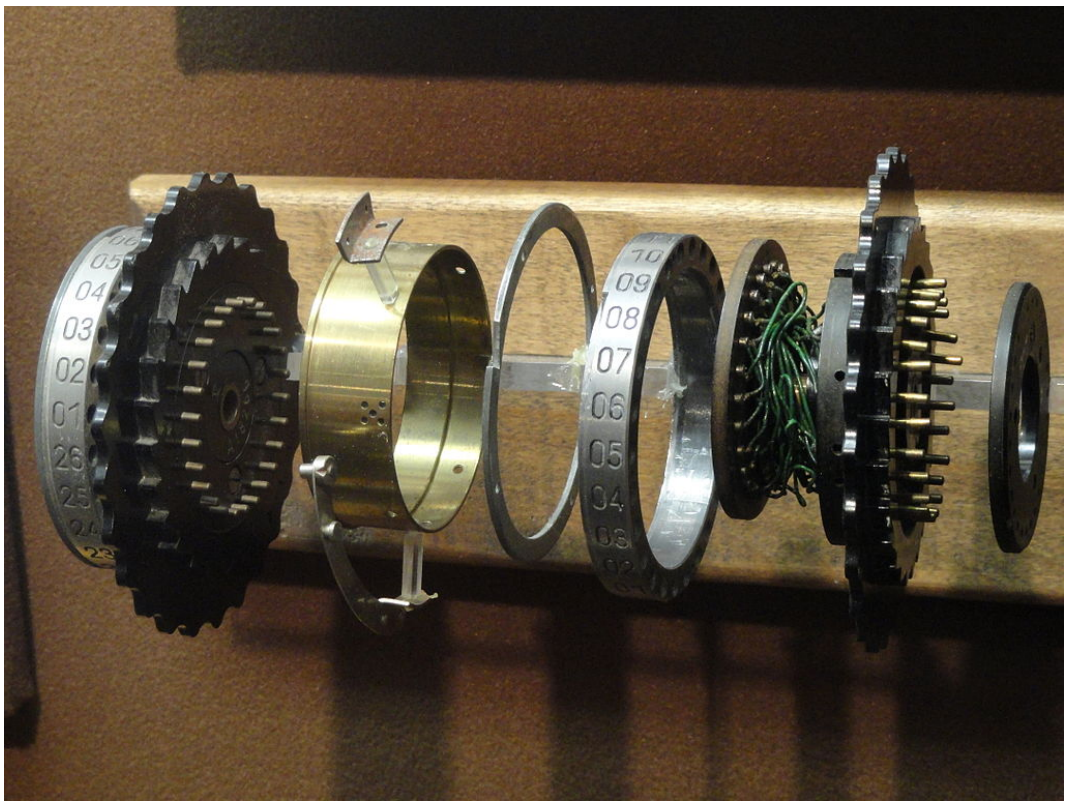


Figura 14: Os cilindros

Figura 15: Marian Rejweski

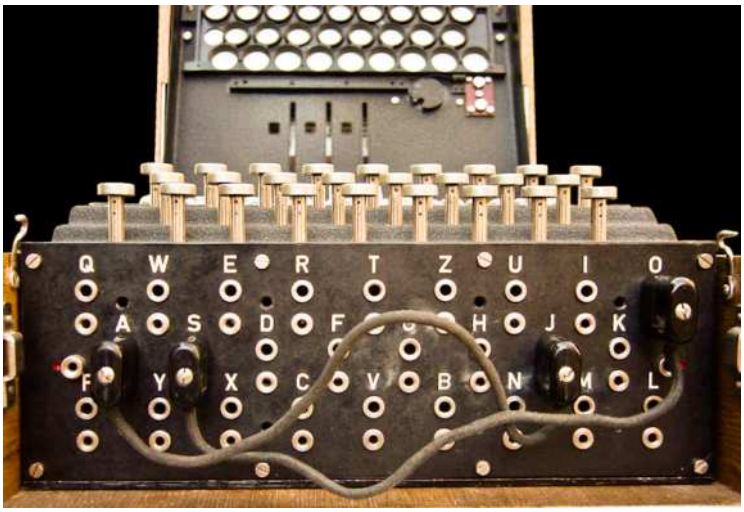


Figura 16: O painel de ligações

Expresso por uma fórmula, a substituição M (= Maquina) da Enigma descreve-se pela concatenação das substituições

$$M = P \circ E \circ C_R(c_R) \circ C_M(c_M) \circ C_L(c_L) \circ R \circ C_L(c_L)^{-1} \circ C_M(c_M)^{-1} \circ C_R(c_R)^{-1} \circ E^{-1} \circ P^{-1}$$

onde

- P é a substituição pelo Painel de ligações
- E é a substituição pelo cilindro de Entrada
- R é a substituição pelo cilindro de Retorno.
- $C_R(c_R)$, $C_M(c_M)$ e $C_L(c_L)$ são as substituições pelo Cilindro Rápido, Médio e Lento na posição c_R, c_M e c_L, onde

  $$C_{R/M/L}(c_{R/M/L}) = T^{c_{R/M/L}} C_{R/M/L} T^{-c_{R/M/L}}$$

  com

  - $c_{R/M/L} = r_{R/M/L} - a_{R/M/L}$ com
    * $r_{R/M/L}$ = posição do Rotor, e
    * $a_{R/M/L}$ = posição do Anel;
  - T = a Traslação = a substituição que traslada cada letra do alfabeto à sua vizinha, isto é $A \mapsto B$, $B \mapsto C$, …, $Y \mapsto Z$, $Z \mapsto A$, e
  - $C_{R/M/L}$ = a substituição do cilindro.

O CrypTool 2 inclui uma animação do funcionamento criptográfico da Enigma. (Mencionamos que o funcionamento do anel é errôneo, porque não afeta o momento do vai-um.)

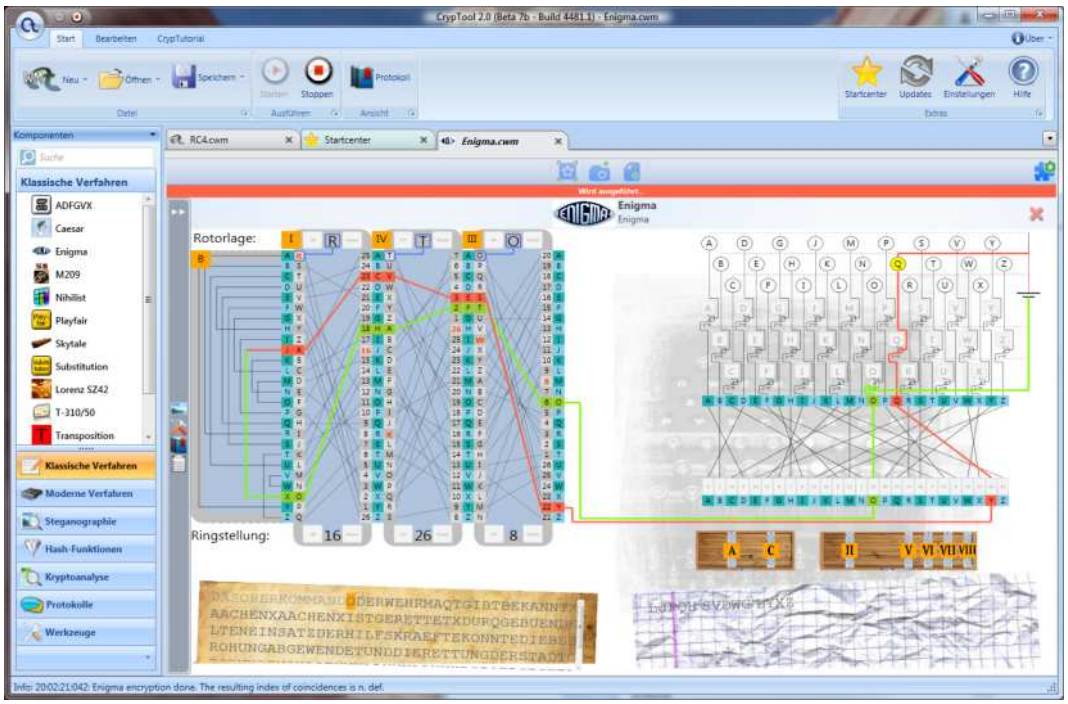


Figura 17: Cifrar com a Enigma no CrypTool 2

Ataque Estatístico pelas Frequências das Letras. Em particular, como a cada letra teclada (ao menos) o rotor rápido avança uma posição, isto é r_R é aumentado por 1, toda a substituição M muda. (E a cada rotação completa do rotor rápido, após 26 avanços, o rotor médio avança, e igualmente, a cada rotação completa do rotor médio, após 26 avanços, o rotor lento avança.)

Como a cada letra teclada o rotor rápido avança uma posição,

- isto é $r_R \rightsquigarrow r_R + 1$, e
- logo $c_R = r_R - a_R \rightsquigarrow c_R + 1$,
- segue que $C_R(c_R + 1) = T^{c_R} C_R(c_R) T^{-c_R}$ muda, e
- por isso toda a substituição M muda!

Por isso, as mesmas substituições só reaparecem após uma rotação completa de todos os cilindros, em particular, do cilindro lento; isto é, após cerca de 17.000 tecladas (a *periodicidade*). A Enigma é uma cifração por uma **substituição poli-alfabética**.

Recordemo-nos de que o ataque estatístico pelas frequências das letras

1. conta o número de cada letra no texto cifrado, e
2. associa a letra mais frequente no texto cifrado à letra mais frequente do idioma do texto claro, por exemplo, aqui, do alemão; em seguida, a segunda letra mais frequente no texto cifrado à segunda letra mais frequente do idioma do texto claro, e assim por diante.

Esta correspondência entre a frequência da letra cifrada e a da letra clara necessita que cada letra seja substituída pela *mesma* substituição!

Logo, o ataque estatístico pela frequência das letras (por exemplo, da língua alemã) para decifrar o texto,

- na *teoria*, aplica-se em textos cujo número de letras é um grande múltiplo desta periodicidade de 17.000 (porque, por exemplo, todas as letras nas posições 1, 17.001, 34.001 e assim por diante são substituídas pela *mesma* substituição); por exemplo, em um texto com 850.000 letras, um livro espessíssimo;
- na *prática*, não se aplica, porque, por ordem, as mensagens alemãs tinham no máximo 250 letras.

Chaves. Para a Engima I, há quatro fatores:

- Ordem e Escolha dos **Cilindros**: Foram escolhidos 3 (o lento, no meio e o rápido) entre 5 cilindros e em qualquer ordem, resultando em $5 \cdot 4 \cdot 3 = 60$ possibilidades.
- Posição do **Anel**: Determina a qual ponto o cilindro seguinte inicialmente avança uma posição, isto é, quando a rodada inicial se completa. Depois, uma rodada se completa a cada 26 letras. Há 26 posições (1 – 26) do anel para o rotor rápido e no meio (enquanto a posição do anel do rotor lento não importa, porque não implica um avanço de um cilindro em seguida), resultando em $26 \cdot 26 = 676$ possibilidades.

- Posição do **Rotor**: Determina em qual ponto a corrente inicialmente entra. Para cada um dos 3 rotores, há 26 possíveis posições (A – Z), resultando em $26 \cdot 26 \cdot 26 = 17.576$ posições. Por causa do mecanismo do escalonamento, $26 \cdot 26 = 676$ posições entre elas são criptograficamente redundantes, sobrando $17.576 - 676 = 16.900$ possibilidades.
- Conexões do **Painel**: Há até 13 cabos com dois conectores para conectar as 26 letras do alfabeto.

  1. Para o *primeiro* cabo, há 26 possibilidades para a letra do conector entrante e 25 possibilidades para a letra do conector sainte. Como não importa a ordem dos 2 conectores, sobram $[26 \cdot 25]/2$ possibilidades.
  2. Semelhantemente, para o *segundo* cabo, há 24 possibilidades para a letra do conector entrante e 23 possibilidades para a letra do conector sainte. Como não importa a ordem dos 2 conectores, sobram $[24 \cdot 23]/2$ possibilidades.
  3. $\cdots$

  Em geral, para o cabo n, há $n(n-1)/2$ possibilidades.

  Como a ordem em que os 13 pares de letras foram conectadas pelos cabos não importa, divide-se por $13 \cdot 12 \cdots 2 \cdot 1 = 13!$.

  Ao total, obtemos

  $$26\,25 \cdot 24\,23 \cdots 21/(2 \cdots 2) \cdot (13 \cdot 12 \cdots 1)$$

  possibilidades. Durante a guerra, a partir de Agosto 1939, foram conectados 10 pares de letras, dando

  $$26\,25 \cdot 24\,23 \cdots 8\,7/(2 \cdots 2) \cdot (10 \cdot 9 \cdots 1) = 150.738.274.937.250$$

  possibilidades.

Resumimos que há

- 60 possibilidades para os cilindros,
- 676 possibilidades para as posições dos anéis,
- 16.900 possibilidades para as posições dos cilindros, e
- 150.738.274.937.250 possibilidades para os pares de letras conectados pelos cabos,

o que dá ao total

$$60 \cdot 676 \cdot 16.900 \cdot 150.738.274.937.250 = 103.325.660.891.587.134.000.000 \sim 10^{23}$$

possibilidades, mais o menos o número de sequências de 80 bites. (Por exemplo, o DES (= Data Encryption Standard) utiliza uma chave de 56 bites.)

Na Segunda Guerra Mundial as mensagens enviadas pelos alemães tiveram, por ordem, no máximo 250 letras. Por isso:

- A posição do anel médio importava pouquíssimo, porque o anel médio só avança a um múltiplo (o qual em média valia 13) de 26. Isto é, este avanço raramente ocorre.
- A posição do anel no início importava mais, mas não tanto, porque em média o avanço ocorre após 13 tecladas, um número maior do que o número de letras da maioria das palavras alemães.

Logo, importam criptograficamente

- as 60 possibilidades para os cilindros (rápido, médio e lento),
- as 16.900 possibilidades para as posições do rotor rápido, médio e lento, e
- as 150.738.274.937.250 possibilidades para os pares de letras trocadas.

A *bomba de Turing* ajudou a reduzir estas possibilidades, sobretudo as do maior fator, as ligações do painel. Sobraram ainda

$$60 \cdot 16.900 = 1.014.000$$

possibilidades. Considerando a força de trabalho de 4200 pessoas (entre elas 80% mulheres) em Bletchley Park, o centro criptográfico inglês, neste ponto, um *ataque de força bruta* é viável, isto é, provando exaustivamente todas as possibilidades sobrantes.

A Bomba de Turing. Presumindo uma configuração do painel de ligações e um **crib**, uma palavra (alemã típica) que provavelmente ocorre no texto cifrado, a *bomba* de Turing (o nome deriva-se do som de uma bomba-relógio que a primeira tal máquina, a *bomba criptográfica polonesa*, emitia) elimina muitas posições de cilindros inválidas, da seguinte maneira:

1. Intui um **crib**, uma palavra provável, por exemplo,

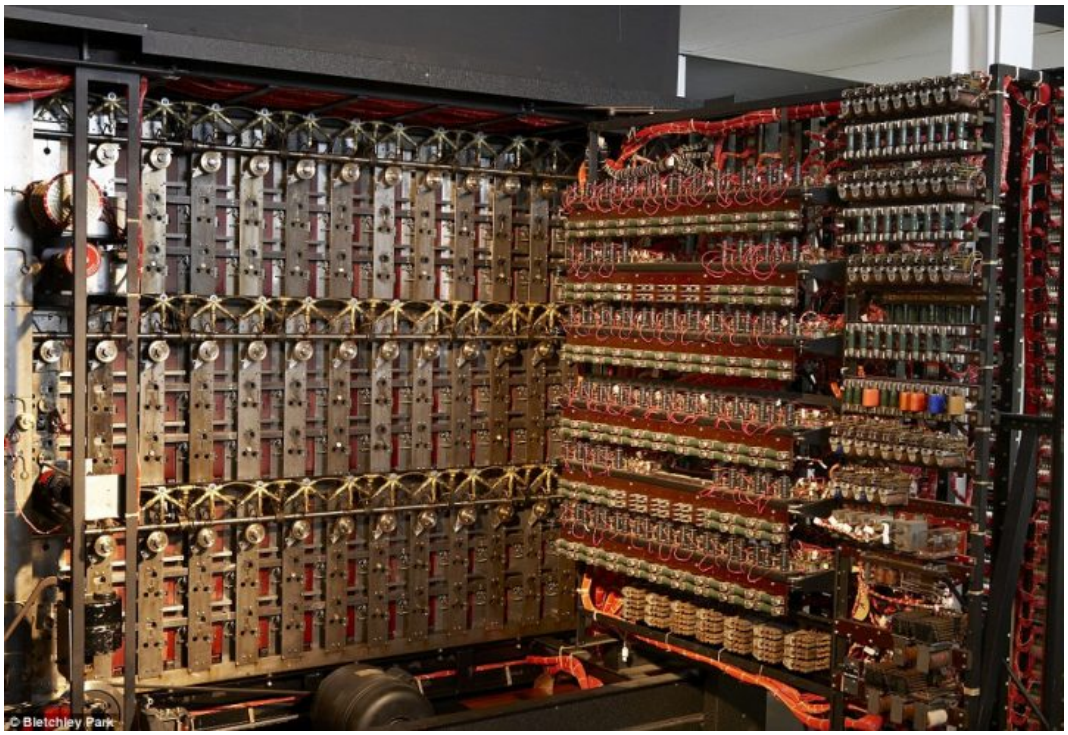


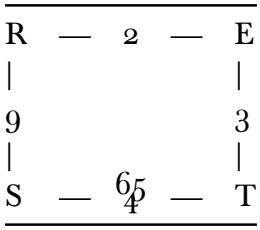
Figura 18: O interior da bomba de Turing

- `OBERKOMMANODERWEHRMACHT`, (= comando supremo do exercito alemão)
- `WETTERBERICHT` (= previsão do tempo)
- `EINS` (= o número um)

e compara com um trecho do texto cifrado. Para encontrar o trecho certo, toma em conta que a Enigma *nunca* cifrava uma letra a si mesma!

1	2	3	4	5	6	7	8	9	10	11	12	13
W	E	T	T	E	R	B	E	R	I	C	H	T
A	R	E	S	T	U	W	L	S	K	H	I	O

2. Cria o **menu**, um diagrama do percurso de uma letra dado pelas substituições entre o texto claro e o texto cifrado; aqui obtemos o circuíto



Isto é, são trocadas as letras

- R e E na segunda posição,
- E e T na terceira posição,
- T e S na quarta posição, e
- S e R na nona posição.

provou cada posição de cilindros para a sua compatibilidade com os circuitos do menu, da seguinte maneira:

Recordemo-nos de que a substituição da uma letra pela Enigma se descreve pela concatenação M das substituições seguintes

$$M = P \circ E \circ C_R \circ C_M \circ C_L \circ R \circ C_L^{-1} \circ C_M^{-1} \circ C_R^{-1} \circ E^{-1} \circ P$$

onde

- P é a substituição pelo painel de ligações
- E é a substituição pelo cilindro de entrada
- R é a substituição pelo cilindro de retorno, e
- C_R, C_M e C_L são as substituições pelo cilindro rápido, médio e lento.

Para facilitar,

- tomamos em conta que E = id é a identidade, e
- abreviamos

$$C := C_R \circ C_M \circ C_L \circ R \circ C_L^{-1} \circ C_M^{-1} \circ C_R^{-1}$$

Isto é,

$$M = P \circ C \circ P$$

Para destacar a dependência da substituição C da posição p da letra no texto a ser cifrado (pelo avanço do cilindro rápido a cada letra teclada), denote

$$C(p) = \text{ a substituição pelos cilindros na posição } p \text{ do texto.}$$

(Conforme à notação anterior e supondo que o cilindro no meio não avance, ela deveria ser $C(\mathbf{c} + (0,0,p))$ onde $\mathbf{c} = (c_L, c_M, c_R)$ é a configuração inicial do cilindro lento, médio e rápido.) Aqui para $p = 2$, obtemos

$$\mathbf{E} = P(C(2)(P(\mathbf{R}))).$$

Como o painel de ligação troca as letras em pares, a substituição pelo painel de ligação é auto-inversa, isto é, P ∘ P = identidade. Por isso, ao aplicarmos P a ambos lados desta equação,

$$P(\mathbf{E}) = C(2)(P(\mathbf{R}));$$

Em seguida, da mesma maneira,

$$\mathrm{P}(\mathbf{T}) = \mathrm{C}(3)(\mathrm{P}(\mathbf{E})) = \mathrm{C}(3)\mathrm{C}(2)(\mathrm{P}(\mathbf{R}));$$

e assim por diante para as outras letras no circuito, até

$$\mathrm{P}(\mathbf{R}) = \mathrm{C}(9) \circ \mathrm{C}(4) \circ \mathrm{C}(3) \circ \mathrm{C}(2)(\mathrm{P}(\mathbf{R})),$$

fechando o circuito.

Isto é, sob esta configuração dos cilindros, obtemos que a substituição $\mathrm{C} = \mathrm{C}(9)\mathrm{C}(4)\mathrm{C}(3)\mathrm{C}(2)$ deixa a letra $\mathrm{P}(\mathbf{R})$ invariante,

$$\mathrm{C}(\mathrm{P}(\mathbf{R})) = \mathrm{P}(\mathbf{R}).$$

Concluímos que cada tal circuito (obtido pelo texto claro e cifrado) exclui muitas configurações. A Bomba calculou quais.

Alan Turing calculou quantas configurações de posições de cilindros são em média compatíveis para um *Crib* com dado número de circuitos e letras :

# circuitos \ # letras	8	9	10	11	12	13	14	15
3	2.2	1.1	0.42	0.14	0.04	<0.01	<0.01	< 0.01
2	58	28	11	3.8	1.2	0.30	0.06	< 0.01
1	1500	720	280	100	31	7.7	1.6	0.28

As equações acima implicam também, a partir de

- uma configuração válida e
- a troca de uma letra do circuito por um conector do painel de ligações,

todas as outras trocas das letras no circuito pelo painel de ligações:

Por exemplo, dado $\mathrm{P}(\mathbf{R})$, o valor $\mathrm{P}(\mathbf{E})$ define-se por

$$\mathrm{P}(\mathbf{E}) = \mathrm{C}(2)(\mathrm{P}(\mathbf{R}));$$

em seguida,

$$\mathrm{P}(\mathbf{T}) = \mathrm{C}(9)(\mathrm{P}(\mathbf{E})) = \mathrm{C}(9)\mathrm{C}(2)(\mathrm{P}(\mathbf{R}));$$

e assim por diante, obtendo as trocas de todas as letras do circuito: **R**, **E**, **T** e **S**.

Como a validade da chave era um dia, e a mesma entre todas as naves (e entre todas as aeronaves e entre todos os trens), logo que uma chave foi obtida, todas as mensagens entre todas as naves durante este dia podiam ser decifradas com ela. Mudavam as posições dos rotores e conectores do painel cada dia, e a escolha e ordem dos cilindros cada mês.

Destacamos a importância do *crib*, da palavra alemã típica, neste método. Os aliados até provocaram pelas suas manobras certas mensagens para aplicar este método. Caso contrário, não conseguiram por exemplo decifrar a comunicação entre os condutores de trem por desconhecimento do jargão (= gírias profissionais) entre eles.

3 Criptografia Simétrica Moderna ou Cifras de Feistel

Hoje em dia, a criptografia estuda a transformação de

dados inteligíveis

$$\Downarrow$$

dados ininteligíveis

tal que só uma informação adicional secreta, a **chave**, permite desfazê-la.

dados	=	arquivo digital (de texto, imagem, som, vídeo, ...)
	=	sequência de bites (= 0, 1)
	=	sequência de baites (= 00, 01, ..., FE, FF)
	=	número (= 0, 1, 2, 3 ...)

3.1 One-time Pad

O `One-time pad` (= Bloco de Uso Único) adiciona (pela operação `XOR`, isto é, ou exclusivo, a disjunção exclusiva) cada baite do texto claro t com o baite (na posição) correspondente de uma chave c que

- é do mesmo comprimento, e
- é descartada após uso, isto é, não será usada para cifra outros textos claros.

Isto é, o texto cifrado $T = (T_1, T_2, ...)$ é

$$T = t \oplus c = (t_1 \oplus c_1, t_2 \oplus c_2, ...).$$

Este método de cifração é tão seguro quanto teoricamente possível!

Se o texto claro tem um único bloco t, então a simples adição (`XOR`) de uma chave, o `one-time pad`, é um algoritmo seguro. Porém, é frequentemente inconveniente ou até praticamente impossível ter uma chave tão grande quanto o texto claro: Por exemplo,

- para cifrar um disco rígido, precisa outro do mesmo tamanho para guardar a chave, e

- para cifrar a comunicação em uma rede (por exemplo, sem fio), precisaria saber já *antes* quanto texto será transferido.

Na prática, imagine-se um agente duplicar gigabaites de ruído em dois meios de armazenamento, por exemplo, um disco rígido e pendrive, e levar um destes meios para cifrar a sua comunicação pelo `one-time pad`.

Infelizmente, é uma péssima ideia (apesar de ser tão natural) de usar a mesma chave para dois blocos diferentes: se, por exemplo, o texto claro tem *dois* blocos b' e b'', então, com este algoritmo, a soma (`XOR`)

$$(b' \oplus c) \oplus (b' \oplus c) = b' \oplus b''$$

dos dois blocos cifrados $b' \oplus c$ e $b'' \oplus c$ iguala a soma $b' \oplus b''$ dos dois blocos claros (porque a adição `XOR` é por definição *auto-inversa*, isto é $x \oplus x = 0$ independente do dígito binário ser $x = 0$ ou $x = 1$)!

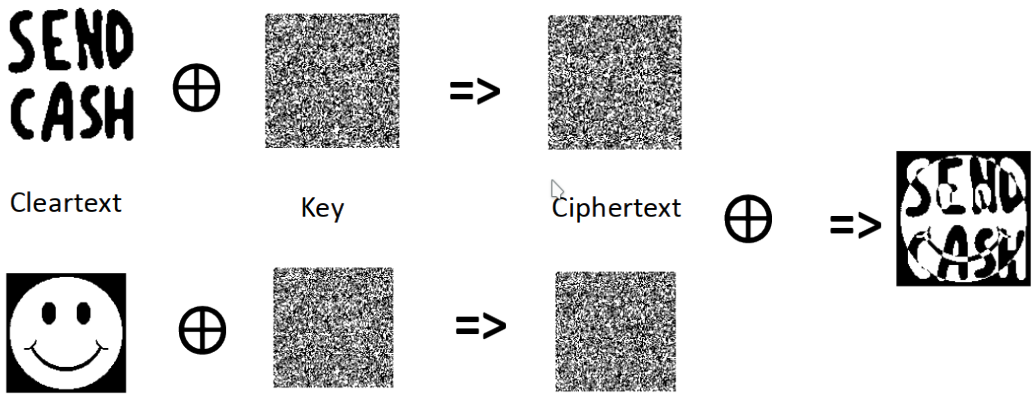


Figura 19: A soma de dois textos claros revela regularidades!

Pode ser visto como a cifração do primeiro bloco por `one-time pad` cuja chave é o segundo bloco. Infelizmente, o segundo bloco não é uma boa chave, porque longe de ser aleatório; ao contrário, normalmente o seu conteúdo semelha ao do primeiro bloco, isto é, a chave é previsível.

3.2 Cifras de Feistel

Por isso, é mais seguro aplicar uma substituição do bloco por outro segunda a escolha da chave.

Porém, o alfabeto desta substituição seria gigantesco, e, por isso, este ideal é praticamente inatingível, sobretudo sobre um hardware tão limitado quanto o de um cartão inteligente com o seu processador de 8 bites.

Para um bloco de por exemplo, 16 baites, esta tabela de substituição teria horrendos $2^{256} \cdot 16$ baites. Por isso, por exemplo, o AES adiciona a chave; depois substitui só cada baite, cada casa do bloco, uma tabela de substituição de $2^8 = 256$ entradas de 1 baite; em seguida, permuta as casas. Veremos que estas operações se complementam tão bem que são quase tão seguros como uma substituição do bloco inteiro. Isto é, em compensa da ausência de uma substituição do bloco inteiro por outro, a permutação imita esta substituição gigantesca da melhor maneira.

Uma *Cifra de Feistel* ou uma *rede de substituição e permutação* (abreviada SPN para Substitution Permutation Network) agrupa o texto (= sequência de baites) em blocos de n baites (por exemplo, $n = 16$ para AES e $n = 2$ no nosso modelo prototípico) e cifra cada bloco pela iteração (por exemplo, 10 vezes no AES, e 5 vezes no nosso modelo prototípico) dos três seguintes passos, em dada ordem:

1. adição (XOR) da chave,
2. substituição do alfabeto (que opera em cada sub-bloco do bloco), e
3. permutação (entre todos os sub-blocos do bloco).

Isto é, após

1. a adição (por Ou Exclusivo) da chave como no One-time pad,

são aplicadas

2. a substituição do alfabeto (por exemplo,
    - no algoritmo de Heys, cada letra hexadecimal por outra, ou
    - no algoritmo AES cada baite, par de letras hexadecimais, por outro),
    e
3. a permutação do texto (do passo atual, chamado de *estado*), por exemplo, por exemplo, no AES, que agrupa o texto em um quadrado de baites (= pares de letras hexadecimais), são permutas as casas nas linhas (e colunas).

Estas duas simples operações,

- a substituição do alfabeto, e
- a permutação do texto

complementam-se muito bem, isto é, geram alta confusão e difusão após de poucas iterações.

Na primeira e última rodada, os passos antes respetivamente depois da adição da chave são omitidos porque não aumentam a segurança criptográfica: Como o algoritmo é público (segundo o critério de Kerckhoff), qualquer atacante pode desfazer os passos que não necessitam o conhecimento da chave.

3.3 Modelo Prototípico por Heys

Usamos o algoritmo de Heys (2002) como modelo simples de uma cifra de Feistel que

- divide o texto claro em blocos de 16 bites, e
- subdivide cada bloco em 4 blocos de 4 bites

e em cada uma das primeiras rodadas 1, 2 e 3,

1. Adiciona a chave da rodada (para cada rodada, tem uma chave correspondente independente) ao bloco, $B \mapsto B \oplus C$.
2. Substitui cada um dos 4 sub-blocos de 4 bites pela tabela

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
E	4	D	1	2	F	B	8	3	A	6	C	5	9	0	7

3. Troca o bite i do sub-bloco j com o bite j do sub-bloco i;

Na penúltima 4ª rodada

1. Adiciona a chave da rodada ao bloco, $B \mapsto B \oplus C$.
2. Substitui cada um dos 4 sub-blocos de 4 bites pela tabela.

Na última 5ª rodada

1. Adiciona a chave da rodada ao bloco, $B \mapsto B \oplus C$.

Isto é:

- Na 5^{a} rodada, os passos seguintes, a substituição e a permutação, são omitidos, pois o algoritmo sendo público (pelo princípio de Kerckhoff) podem ser desfeitos por qualquer decifrador, sem ele conhecer a chave. Isto é, de um ponto de vista criptográfico, são supérfluos.
- Na 4^{a} rodada, o último passo, a permutação, é omitido, pois ele só permutaria a última 5^{a} chave. Isto é, de um ponto de vista criptográfico, é supérfluo.

A tabela origina do algoritmo DES e e comumente chamada de *S-box*, caixa de substituição.

3.4 Criptoanálise Diferencial

Para compreender as razões por trás das escolhas de cada passo de um algoritmo criptográfico simétrico moderno que cifra em blocos, tal como o AES, precisa-se de entender contra quais ataques se robustece. Um dos mais conhecidos ataques é o da criptoanálise *diferencial* que vamos explicar agora pelo exemplo da cifra de bloco prototípica de Heys:

O sonho de todo decifrador é poder aprender se uma parte da chave escolhida é correta, isto é, se coincide com a parte correspondente da chave usada para cifrar o texto: Por exemplo, no algoritmo de Heys a chave consiste de 16 bites: Se for possível saber, para 8 bites da chave provada, se eles coincidem com os bites correspondentes da chave correta, então o decifrador

- ao invés de provar, por um ataque de força bruta, todas as combinações possíveis, de que existem $2^{16} = 65536$,
- apenas, todas as combinações possíveis destes 8 bites (de que existem $2^{8} = 256$) e dos 8 restantes 16 (de que existem $2^{8} = 256$).

Isto é, o número de combinações que precisam ser provadas foi reduzido de $2^{16} = 65536$ a $2 \cdot 256 = 512$.

Mais exatamente, a chave consiste de 4 blocos de 4 bites. Daremos um exemplo da criptoanálise abaixo em que o decifrador terá um critério probabilista para decidir se 2 dos blocos, isto é, 8 bites, da chave provada são corretos.

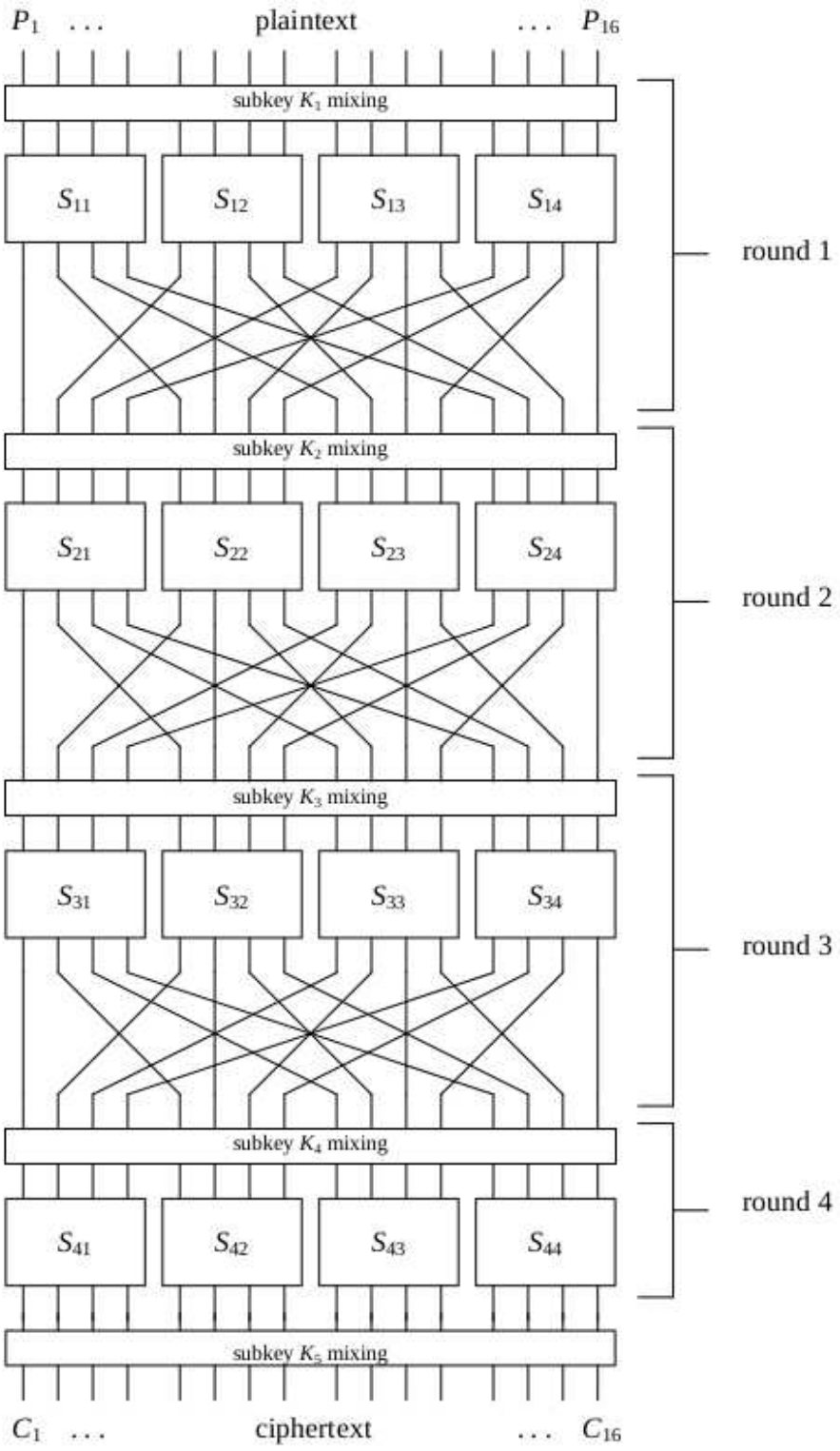


Figura 20: Diagrama das rodadas da cifra de Feistel dada em Heys (2002)

Critério de Decifração. No ataque de força bruta, o decifrador, para cada chave possível, decifra o texto cifrado com ela. Para saber se a chave é *correta*, isto é, se coincide com a chave usada para cifrar o texto, o decifrador verifica se o conteúdo seja *inteligível*; por exemplo, pelo critério de

1. contar as frequências das letras, pares e triples do texto decifrado, e
2. compará-las às frequências do idioma provável em que o texto claro foi escrito: Se elas se aproximam, então, provavelmente, o texto claro é inteligível e a chave usada pelo decifrador é a usada pelo cifrador.

Se a cifra tem uma única rodada, então este critério se aplica. Porém, se a cifra tem duas ou mais rodadas, e o decifrador executa a última rodada do algoritmo da decifração com certa chave, então este critério não presta mais porque o texto obtido é a saída do algoritmo da cifração (com a mesma chave) da penúltima rodada (logo, de qualquer forma, ininteligível). Ao invés dele, o critério da criptoanálise diferencial para ter encontrado a chave correta é probabilista: a chave provada é provavelmente correta se, para uma determinada diferença "entrante" ΔX e uma determinada diferença "sainte" ΔY, pares de textos claros com diferença ΔX resultam na penúltima rodada com determinada probabilidade em pares de textos cifrados com diferença ΔY.

Para a criptoanálise diferencial ser aplicável, o decifrador precisa de poder cifrar pelo cifrador (pela mesma chave)

- um número arbitrário de textos claros
- com conteúdos arbitrários, e

em seguida,

- examina os textos cifrados.

A criptoanálise diferencial explora a alta probabilidade da propagação de uma *diferença* $\Delta X = X' \oplus X''$ entre dois textos claros X' e X'' a uma diferença $\Delta Y = Y' \oplus Y''$ entre os dois textos cifrados Y' e Y'' de X' e X'' na penúltima rodada. (Recordemo-nos de que $X' \oplus X''$ é a adição `XOR` bite a bite, em que a saída é 1 se, e tão-somente se, as duas entradas são diferentes. Isto é, ΔX, indica todos os bites em que X' e X'' diferem.)

Chamemos este par $D = (\Delta X, \Delta Y)$ o *diferencial*. Para a criptoanálise diferencial ser eficiente, é preciso existir um diferencial D com *alta probabilidade* p_D (onde *alto* é quantificado em Equação 1); isto é, entre todos os pares entrantes com a

diferença ΔX, a probabilidade de um par sainte ter diferença ΔY (na penúltima rodada) é p_D. Mais exatamente, o cifrador cifrará um número estatisticamente significante de pares ($> 1/p_D$) de textos claros com diferença ΔX para contar o número de pares dos textos cifrados com diferença ΔY.

Frequência das Diferenças para uma Tabela de Substituição. Observemos que para uma aplicação A *afim* (isto é, dada por um polinômio de grau 1, ou, equivalentemente, a composição

- de uma aplicação *linear*, isto é, $A(x \oplus y) = A(x) \oplus A(y)$ para todo x e y, e
- de uma *traslação*, isto é, $A(x) = x \oplus x_0$ para algum x_0 fixo),

o diferencial $(\Delta X, \Delta Y)$ independe do par X' e X'' entrante: Se a aplicação A

- é linear, então sempre, isto é, para quaisquer par com diferença ΔX o resultado é $\Delta Y = A(\Delta X)$, e se
- é uma traslação, isto é, $A = \cdot \oplus a_0$ para algum a_0 fixo, então sempre $\Delta Y = \Delta X$.

Aplicada esta observação à uma cifre de Feistel, vemos que

- a permutação é uma aplicação linear,
- a adição da chave é uma traslação,

isto é, a diferencial independe do par X' e X'' entrante. Porém, a diferença sainte ΔY da substituição *não* é determinada apenas pela diferença ΔX, mas ela *depende* de X' e X''!

Logo, para encontrar tal diferencial D com uma alta probabilidade p_D, precisamos de investigar apenas a tabela de substituição. Estamos interessados na frequência de uma diferença sainte ΔY dada uma diferença entrante ΔX: Dada ΔX, há $2^4 = 16$ possibilidades para X' (e o qual determina $X'' = X' \oplus \Delta X$), e contamos as frequências das $2^4 = 16$ possíveis saídas $\Delta Y = 0, 1, \ldots, F$.

Esta tabela alista, para umas diferenças entrantes e todos os pares entrantes com estas diferenças, as suas diferenças saintes.

Tabela 12: as diferenças saintes ΔY para três diferenças entrantes ΔX (alistadas horizontalmente) e todas as entradas X (alistadas verticalmente).

X \ ΔX	1011	1000	0100
0000	0010	1101	1100
0001	0010	1110	1011
0010	0111	0101	0110
0011	0010	1011	1001
0100	0101	0111	1100
0101	1111	0110	1011
0110	0010	1011	0110
0111	1101	1111	1001
1000	0010	1101	0110
1001	0111	1110	0011
1010	0010	0101	0110
1011	0010	1011	1011
1100	1101	0111	0110
1101	0010	0110	0011
1110	1111	1011	0110
1111	0101	1111	1011

Contemos, para toda diferença entrante ΔX, quantas vezes cada diferença sainte ΔY resulta entre todos os possíveis pares entrantes X′ e X″ com $X' \oplus X'' = \Delta X$.

Tabela 13: A frequência das diferenças saintes ΔY (alistadas horizontalmente) para toda as diferenças entrantes ΔX (alistadas verticalmente).

ΔX $backslash ΔY	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	0	0	0	2	0	2	4	0	4	2	0	0
2	0	0	0	2	0	6	2	2	0	2	0	0	0	0	2	0
3	0	0	2	0	2	0	0	0	0	4	2	0	2	0	0	4
4	0	0	0	2	0	0	6	0	0	2	0	4	2	0	0	0
5	0	4	0	0	0	2	2	0	0	0	4	0	2	0	0	2
6	0	0	0	4	0	4	0	0	0	0	0	0	2	2	2	2
7	0	0	2	2	2	0	2	0	0	2	2	0	0	0	0	4
8	0	0	0	0	0	0	2	2	0	0	0	4	0	4	2	2

ΔX $backslash ΔY	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
9	0	2	0	0	2	0	0	4	2	0	2	2	2	0	0	0
A	0	2	2	0	0	0	0	0	6	0	0	2	0	0	4	0
B	0	0	8	0	0	2	0	2	0	0	0	0	0	2	0	2
C	0	2	0	0	2	2	2	0	0	0	0	2	0	6	0	0
D	0	4	0	0	0	0	0	4	2	0	2	0	2	0	2	0
E	0	0	2	4	2	0	0	0	6	0	0	0	0	0	2	0
F	0	2	0	0	6	0	0	0	0	4	0	2	0	0	2	0

As entradas de cada linha somam-se a 16, o número dos possíveis pares para uma dada diferença entre eles. A primeira linha constata que duas entradas iguais resultam em duas saídas iguais. O número maior é 8 e atingido para $\Delta X = B$ e $\Delta Y = 2$. Além dele, surge cinco vezes o número 6.

Exemplo. A tabela de frequência

- para uma traslação, como a adição da chave, todas as casas são nulas exceto as da primeira coluna que têm o valor 16;
- para uma operação linear, como a permutação dos bits, em cada linha todas as casas são nulas exceto uma que tem o valor 16.

Escolheremos os nossos diferenciais entre os com estas altas frequências:

Trilhas Diferenciais. Para uma cifra de Feistel, uma *trilha diferencial* é uma sequência finita de diferenças

$$(\Delta U_1, \Delta U_2, \ldots)$$

tal que toda diferença ΔU_i é a entrada da `S-box` da rodada i da cifra. Dada a saída ΔV_i da `S-box` da rodada i, a entrada ΔU_{i+1} da rodada seguinte é o resultado da aplicação da permutação a ΔV_i.

Queremos encontrar o mais provável trilha diferencial $D = (\Delta U_1, \Delta U_2, \Delta U_3, \Delta U_4)$ da cifra de Heys, ou, pelo menos, tal que cada diferencial $(\Delta U_i, \Delta V_i)$ seja entre os mais prováveis.

Todo diferencial consiste de 4 sub-diferenciais, correspondente aos 4 sub-blocos de 4 bites que constituem o bloco de 16 bites. Para encontrar uma tal trilha diferencial provável, queremos, em cada rodada,

- *maximizar a frequência* de cada sub-diferencial, isto é, o número de vezes a S-box substitui a diferença entrante pela diferença sainte do sub-diferencial;
- em particular, *minimizar* o *número* dos sub-diferenciais diferentes de 0 (que de ora em diante serão chamados de *ativos*).

Um exemplo de uma tal trilha D é a seguinte: Seja a diferença entrante na primeira rodada

$$\Delta U_1 = [0000\ 1011\ 0000\ 0000],$$

que é pela S-box 2 substituída por

$$\Delta V_1 = [0000\ 0010\ 0000\ 0000].$$

Pela permutação subsequente, obtemos a diferença entrante na segunda rodada

$$\Delta U_2 = [0000\ 0000\ 0100\ 0000]$$

que é pela S-box 3 substituída por

$$\Delta V_2 = [0000\ 0000\ 0110\ 0000].$$

Pelos bites número 2 e 3 diferentes de zero, obtemos na terceira rodada pela permutação a diferença entrante com dois sub-diferenciais ativos

$$\Delta U_3 = [0000\ 0010\ 0010\ 0000]$$

que é pelas S-boxes 2 e 3 substituída por

$$\Delta V_3 = [0000\ 0101\ 0101\ 0000].$$

Finalmente, obtemos pela permutação como entrada da quarta rodada

$$\Delta U_4 = [0000\ 0110\ 0000\ 0110].$$

Denote $S_{i,j}$ a substituição do sub-bloco j pela S-box na rodada i. Na nossa trilha diferencial, alistamos

- para cada rodada $i = 1, 2, 3$ e
- para cada sub-diferencial $j = 1, 2, 3, 4$ diferente de zero,

a probabilidade que, se a diferença ΔX entre na substituição $S_{i,j}$, então a diferença ΔY saia:

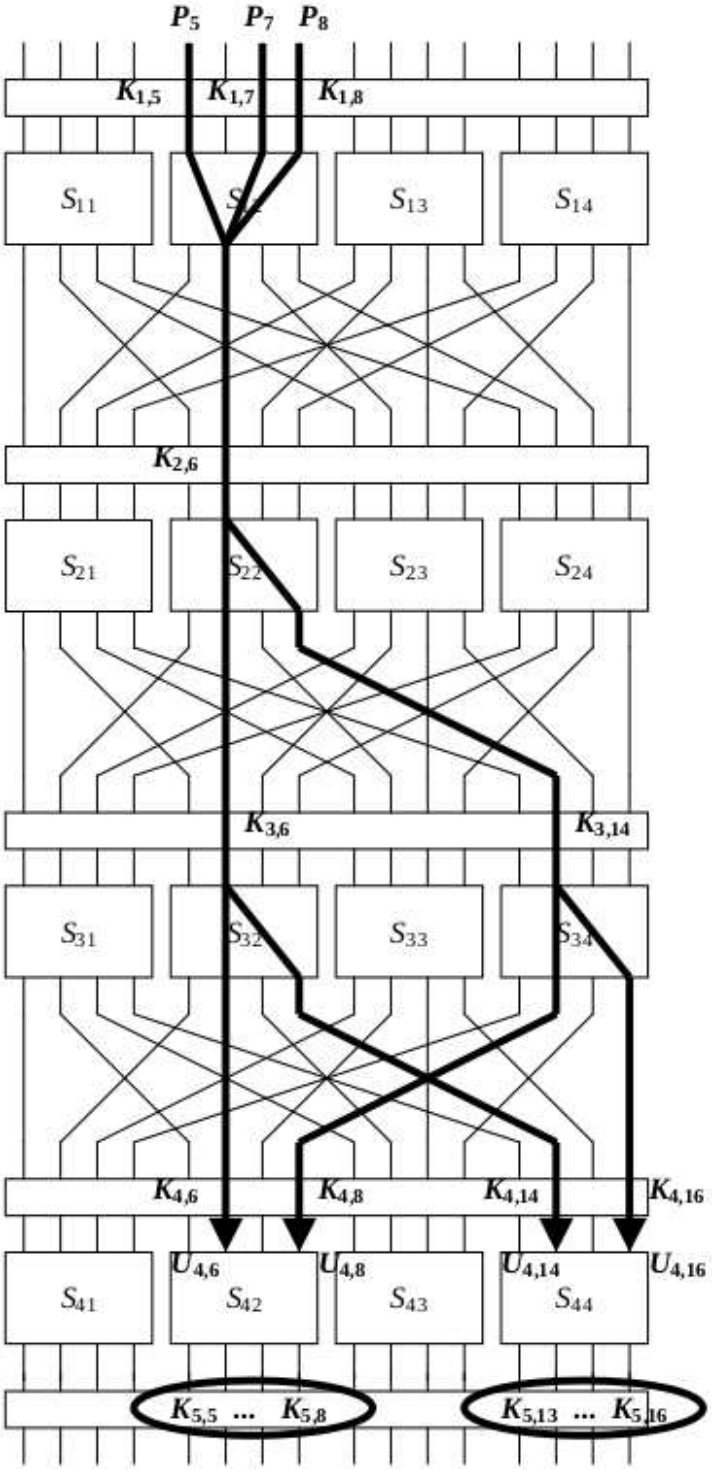


Figure 3. Sample Linear Approximation

Figura 21: Trilha Diferencial

Substituição	Entrada	Saída	Probabilidade
S_{12}	B	2	8/16
S_{23}	4	6	6/16
S_{32}	2	5	6/16
S_{33}	2	5	6/16

Se suponhamos que os diferenciais de uma rodada sejam independentes dos diferenciais da rodada anterior, então a probabilidade p_D da substituição

$$\Delta U_1 = [0000\ 1011\ 0000\ 0000]$$

por

$$\Delta U_4 = [0000\ 0110\ 0000\ 0110].$$

é o produto das probabilidades de cada substituição,

$$p_D = 8/16 \cdot 6/16 \cdot (6/16 \cdot 6/16) = 27/1024.$$

A fim de encontrar a chave, para

- toda possível combinação de bites $K_{5,5}, \ldots, K_{5,8}$ e $K_{5,13}, \ldots, K_{5,16}$
- um múltiplo inteiro m de $1/p_D \approx 38$ pares de textos claros U_1' e U_1'' com diferença ΔU_1,

o decifrador

1. cifra o par U_1' e U_1'',
2. inverte a cifração até a entrada da `S-box` na quarta rodada pela chave

   $$K_5 = [0000\ K_{5,5}, \ldots, K_{5,8}\ 0000\ K_{5,13}, \ldots, K_{5,16}]$$

   para obter o par Y_4' e Y_4'' e assim a diferença ΔY_4, e
3. compara a diferença ΔY_4 com ΔU_4 e, se coincidem, então aumenta a contagem n por 1.

Se para uma combinação de sub-blocos $K_{5,5}, \ldots, K_{5,8}$ e $K_{5,13}, \ldots, K_{5,16}$ vale $n/m \approx p_D$, isto é, a proporção entre

- o número n dos pares com esta coincidência e
- o número m dos pares ao total

é próxima da probabilidade p_D, então estes sub-blocos são provavelmente os sub-blocos 2 e 4 da chave 5 usada pelo cifrador.

Observação. Para concluir ter encontrado os sub-blocos corretos, usamos as hipóteses

1. que os diferenciais de uma rodada sejam independentes dos diferenciais da rodada anterior, e
2. que uma probabilidade de pares coincidentes próxima da calculada indica a chave correta.

Os dois não tem fundamento matemático rígido, mas são apenas plausíveis, porque, respetivamente:

1. Cada rodada visa difundir o máximo, isto é, tornar o valor de cada bite da saída praticamente independente de todos os bites da entrada.
2. É improvável existir uma chave que seja diferente mas reproduza a mesma probabilidade.

Notemos que para este ataque ser mais rápido, isto é, ser mais eficaz do que o ataque de força bruta (que simplesmente prova todas as chaves possíveis), é preciso

$$\#\{\text{ bites ativos }\} - \log_2 p_D < \#\{\text{ bites da chave }\} \tag{1}$$

onde

- um bite é *ativo* se pertence a um sub-diferencial ativo na penúltima rodada (na nossa trilha diferencial, são os bites do sub-diferencial número 2 e 4),
- a probabilidade p_D é a do diferencial ΔU_1 entrante da primeira rodada levar ao diferencial ΔU_4 entrante da penúltima rodada (na nossa trilha $\log_2 p_D = \log_2(27/1024) \approx -5$), e
- a chave é a da última rodada (que tem 16 bites nesta cifra).

Logo, é necessário que a trilha seja *estreita*, isto é, tenha poucos blocos ativos, para poder aprender se a chave provada é correta, isto é, coincide com a chave usada, apenas nestes blocos ativos (o que reduz o número de combinações logaritmicamente). No exemplo dado, apenas 2 dos 4 sub-diferenciais são ativos, o que permitiu ao decifrador aprender nestes 2 blocos se a chave é correta: o número de combinações que precisam ser provadas foi reduzido de $2^{16} = 65536$ a $2 \cdot 256 = 512$.

3.5 AES

Os algoritmos criptográficos simétricos modernos, tais como DES e AES, cifram dados em vez de textos, isto é, sequências de baites ou bites. Ambos, o DES e o AES, são *cifras de bloco*, isto é, agrupam o texto claro em blocos de um tamanho de baites determinado (no caso do AES, de 16, 24 ou 32 baites).

Lembremo-nos de que, idealmente, a rodada consistiria só

- na adição da chave e
- na substituição do bloco inteiro por outro.

Porém, o alfabeto desta substituição seria tão gigantesco que este ideal é praticamente inatingível; sobretudo sobre um hardware tão limitado quanto o de um cartão inteligente.

Para um bloco de por exemplo, 16 baites, esta tabela de substituição teria horrendos $2^256 \cdot 16$ baites. Por isso o AES substitui só cada baite, cada casa do bloco, uma tabela de substituição de $2^8 = 256$ entradas de 1 baite; em seguida, considera o bloco como quadrado de 4×4 casas, e

1. permuta as casas de cada linha, e
2. adiciona as colunas entre elas.

Os criadores do AES conseguiram demonstrar em Daemen e Rijmen (1999) que estas duas operações se complementam tão bem que, após várias iterações, quase compensam da ausência de uma substituição do bloco inteiro por outro. Isto é, conseguiram demonstrar que é imune contra um ataque de criptoanálise diferencial em Seção 3.4 pela sua ótima difusão. Para uma fonte mais detalhada, vide Daemen e Rijmen (2002).

Em vez disto, para conseguirem uma boa confusão e difusão (como definidas por Shannon), iteram ambas as operações (conhecidas pelos algoritmos antigos que operam sobre textos),

- a *substituição* e
- a *transposição*,

além da operação (nova sobre dados = sequências de bites),

- a adição e multiplicação de bites.

O algoritmo AES é o vencedor de uma competição anunciada pelo National Institute of Standards and Technology of the United States (NIST) de 1997 a 2000 para substituir o então padrão de criptografia simétrico, o Data Encryption Standard (DES).

Antes de tornar-se o AES, este algoritmo foi denominado pelos seus criadores Vincent Rijmen e Joan Daemen Rijndael, pelas letras iniciais dos seus sobrenomes.

No final da competição, restaram estes cinco algoritmos com as seguintes votações:

- Rijndael: 86 votos positivos, 10 negativos
- Serpent: 59 votos positivos, 7 negativos
- Twofish: 31 votos positivos, 21 negativos
- RC6: 23 votos positivos, 37 negativos
- MARS: 13 votos positivos, 84 negativos

Entre eles, nenhum deles se destacou pela sua maior segurança, mas o Rijndael, sim, pela sua simplicidade, ou clareza, e em particular economia computacional, na implementação. Como este algoritmo será a rodar por toda parte, por exemplo, nos processadores minúsculos de 8 bites nos cartões inteligentes (smartcards), a decisão foi tomada em favor do Rijndael.

Até hoje, este algoritmo continua firme e forte e é considerado o mais seguro; não há necessidade para outro padrão de algoritmo criptográfico simétrico.

E com efeito roda em todo lugar. Por exemplo, para cifrar uma rede sem fio, usa-se uma chave só, então a criptografia é simétrica. A opção mais segura, e logo mais recomendada, é a cifração por AES.

Vamos conhecer este algoritmo tão simples!

Cifração em Blocos. O algoritmo AES agrupa o texto claro (e as chaves) em retângulos de $4 \times B$ baites onde

$$B := \text{ número das colunas do bloco } = 4, 6 \text{ ou } 8.$$

Comumente, e para nós de agora para diante, B = 4, isto é, os retângulos são quadrados. Em base hexadecimal (= cujos algarismos são 0 – 9, A = 10, B = 11, C = 12, D = 13, E = 14 e F = 15), um tal quadrado tem por exemplo a forma

Figura 22: Cifração de uma rede sem fio pelo AES

A1	13	B1	4A
A3	AF	04	1E
3D	13	C1	55
B1	92	83	72

O corpo binário Rijndael. Não para implementá-lo, mas para entendermos as funções do AES, principalmente SubBytes e MixColumn, precisamos de uma digressão matemática: Um baite $b_7 \ldots b_1 b_0$ em $\{0,1\} \times \cdots \times \{0,1\}$ é considerado como polinômio com coeficientes binários por

$$b_7 \ldots b_1 b_0 \mapsto b_7 \mathrm{X}^7 + \cdots + b_1 \mathrm{X} + b_0$$

Por exemplo, o número hexadecimal $0x57$, ou baite 010101111, corresponde a

$$x^6 + x^5 + x^2 + x + 1.$$

Todas as adições e multiplicações têm lugar no *corpo binário* $\mathbb{F}_{2^8}$ com $2^8 = 256$ elementos, o qual é um conjunto de números com uma adição e multiplicação (que satisfaz a leis comutativa, associativa e distributiva; como, por exemplo, $\mathbb{Q}$) construído como segue: Seja

$$\mathbb{F}_2 = \{0,1\}$$

o *corpo de dois elementos* com

- as adições $1 + 0 = 0 + 1 = 1$ e $0 + 0 = 1 + 1 = 0$ (é a adição $\oplus$ dada por XOR), e as
- multiplicações $1 \cdot 0 = 0 \cdot 1 = 0 \cdot 0 = 0$ e $1 \cdot 1 = 1$ (é a multiplicação natural).

Seja

$$\mathbb{F}_2[\mathrm{X}] = \mathbb{Z}/2\mathbb{Z} = \text{ os polinômios sobre } \mathbb{F}_2 \,,$$

isto é, as somas finitas $a_0 + a_1 \mathrm{X} + a_2 \mathrm{X}^2 + \cdots + a_n \mathrm{X}^n$ para a_0, a_1, …, a_n em $\mathbb{F}_2$ e seja

Adição. A adição + de dois polinômios é a adição em $\mathbb{F}_2$ coeficiente a coeficiente. Isto é, como baites, a adição + é dada pela adição XOR.

Multiplicação. A multiplicação $\cdot$ é dada pela multiplicação natural seguida pela divisão com resto pelo polinômio

$$m(x) = x^8 + x^4 + x^3 + x + 1.$$

Por exemplo, em notação hexadecimal, $57 \cdot 83 = C1$, pois

$$(x^6 + x^4 + x^2 + x + 1)(x^7 + x + 1) = x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1$$

e

$$x^{13}+x^{11}+x^9+x^8+x^6+x^5+x^4+x^3+1 = (x^5+x^3+1)(x^8+x^4+x^3+x+1)+x^7+x^6+1$$

A multiplicação pelo polinômio 1 não muda nada, é o *elemento neutro*. Para qualquer polinômio $b(x)$, o algoritmo estendido de Euclides, calcula polinômios $a(x)$ e $c(x)$ tais que

$$b(x)a(x) + m(x)c(x) = 1.$$

Isto é, na divisão com resto de $a(x) \cdot b(x)$ por $m(x)$ sobra o resto 1. Quer dizer, a é o *inverso multiplicativo* em $\mathbb{F}_{2^8}$,

$$b^{-1}(x) = a(x) \text{ em } \mathbb{F}_{2^8}$$

Quando invertermos um baite b em $\mathbb{F}_{2^8}$, referimo-nos ao baite $a = b^{-1}$.

Com efeito, a

- adição e multiplicação em $\mathbb{F}_{2^8}$, a partir da adição e multiplicação em $\mathbf{F}_2[X]$,

é o análogo da

- adição e multiplicação em $\mathbb{Z}/m\mathbb{Z}$, a partir da adição e multiplicação em $\mathbb{Z}$ como descrito em Seção 5.2.

Rodadas. O AES cifra cada bloco iterativamente, em rodadas. Seja

$$\mathrm{R} := \text{ o número de rodadas}$$

que depende de B, há

- R = 10 rodadas para B = 4 colunas,
- R = 12 rodadas para B = 6 colunas
- R = 14 rodadas para B = 8 colunas.

Então, para nós, R = 10. Nestas rodadas, são geradas chaves, o texto claro substituído e transposto pelas seguintes operações:

1. Rodada $r = 0$:
    - AddRoundKey para adicionar (por XOR) a chave ao quadrado
2. Rodadas $r = 1, \ldots, \mathrm{R} - 1$ para cifrar, aplicando as seguintes funções:
    1. SubBytes para substituir cada casa do quadrado (isto é, byte, sequencia de oito bites) por uma sequência de bites melhor distribuída,
    2. ShiftRows para transpôr as casas das linhas do quadrado,
    3. MixColumn para adicionar as casas das colunas do quadrado entre elas,
    4. AddRoundKey para gerar uma chave a partir da chave da rodada anterior e adicioná-la (por XOR) ao quadrado.
3. Rodada $r = \mathrm{R}$ para cifrar, aplicando as seguintes funções:
    1. SubBytes
    2. ShiftRows
    3. AddRoundKey

    Isto é, em comparação às rodadas anteriores, a função MixColumn é omitida: Revela-se que MixColumn e AddRoundKey, após uma leve modificação de AddRoundkey, podem trocar a ordem sem modificar o resultado final de ambas operações. Nesta equivalente ordem, a operação MixColumn não aumenta a segurança criptográfica, sendo a última operação invertível sem chave. Então, pode ser omitida.

O CrypTool 1 oferece no Menu Individual Procedures -> Visualization of Algorithms -> AES

- uma entrada `Animation` para ver a animação em Figura 23 das rodadas, e
- uma entrada `Inspector` em Figura 27 para experimentar com os valores do texto claro e da chave.

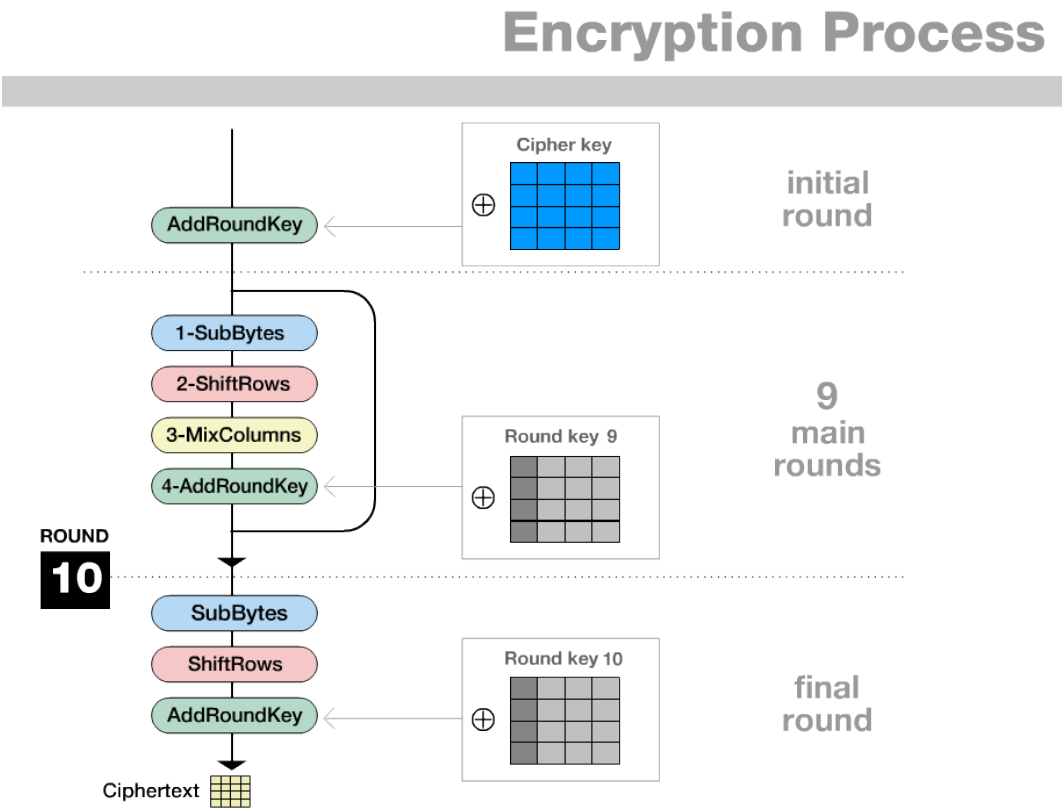


Figura 23: As rodadas do `AES` no `CrypTool 1`

Vamos descrever estas funções em mais detalhes:

`SubBytes`. `SubBytes` substitui cada baite do bloco por outro baite pela tabela de substituição `S-box` dada abaixo.

Para calcular o valor da casa pelo qual o `S-box` substitui o dado baite:

1. Calcula o seu inverso multiplicativo B em $\mathbb{F}_{2^8}$,

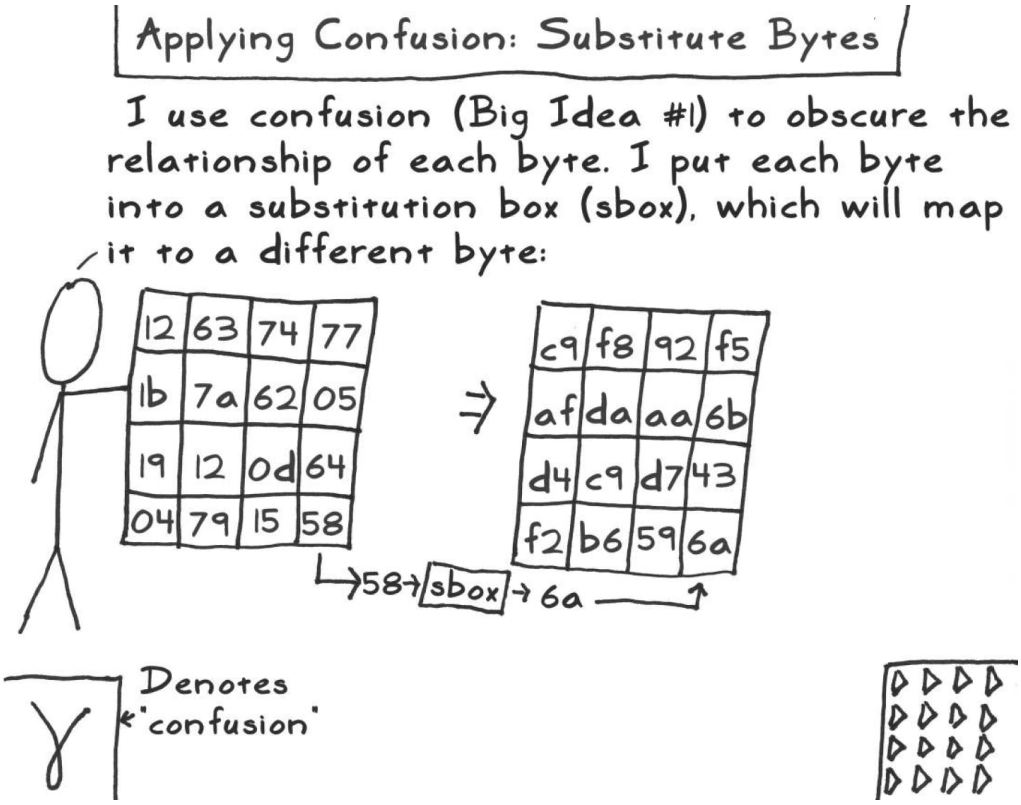


Figura 24: substituição no algoritmo AES

2. Calcula

$$a_i = b_i + b_{i+4} + b_{i+5} + b_{i+6} + b_{i+7} + c_i$$

onde $i = 0, 1, \ldots, 7$ é o índice de cada bite de um baite, e

- B = $b_7b_6b_5b_4b_3b_2b_1b_0$ é o baite de entrada,
- A = $a_7a_6a_5a_4a_3a_2a_1a_0$ é o baite de saída da operação e
- c é o baite constante 01100011.

Em forma matricial,

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

e como tabela calculada de antemão em notação hexadecimal (onde o número da linha corresponde ao primeiro e o número da coluna ao segundo digito do baite a ser substituído):

63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

`ShiftRows`. `ShiftRows` traslada a linha l do quadrado l posições para a esquerda (onde as linhas são enumeradas a partir de zero, isto é, l percorre $0, 1, 2$ e 3 e a traslação é cíclica). (Em particular, primeira linha *não* é trasladada.) Visualmente, o quadrado com entradas

B_00	B_{01}	B_{02}	B_{03}
B_10	B_{11}	B_{12}	B_{13}
B_20	B_{21}	B_{22}	B_{23}
B_30	B_{31}	B_{32}	B_{33}

é transformada em um com entradas

B_00	B_{01}	B_{02}	B_{03}
B_11	B_{12}	B_{13}	B_{10}
B_22	B_{23}	B_{20}	B_{21}
B_33	B_{30}	B_{31}	B_{32}

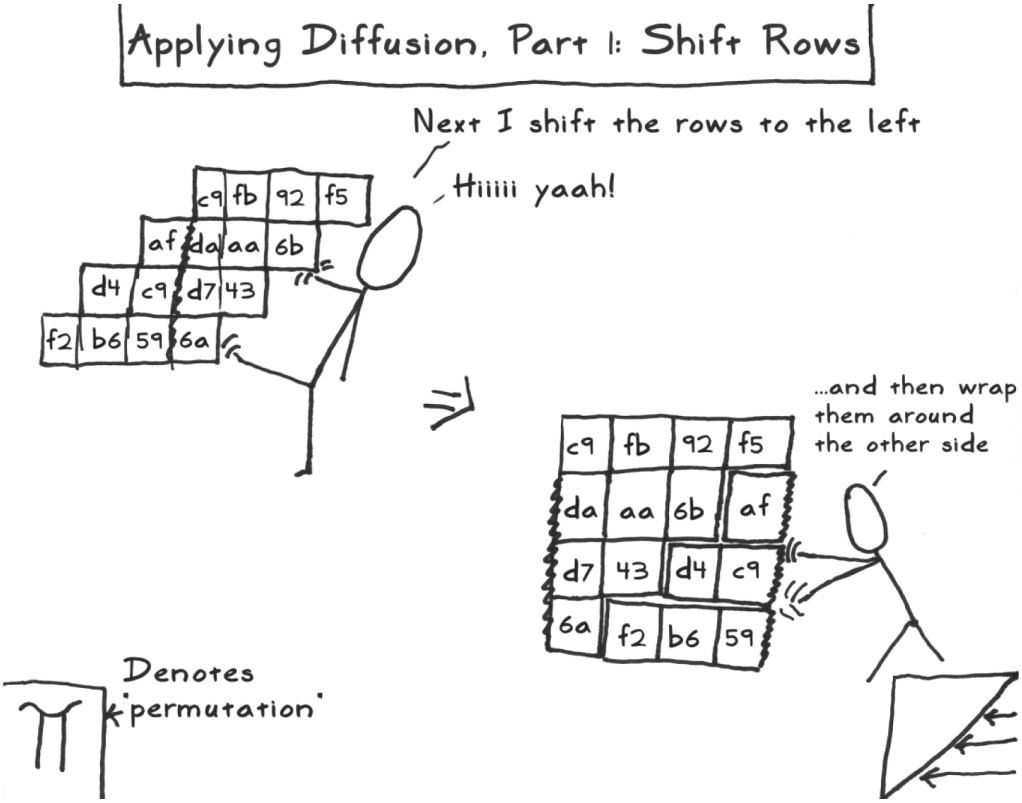


Figura 25: transposição no algoritmo AES

`MixColumn`. `MixColumn` multiplica cada coluna do bloco por uma matriz fixa. Mais exatamente,

- se B_j (com coeficientes $b_{0,j}$, $b_{1,j}$, $b_{2,j}$ e $b_{3,j}$) corresponde à coluna j do bloco de entrada, e
- se A_j (com coeficientes $a_{0,j}$, $a_{1,j}$, $a_{2,j}$ e $a_{3,j}$) corresponde à coluna j do bloco de saída da operação,

então

$$\begin{pmatrix} a_{0,j} \\ a_{1,j} \\ a_{2,j} \\ a_{3,j} \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} b_{0,j} \\ b_{1,j} \\ b_{2,j} \\ b_{3,j} \end{pmatrix}$$

Por exemplo, o baite $a_{0,j}$ é calculado por

$$a_{0,j} = 2 \cdot b_{0,j} + 3 \cdot b_{1,j} + b_{2,j} + b_{3,j}$$

`AddRoundKey`. `AddRoundKey` adiciona, pela operação `XOR`, a chave $\mathrm{W}(r)$ da rodada atual r ao quadrado B do texto cifrado, isto é,

$$\mathrm{B} \oplus \mathrm{W}(r).$$

A chave é gerada coluna a coluna. Denotemo-las por $\mathrm{W}(r)_0$, $\mathrm{W}(r)_1$, $\mathrm{W}(r)_2$ e $\mathrm{W}(r)_3$; isto é,

$$\mathrm{W}(r) = \mathrm{W}(r)_0 \mid \mathrm{W}(r)_1 \mid \mathrm{W}(r)_2 \mid \mathrm{W}(r)_3.$$

Como a chave tem 16 baites, cada coluna tem 4.

1. A primeira chave $\mathrm{W}(0)$, isto é, da primeira rodada, é dada pela chave inicial W.
2. Para $r = 1, \ldots, \mathrm{R}$ (onde R é o número total de rodadas, $\mathrm{R} = 10$ para nós), as quatro colunas $\mathrm{W}(r)_0$, $\mathrm{W}(r)_1$, $\mathrm{W}(r)_2$ e $\mathrm{W}(r)_3$ da nova chave são geradas a partir das colunas da antiga chave $\mathrm{W}(r-1)$ como segue:
    1. A primeira coluna $\mathrm{W}(r)_0$ é dada por

    $$\mathrm{W}(r)_0 = \mathrm{W}(r-1)_0 \oplus \mathrm{ScheduleCore}(\mathrm{W}(r-1)_3);$$

    isto é, a última coluna da chave da rodada precedente mais o resultado de `ScheduleCore` aplicada à primeira coluna da chave da rodada precedente (que denotemos por T); aqui `ScheduleChore` é a composição das transformações:

1. `SubWord`: Substitui cada um dos 4 baites de T segundo a `S-box` em `SubBytes`.
2. `RotWord`: Traslada T um baite à esquerda.
3. `Rcon`(r): Adiciona (por `XOR`) a T o valor constante, em notação hexadecimal, $[(02)^{r-1}$ 00 00 00] onde a potencia (= produto iterado) é calculada no corpo de Rijndael $\mathbb{F}_{2^8}$. Isto é, o único baite que muda é o primeiro, pela adição, ou do o valor 2^{r-1} (para $r \leq 8$), ou do valor 2^{r-1} em $\mathbb{F}_{2^8}$ para $r = 9,10$.

2. As colunas seguintes $\mathrm{W}(r)_1$, $\mathrm{W}(r)_2$ e $\mathrm{W}(r)_3$ são dadas, para $i = 1,2$ e 3, por

$$\mathrm{W}(r)_i = \mathrm{W}(r)_{i-1} \oplus \mathrm{W}(r-1)_i;$$

isto é, a coluna precedente da chave da rodada atual mais a coluna atual da chave da rodada precedente.

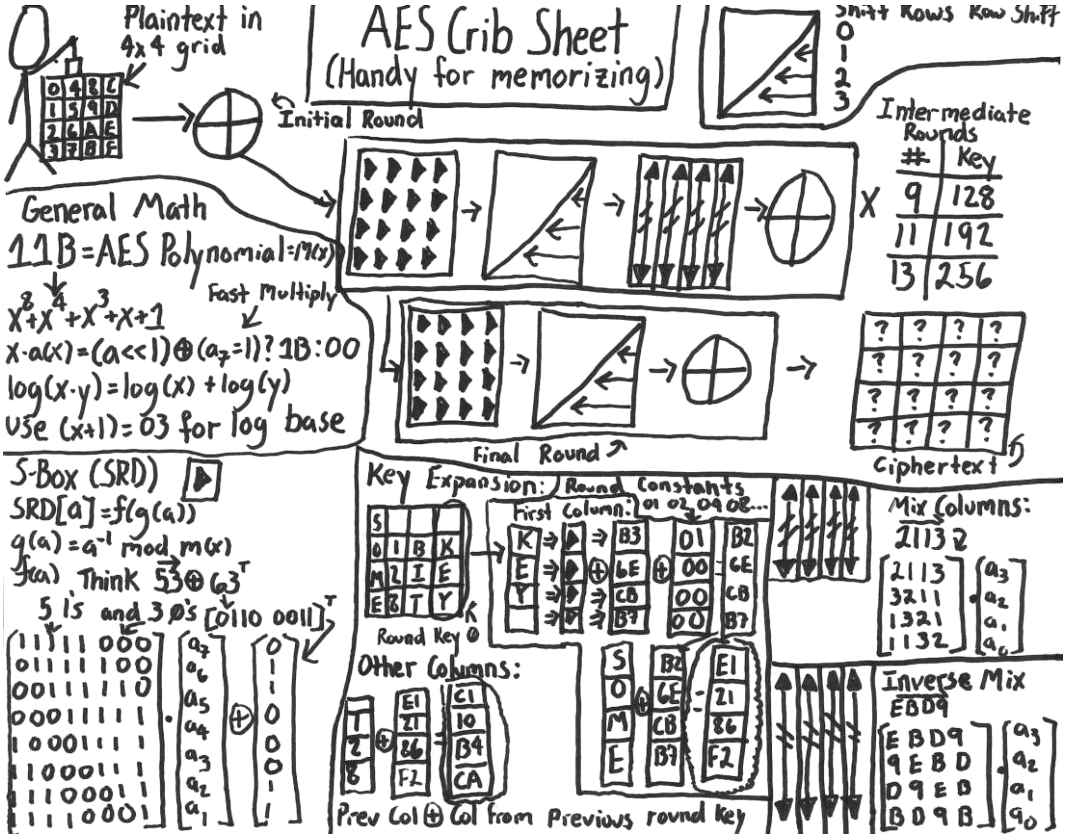


Figura 26: Algoritmo Completo

Observamos que a única transformação que *não é afim* (isto é, dada por um polinômio de grau 1, ou, equivalentemente, a composição de uma aplicação linear e um traslação) é a inversão no corpo $\mathbb{F}_{2^8}$ na operação `SubBytes`. Com efeito

1. Na operação `SubBytes` são aplicadas, nesta ordem,
    1. a inversão em $\mathbb{F}_{2^8}$,
    2. uma aplicação linear, e
    3. a traslação por um vetor constante.
2. `ShiftRows` é uma permutação, em particular,linear.
3. `MixColumn` é uma adição, em particular, linear.
4. `AddRoundKey` é a traslação pela chave da rodada.

Quanto às metas da *difusão* e *confusão*, podemos ressaltar que a cada etapa são substituídos e transpostos cerca da metade dos bites (no `SubBytes`) ou baites (no `MixColumn` e `ShiftRows`). Para convencer-se da complementaridade das simples operações para segurança criptográfica, isto é, que geram em conjunção alta confusão e difusão após de poucas iterações,

- da substituição do alfabeto, e
- da permutação do texto, em particular,
    - da permutação entre as casas da *linha*, e
    - da permutação entre as casas da *coluna*,

vale a pena experimentar em `Individual Procedures -> Visualization of Algorithms -> AES -> Inspector` com uns valores patológicos, por exemplo:

- Todas as entradas da chave e do texto claro iguais a `00`, e
- todas as entradas da chave igual a `00` e do texto claro iguais a `00` exceto uma entrada sendo igual a `01`, isto é, mudar um singelo bite.

Vemos como se espalha esta pequena diferença inicial, gerando já resultados totalmente diferente após, digamos, quatro rodadas! Isto torna plausível a imunidade do `AES` contra a criptoanálise diferencial como será mostrado em Seção 3.6.

No caso que todas as entradas da chave e do texto claro sejam iguais a `00`, compreendemos também o impacto da adição da constante $\texttt{Rcon}(r)$ à chave em cada rodada: é dela que provém toda a confusão!

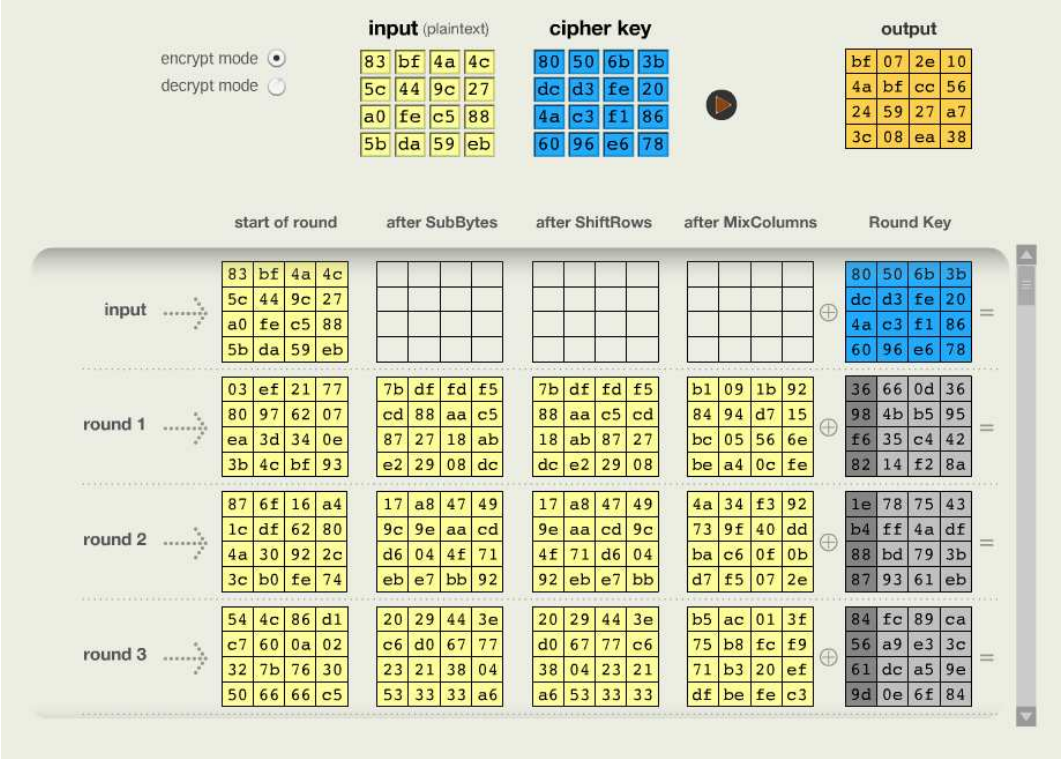


Figura 27: Os valores dos bloques ao longo das rodadas no AES no CrypTool 1

3.6 Imunidade do AES contra a Criptoanálise Diferencial

O principio principal é o de Shannon, da boa *difusão* e *confusão*. Idealmente, cada bite cifrado depende de cada bite da chave e do texto claro; se um bite da chave ou do texto claro muda, então todo bite do texto cifrado muda com uma probabilidade de 50%.

Contra a criptoanálise diferencial, isto é, para evitar trilhas diferenciais altamente prováveis, o AES usa

- uma S-box cuja tabela de distribuição diferencial tem como entrada máxima $p = 1/2^6$ (sendo $1/2^8$ o mínimo), mas
- principalmente a estratégia de *trilhas largas* para obter os diferenciais com muitos baites *ativos*, isto é, diferentes de 0, ao longo da trilha diferencial do início à saída.

Trilhas e Pesos. O *quociente de propagação* $p(\Delta X \to \Delta Y)$ para um diferencial D , uma diferença entrante ΔX e diferença sainte ΔY para uma aplicação h é

$$p(\Delta X \to \Delta Y) = 2^{-n}\#\{X' \text{ tal que a diferença das imagens de } X' \text{ e } X'' = X' \oplus \Delta \text{ seja } \Delta Y\}$$

O *peso de restrição* $w(\Delta X \to \Delta Y)$ de um diferencial é

$$w(\Delta X \to \Delta Y) = -\log_2 p(\Delta X \to \Delta Y)$$

Se a aplicação $h\colon \Delta X \mapsto \Delta Y$ é *afim* (isto é, $h = A \cdot + k$, um polinômio de grau 1, ou, equivalentemente, a composição de uma aplicação linear A e um traslação $\cdot + k$), o quociente de propagação é 1 para $\Delta Y = A\Delta X$ e 0 senão.

Se $h = h_1 \oplus \cdots \oplus h_n$, então $h\colon \Delta X \mapsto \Delta Y$ para $\Delta X = \Delta X_1 \oplus \cdots \oplus \Delta X_n$ e $\Delta Y = \Delta Y_1 \oplus \cdots \oplus \Delta Y_n$ tem quociente de propagação

$$p(h) = p(h_1) \cdots p(h_n)$$

e peso de restrição

$$w(h) = w(h_1) + \cdots + w(h_n).$$

Uma *trilha diferencial* é uma sequência finita de diferenças $\Delta U_1 \to \Delta U_2 \to \ldots$ tal que o resultado do diferencial $\Delta U_{i-1} \to \Delta U_i$ seja a entrada do diferencial seguinte $\Delta U_i \to \Delta U_{i+1}$.

O *peso de restrição* $w(D)$ de uma trilha diferencial é

$$w(D) = w(\Delta U_1) + w(\Delta U_2) + \cdots$$

É inviável calcular o quociente de propagação de uma trilha diferencial, mas fácil calcular o seu peso de restrição. Se os diferenciais da trilha diferencial são pouco correlacionados, então

$$p(D) \approxeq 2^{-w(D)} \qquad (*)$$

Esta aproximação, por exemplo, para DES, é muito boa se $w(D) < n$. Se $w(D) > n$, então $(*)$ não pode valer mais. Neste caso, em média apenas uma fração $2^{n-w(D)}$ das trilhas diferenciais D com peso $w(D)$ com uma diferença inicial ΔU_1 ocorrem.

Observamos que o peso de restrição de uma trilha independe das chaves das rodadas, porque cada peso de restrição é. Contudo, o quociente de propagação da trilha depende das chaves. Porém, pela aproximação $(*)$, uma trilha diferencial com peso de restrição significantemente $< n$ tem um quociente de propagação que pode ser considerado independente das chaves das rodadas.

Trilhas largas. O passo da permutação é subdividido em dois sub-passos

- `ShiftRows`, que opera sobre cada linha do bloco
- `MixColumn`, que opera sobre cada coluna do bloco

e os quais, em combinação, conseguem uma excelente confusão e difusão porque garantem trilhas largas, diferenciais com muitos baites *ativos*, após poucas rodadas. Mais exatamente, Teorema 2 mostrará que após quatro rodadas, qualquer trilha diferencial tem 25 baites ativos, isto é, o dos baites diferentes de 0 em todos os diferenciais na trilha diferenciais é ≥ 25.

Daqui por diante, um *vetor* é uma sequência finita. Um vetor de bites (por exemplo, um bite, um baite, um vetor de baites) é *ativo* se é diferente de 0. Para um vetor de baites a, seja

$$\mathrm{W}(a) := \#\{ \text{ baites ativos } \},$$

o *peso* de a. Recordemo-nos de que uma aplicação F entre vetores de baites é *linear* se $\mathrm{F}(x + y) = \mathrm{F}(x) + \mathrm{F}(y)$.

Definição. Para uma aplicação linear F entre espaços de vetores de baites, seja

$$\mathrm{W}(\mathrm{F}) := \min\{\mathrm{W}(v) + \mathrm{W}(\mathrm{F}(v)) \text{ para todos os vetores de baites } v\}$$

o seu *número de ramificação*.

Isto é, para $\mathrm{W}(\mathrm{F})$ ser grande, quanto menos baites diferentes de 0 em v, tanto mais em $\mathrm{F}(v)$. Como `F = MixColumn` opera sobre vetores de 4 baites, $\mathrm{W}(\mathrm{F}) \leq 5$. Com efeito, vale igualdade:

Lema 1. O número de ramificação da aplicação linear `MixColumn` é 5.

Em particular, para qualquer vetor v que tem um único baite diferente de 0, a sua imagem $\mathrm{F}(v)$ tem quatro baites diferentes de 0.

Para $i = 1, 2, \ldots,$

- denote a_{i-1} os baites ativos do bloco no início da rodada i, e
- denote b_{i-1} os baites ativos do bloco após aplicação de `ShiftRows` na rodada i.

Em particular, é aplicado `MixColumn` a_{i-1} para obter b_i.

Uma trilha (diferencial) é uma sequência finita de diferenciais (de pares de 16 baites); em particular, é uma sequência finita, ou vetor, de baites e podemos contar o seu peso, isto é, contar ao início de cada rodada o número de baites diferentes de 0 e somá-los. Isto é, o *peso* $\mathrm{W}(\mathrm{D})$ de uma trilha D é $\mathrm{W}(a_0) + \mathrm{W}(a_1) + \cdots$.

Como `SubBytes` opera baite a baite, e `AddRoundKey` não muda o diferencial, observamos que as únicas operações que mudam os baites ativos em uma trilha diferencial são `ShiftRows` e `MixColumn`. Além disso,

- a única operação que muda o número de *baites* ativos W do bloco é `MixColumn`, e
- a única operação que muda o número de *colunas* ativas $\mathrm{W_C}$ do bloco é `ShiftRows`.

Em particular, $\mathrm{W}(a_i) = \mathrm{W}(b_i)$ e $\mathrm{W_C}(b_i) = \mathrm{W_C}(a_{i+1})$.

Teorema 1. Para uma trilha diferencial D sobre duas rodadas,

$$\mathrm{W}(\mathrm{D}) \geq 5\mathrm{W_C}(a_1)$$

onde $\mathrm{W}(\mathrm{D})$ é o número de baites ativos da trilha e $\mathrm{W_C}(a_1)$ é o número de colunas ativas na entrada da segunda rodada.

Demonstração: Como pelo Lema 1 o número de ramificação de `MixColumn` é ≥ 5, em cada coluna, a soma do número de baites ativos em a_0 e a_1 é ≥ 5. Logo,

$$\text{peso da trilha } = \#\{ \text{ baites ativos em } a_0 \text{ e } a_1 \} \geq 5\mathrm{W_C}(a_1)$$

Em particular, toda trilha sobre duas rodadas tem ≥ 5 baites ativos.

Lema 2. Em uma trilha diferencial sobre duas rodadas,

$$\mathrm{W_C}(a_0) + \mathrm{W_C}(a_2) \geq 5$$

onde $\mathrm{W_C}(a_0)$ e $\mathrm{W_C}(a_2)$ são os números de colunas ativas na entrada e saída da trilha.

Demonstração: Como `ShiftRows` move cada baite diferente a uma coluna diferente, observamos

$$\#\{ \text{ colunas ativas depois } \} \geq \max\{\#\{ \text{ baites ativos } \} \text{ em cada coluna antes } \},$$

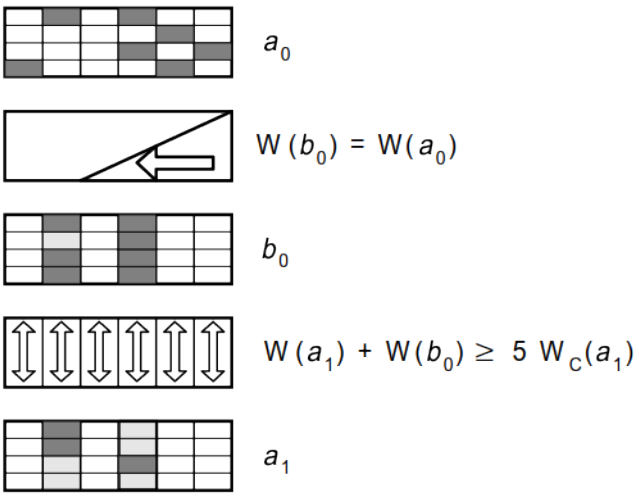


Figura 28: Teorema 1 com $W_C(a_1) = 2$

e

$$\#\{\text{ colunas ativas antes }\} \geq \max\{\#\{\text{ baites ativos }\} \text{ em cada coluna depois }\},$$

quer dizer, antes e depois de `ShiftRows`.

Seja g uma coluna (isto é, os baites do bloco nas posições dadas por esta coluna, independente da rodada) que é ativa em a_1, na entrada da segunda rodada (ou, equivalentemente, em b_0, após `ShiftRows` na primeira rodada). Temos

$$\#\{\text{ colunas ativas em } a_0 \} \geq \#\{\text{ baites ativos na coluna } g \text{ em } b_0 \},$$

e, da mesma maneira,

$$\#\{\text{ colunas ativas em } b_1 \} \geq \#\{\text{ baites ativos na coluna } g \text{ em } a_1 \}.$$

Como `MixColumn` tem número de ramificação 5,

$$\#\{\text{ baites ativos em } g \text{ em } b_0\} + \#\{\text{ baites ativos em } g \text{ em } a_1\} \geq 5$$

e pois

$$\#\{\text{ colunas ativas em } b_1 \} = \#\{\text{ colunas ativas em } a_2 \},$$

segue

$$\#\{\text{ colunas ativas em } a_0 \} + \#\{\text{ colunas ativas em } a_2 \} \geq 5.$$

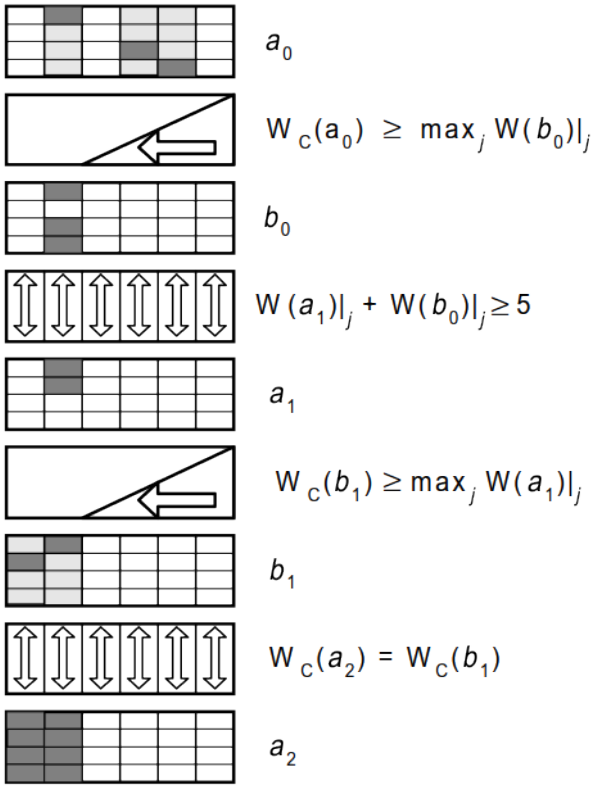


Figura 29: Lema 2 com uma coluna ativa em a_1

Teorema 2. Toda trilha diferencial sobre quatro rodadas tem ≥ 25 baites ativos.

Demonstração: Tem-se

$$\#\{ \text{ baites ativos } \} \geq 5[\mathrm{W_C}(a_1) + \mathrm{W_C}(a_3)] \geq 25,$$

onde

- os pesos $\mathrm{W_C}(a_1)$ e $\mathrm{W_C}(a_3)$ denotam as colunas ativas na entrada da primeira e após a segunda rodada,
- a primeira desigualdade vale pelo Teorema 1, aplicado às primeiras duas e à últimas duas rodadas, e
- a segunda desigualdade vale pelo Lema 2.

Corolário: Não há trilhas com uma probabilidade maior que 2^{-150} para 4 rodadas de AES.

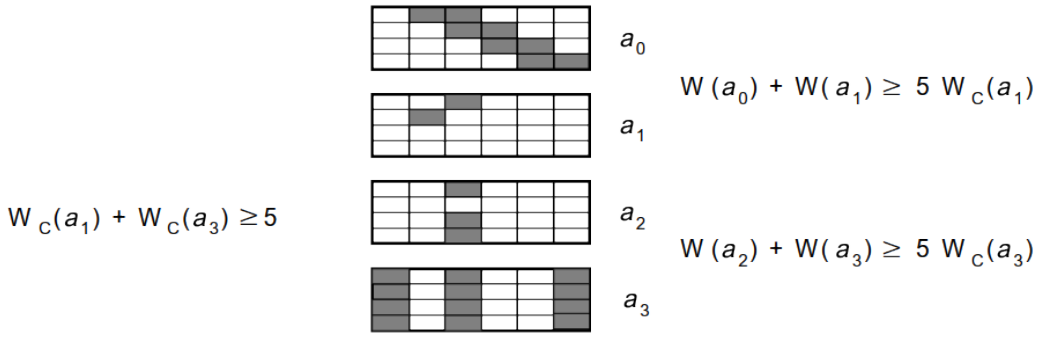


Figura 30: Teorema 2

Demonstração: Como a S-box tem um quociente de propagação de 2^{-6} e uma trilha com 4 rodadas pelo menos 25 S-boxes ativas. Por Capítulo 5 em Daemen (1995), a probabilidade da trilha é aproximadamente o produto da probabilidade de cada S-box.

4 Criptografia Assimétrica

A criptografia assimétrica foi sugerida pela primeira vez, publicamente, em Diffie e Hellman (1976).

Figura 31: Diffie e Hellman, os inventores da ideia da criptografia assimétrica

Por trás dela figura uma *função alçapão* (mais especificamente, nesse artigo, a exponencial modular), uma função invertível que é facilmente computável, mas cujo inverso é dificilimamente computável em ausência de uma informação adicional, a *chave secreta.*

Para cifrar, aplica-se a função, para decifrar, seu inverso com a chave secreta. Por exemplo, na abordagem segundo Diffie e Hellman, esta função é a exponencial, porém, sobre outro domínio que os números reais a que estamos acostumados, explicado em Seção 5.1.

Com efeito, Diffie e Hellman (1976) introduziu apenas um esquema para trocar uma chave secreta através de um canal inseguro. Foi foi realizado pela primeira vez

- em Rivest, Shamir, e Adleman (1978), em que o algoritmo criptográfico `RSA` foi criado, e
- pelo algoritmo `ElGamal`, mais recente, mas é o exemplo mais próximo do esquema original.

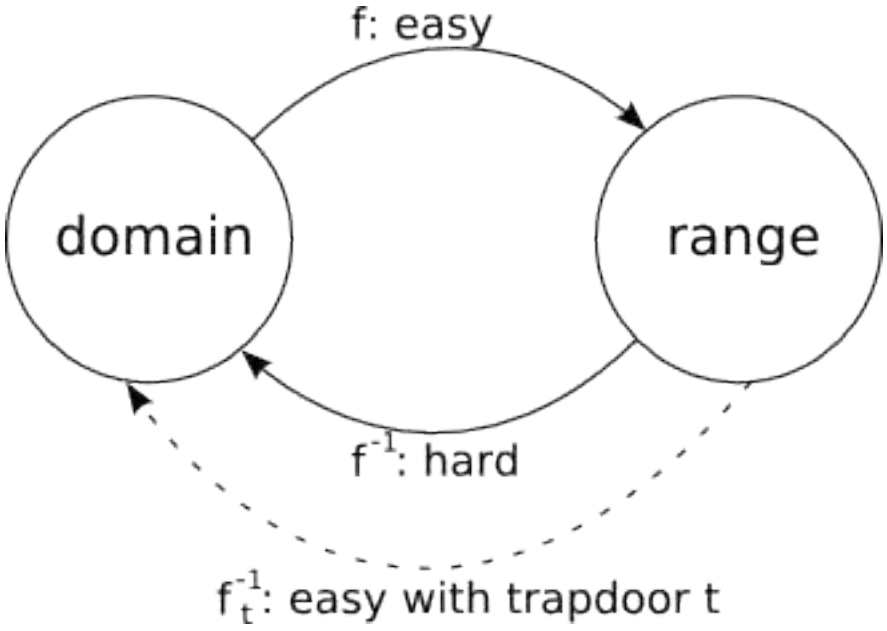


Figura 32: função alçapão

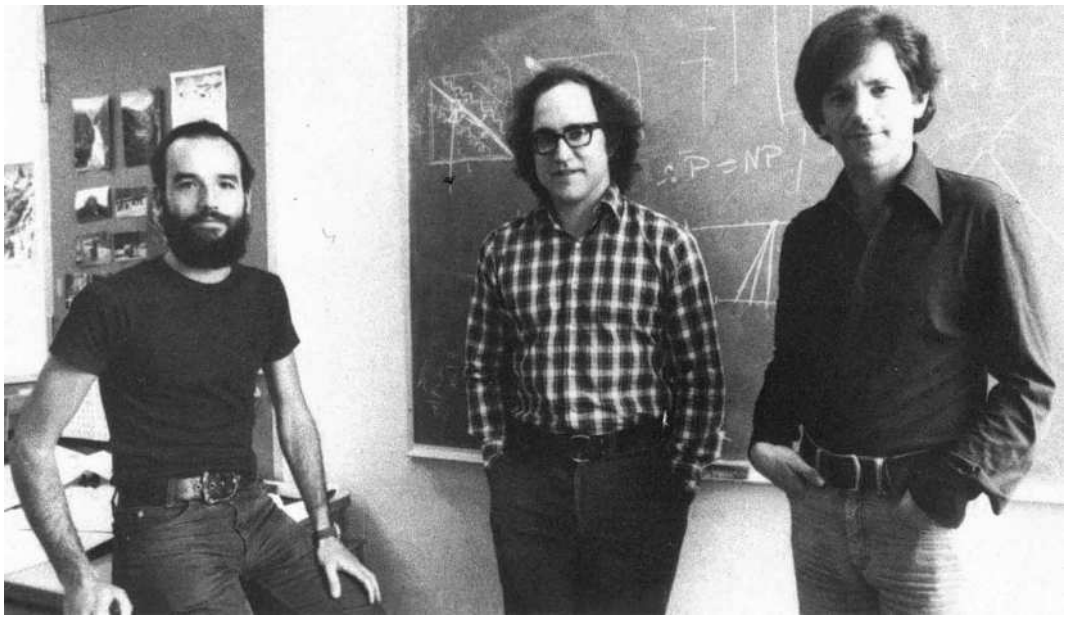


Figura 33: Os inventores do algoritmo RSA, Rivest, Shamir e Adleman

4.1 Chave pública e privada

Recordemo-nos de que há duas chaves, uma chave *pública* e outra *privada*, Comumente:

- A chave *pública* usa-se para cifrar, enquanto a chave *privada* para decifrar.

Assim, um texto pode ser transferido do cifrador (Alice) a uma pessoa só, o decifrador (Bob).

Os papéis das chaves públicas e privada podem ser invertidos:

- A chave *privada* usa-se para cifrar, enquanto a chave *pública* para decifrar.

Assim, o cifrador pode provar a todos os decifradores (os que têm a chave pública) a sua posse da chave privada; a *assinatura digital.*

A teoria (quer dizer a matemática) por trás da cifração pela chave pública (mensagens digitais) ou privada (assinatura digital) é quase a mesma; unicamente os papéis dos argumentos da função alçapão são invertidos. (Por exemplo, no algoritmo RSA, esta troca de variáveis é verdadeiramente tudo o que acontece.) Porém, na prática, geralmente são cifrados:

- *acolchamentos* do texto claro pela chave pública (para evitar patologias que revelam a chave quando o texto é muito curto), e
- uma *soma de verificação* (ou *hash*, isto é, uma função que manda praticamente sempre textos diferentes a números diferentes) pela chave privada. (Este hash é geralmente um *hash criptográfico*, isto é, dada a sua saída, é praticamente impossível deduzir a sua entrada. Isto permite de verificar a *integridade* da mensagem assinada, isto é, que ela não tenha sido modificada no caminho.)

Isto é, enquanto

- para a cifração pela chave pública, a função usada para primeiro transformar o texto (o acolchamento) é facilmente invertível,
- para a cifração pela chave privada, a função usada para primeiro transformar o texto (o hash) é dificilimamente invertível.

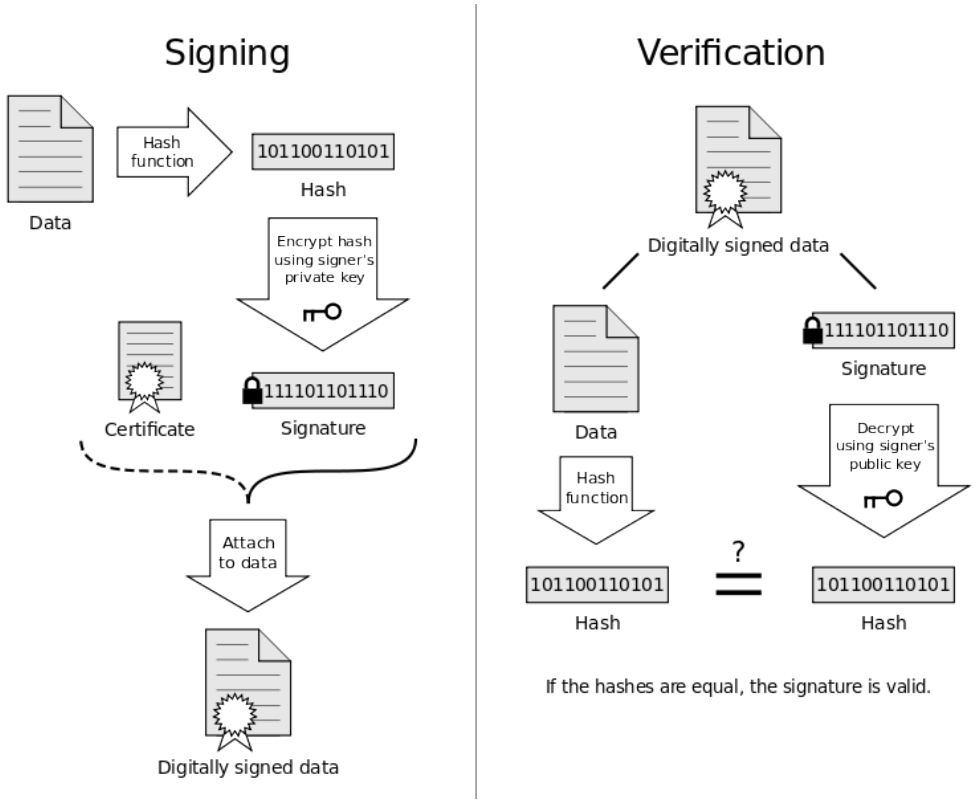


Figura 34: assinatura digital

Subchaves Efêmeras. Uma **subchave** de uma chave principal é uma chave (cuja soma de verificação criptográfica é) assinada pela chave principal.

O dono frequentemente cria subchaves a fim de usar a chave (pública e privada) principal unicamente

- para *assinar* (ou revocar a assinatura d')
  - a chave de outra pessoa,
  - de uma subchave
- para revocar uma chave (isto é, assinar uma revocação),
- para mudar a data de expiração da chave principal ou de uma subchave.

e as subchaves para os outros *fins quotidianos* (assinar e decifrar) com um maior risco de comprometimento.

Desta maneira o eventual comprometimento de uma subchave (o que é mais provável pelo seu uso quotidiano!) não comprometerá a chave principal. Neste caso,

- o dono revoca-a (publica uma nota de invalidação da chave pública comprometida que é digitalmente assinada pela sua chave principal privada),
- cria outra.

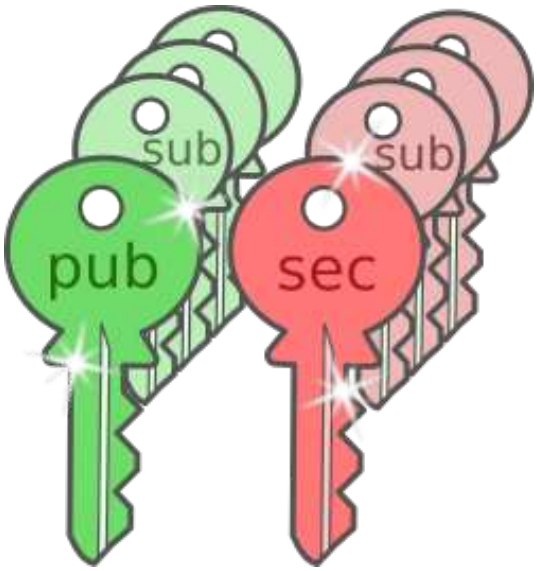


Figura 35: subchaves

Estudamos como as *subchaves* funcionam em prática no aplicativo criptográfico principal, o programa de linha-de-comando GPG discutido em Seção 4.6. Uma boa referência é https://wiki.debian.org/Subkeys.

Subchaves para o Dia-a-Dia. Para mais segurança, o usuário cria (por exemplo, em GPG)

- primeiro uma chave (pública e privada) **principal** e
- depois várias **subchaves** com data de validade, para o uso no dia-a-dia:
  - uma subchave para **decifrar** no dia-a-dia (por exemplo, os e-mails cifrados que recebe), e
  - uma subchave para **assinar** no dia-a-dia (por exemplo, os e-mails que envia),

cada uma com a sua data de **validade**. Antes das suas invalidações, ou prolonga-as, ou revoca-as e cria outras.

```
 Empreinte de la clef = F43E 7A29 8FB9 77DB 7B2A  5003 992E 7026 125D FBE9
sub   rsa4096/0x7A7AF3571E1B5A03 2016-10-06 [S] [expire : 2019-10-06]
 Empreinte de la clef = 2D0E 8856 08CC B7C6 DAAD  2524 7A7A F357 1E1B 5A03
sub   rsa4096/0x039B4513D99ACDC5 2016-10-06 [E]
```

Figura 36: Impressões digitais de subchaves no GPG para **S**ubscrever e **E**ncriptar

Quanto ao uso de chaves diferentes para assinar e cifrar,

- ele é necessário para alguns algoritmos, por exemplo,
  - sim, no algoritmo ElGamal,
  - mas não no RSA.

  (Ambos os algoritmos serão apresentados em Seção 8.)
- ele é mais seguro (porém, mais inconveniente, o que pode levar a certo desleixo do usuário e logo na prática até mais inseguro!) ter uma chave para decifrar e outra para assinar,
  - pois
    * é importante guardar uma cópia da chave privada para decifrar (para ainda poder ler os arquivos cifrados pela sua chave pública correspondente)
    * é inútil guardar uma cópia da chave privada para assinar (porque uma vez perdida outra pessoa pode assinar com ela também)
  - pois, por exemplo no algoritmo RSA, a assinatura e a decifração (pela chave privada) são matematicamente iguais!

    Por isso, a assinatura (pela chave privada) de um documento cifrado pela chave pública equivale à decifração! Porém, esta possibilidade é teórica, mas não pratica: Todas as implementações do RSA protegem o usuário pelo fato que sempre e exclusivamente
    * cifram recheamentos de um documento, e
    * assinam uma soma de verificação *criptográfica* (que não permite deduzir o seu conteúdo original) de um documento.

Melhores Práticas para a Gerência das Chaves. Unicamente a chave principal é imutavelmente ligada a identidade do dono, e todas as outras substituíveis: Enquanto a

- chave **principal** é guardada em um **cofre** em casa e sobretudo só vê a luz quando precisar de assinar chaves (ou do dono, ou de outrem [por exemplo, para estabelecer a teia de confiança]). Na prática,
  - é armazenada num pendrive ou cartão de memória,
  - e para ter mais durabilidade até impressa, por exemplo:
    * Pelo programa `paperkey` que extrai a parte secreta (do arquivo) da chave privada (isto é, omite todas as informações publicas como
      · a identidade,
      · o algoritmo, ...),

      e codifica esta parte em notação hexadecimal (armazenado em um arquivo de texto). (Por isso, se a chave tem, por exemplo, 4096 bites, o arquivo gerado por `paperkey` tem ≥ 1024 baites.) Este comando

      ```
      gpg --export-secret-key 0xAE46173C6C25A1A1! > ~/private.sec
      paperkey --secret-key ~/private.sec > ~/private.paperkey
      ```

      · exporta apenas (indicado pelo `!`) a chave secreta principal (`0xAE46173C6C25A1A1`) das chaves privadas, e
      · extrai e converte-a por `paperkey` em um arquivo de texto.
    * Pelo programa `qrencoder` (para chaves cujo arquivo tem < 2953 caracteres) que a codifica em um código QR.
- as **subchaves** são armazenadas em um **cartão inteligente** (smartcard) que se acede por um leitor de USB com o seu próprio teclado. Em comparação ao uso de um arquivo digital, ele tem a vantagem que
  - a leitura das chaves de um cartão inteligente é muito mais difícil do que de um arquivo (armazenado num `pendrive` ou `HD`)
  - deixa menos traços:
    * Nunca revela a chave, mas apenas prova que a possua, e
    * é imune contra `keyloggers` que gravam as letras tecladas.

Figura 37: leitor de cartão inteligente

(Perfect) Forward Secrecy. Na `(Perfect) Forward Secrecy` (sigilo futuro perfeito), depois que os correspondentes trocaram as suas chaves públicas (permanentes) e estabeleceram confiança mútua,

- *antes* da correspondência cada correspondente *cria* uma chave efêmera (a *chave de sessão*) e assina-a pela chave privada (permanente) para evitar um ataque `MITM` (vide Seção 4.2).,
- *depois* da correspondência diálogo cada correspondente *apaga* a sua chave (efêmera) privada.

Desta maneira, mesmo se a correspondência foi espreitada e gravada, não pode ser decifrada posteriormente; em particular, não pode ser decifrada pela obtenção da chave privada de um correspondente.

Por exemplo, o protocolo `TLS` que cifra a comunicação em grande parte da internet, tem em versão 1.2 suporte para Perfect Forward Secrecy:

- Mais especificamente, no aperto-de-mão entre o cliente e servidor em Seção 4.4,
  - após o cliente ter recebido (e confiado em) o certificado do servidor

– o servidor e o cliente trocam uma chave pública *efêmera* que serve para cifrar a comunicação deste correspondência. Esta chave efêmera é assinada pela chave pública (perminante) do servidor.

- A criação desta chave assimétrica na Perfect Forward Secrecy torna supérflua a criação de uma chave simétrica *preliminar* pelo cliente no penúltimo passo no aperto de mão no protocolo TLS.

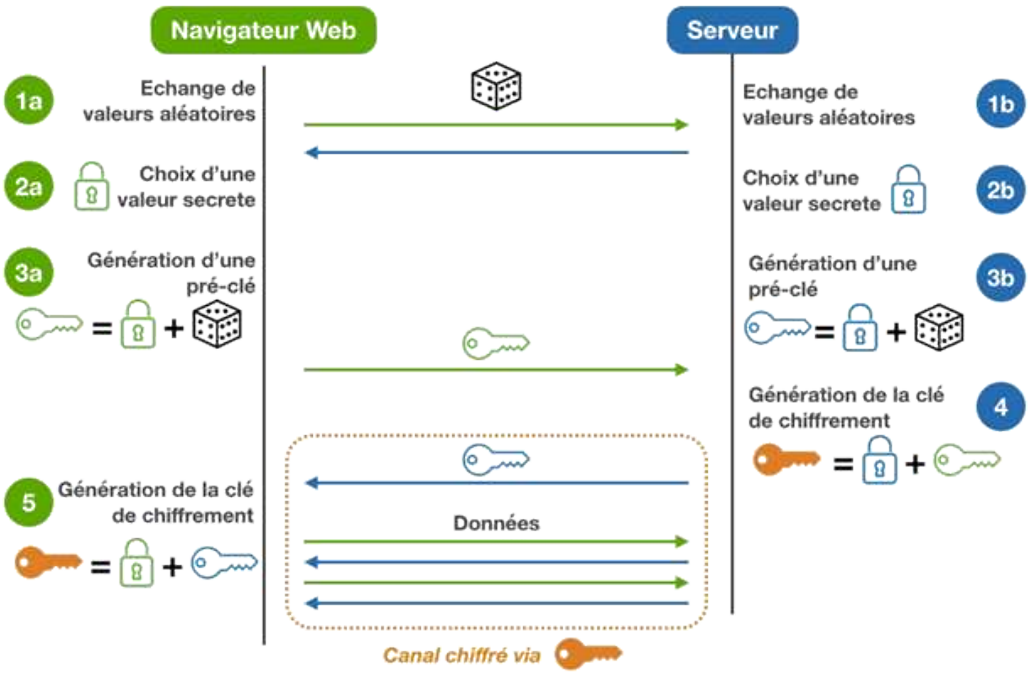


Figura 38: Perfect Forward Secrecy

Assinatura. Deve ser pensado com solenidade antes de assinar qualquer documento. O mesmo vale para assinaturas digitais.

Vantagem. Uma assinatura tem como vantagem a garantia que o dono da chave privada escrevesse o conteúdo. Na comunicação por e-mails, evita o risco que um atacante

- imita o endereço de e-mail da remetente Alice, e
- cifra com a chave do destinatário Bob.

Inconveniência. Porém, como inconveniência, caro leitor, se a comunicação contém algo que não desejas ser visto por terceiros (por exemplo, ser lido em voz alta pelo promotor público no tribunal), melhor não comproves a tua origem pela tua assinatura! Como esta mensagem pode ser espreitada, o teu correspondente mudar de ideia sobre a tua privacidade ou a sua conta ser hackeada, ...

Para dar uma analogia à realidade nossa, uma assinatura automática compara-se a gravação de toda conversa privada; esperemos que ninguém tenha acesso a ela!

Assinatura de Grupo. Observamos que Alice quer provar ao Bob que ela é a remetente, mas não a terceiros! Por isso surgiu a ideia da assinatura de grupo: uma chave efêmera para assinar é criada e compartilhada (isto é, a chave pública e *privada*) entre Alice e Bob. Desta maneira, Alice e Bob têm certeza sobre o originário, mas terceiros apenas que ele pertença ao grupo.

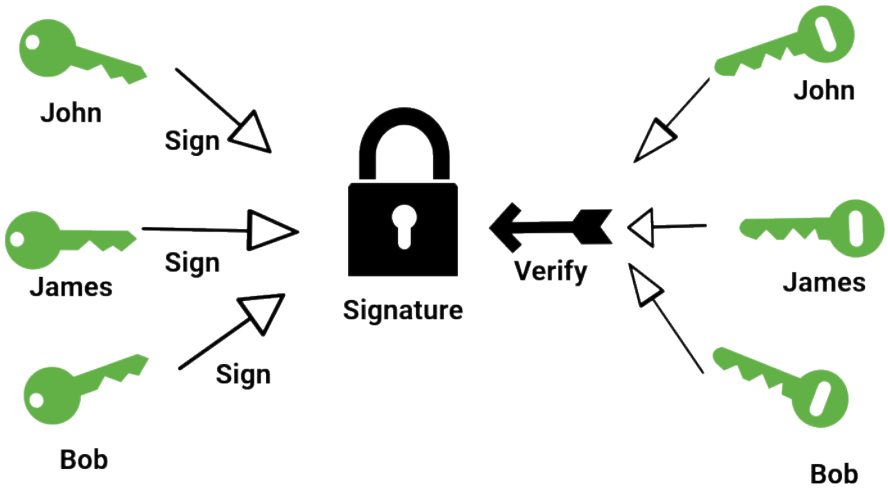


Figura 39: Assinatura de Grupo

4.2 O dono desconhecido da chave privada ou o Ataque MITM

A criptografia assimétrica permite a Alice, que conhece uma chave pública,

- enviar um texto cifrado ao dono da chave privada, ou
- confirmar que um texto provenha do dono da chave privada.

Com este conforte que a criptografia assimétrica (em contraste com a criptografia simétrica) permite à Alice, obter a chave pública impessoalmente sem encontrar o Bob, surge o problema da identidade do dono da chave pública: Considerando que a Alice

- quer enviar um texto cifrado (ou confirmar a origem de um texto supostamente pertencendo) ao Bob, e
- não obteve a chave pública pessoalmente do Bob (a utilidade da criptografia assimétrica),

como garantir à Alice a identidade da chave privada, isto é, que o dono da chave privada seja verdadeiramente o Bob?

Quer dizer, como evitar um ataque "man-in-the middle" — MIM (talvez "Maria-Bonita-no-Meio" em português nordestino) ? No MIM, o atacante se interpõe entre os correspondentes, assumindo a identidade de cada um, assim observando e interceptando suas mensagens:

Suponhamos que Maria Bonita intercepte a correspondência entre Alice e Bob.

1. Bob envia a sua chave pública à Alice. Maria Bonita intercepta-a, e envia à Alice a sua chave pública *própria* que alega Bob como o seu dono.
2. Se Alice enviar uma mensagem ao Bob, ela usa, sem ter noção, a chave pública da *Maria Bonita*!
3. Alice cifra então uma mensagem com a chave pública de *Maria Bonita* e envia-a ao Bob.
4. Maria Bonita intercepta a mensagem, decifra-a com a sua chave privada; ela pode ler a mensagem e, se quiser, modificá-la.
5. Cifra a mensagem com a chave pública do Bob.
6. Bob decifra a sua mensagem com a sua chave privada e não suspeita de nada.

Figura 40: MITM

Assim, Alice como Bob são persuadidos que usem a chave pública do outro, mas, na verdade, é a da Maria Bonita!

Em termos práticos, este problema ocorre por exemplo no protocolo ARP, o Protocolo de Resolução de Endereços (do inglês Address Resolution Protocol) de 1982 (na RFC 826) que padroniza a resolução de endereços (conversão) da camada de internet em endereços da camada de enlace. Isto é, o ARP mapeia um endereço de rede (por exemplo, um endereço IPv4) em um endereço físico como um endereço Ethernet (ou endereço MAC). (Em redes Internet Protocol Version 6 (IPv6), o ARP foi substituído pelo NDP, o Protocolo de Descoberta de Vizinhos (do inglês Neighbor Discovery Protocol).)

O ataque de ARP poisoning (= poluição de cache ARP) procede como segue:

Maria Bonita quer interceptar as mensagens da Alice ao Bob, todos os três fazendo parte na mesma rede física.

1. Maria Bonita envia um pacote `arp who-has` à Alice que contem como endereço de IP fonte o do Bob cuja identidade queremos usurpar (ARP spoofing) e o endereço físico MAC da placa de rede de Maria Bonita.
2. A Alice criará uma entrada que associa o endereço MAC de Maria Bonita ao endereço IP do Bob.
3. Assim quando Alice comunicar com o Bob no nível IP, será Maria Bonita que receberá os pacotes da Alice!

Filosofia das Soluções. Esta garantia da integridade da identidade é estabelecida por *terceiros*, isto é, identidades com chaves privadas que confirmam pelas suas assinaturas digitais que o Bob é o dono da chave privada.

O problema da identidade das chaves públicas surge outra vez: Como garantir que as identidades dos donos das chaves privadas sejam verdadeiras?

Há duas soluções:

- as *autoridades hierárquicas*, e

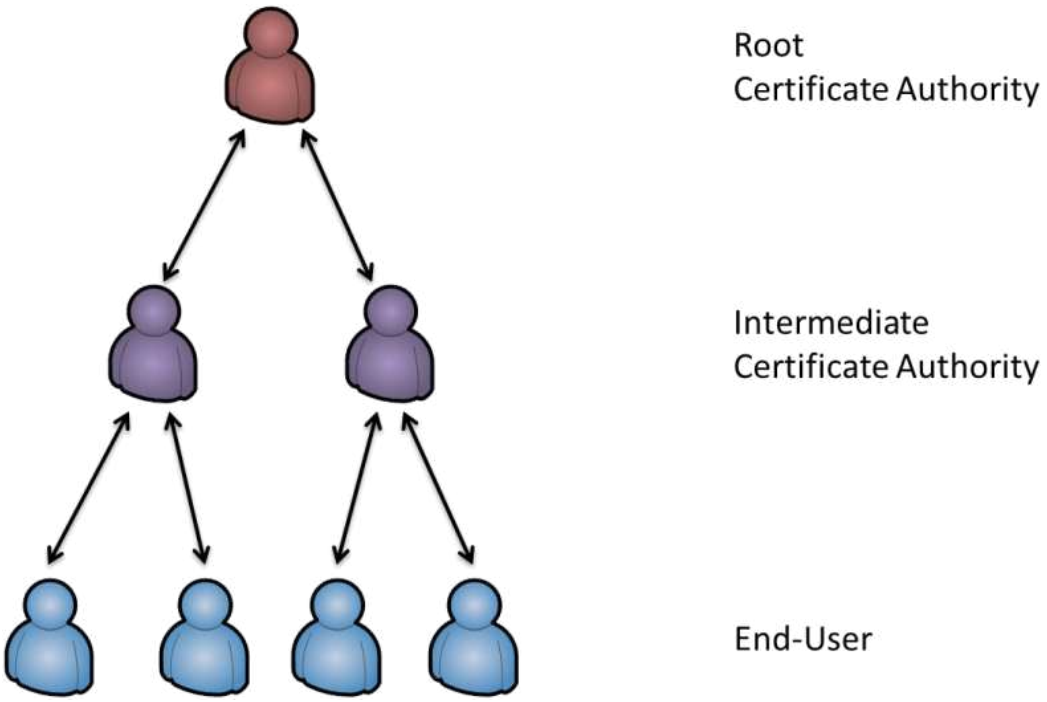


Figura 41: autoridades hierárquicas

- a *teia de confiança.*

Autoridades Hierárquicas. Nas autoridades hierárquicas, os donos de chaves privadas distinguem-se por níveis hierárquicos. No nível o mais alto jazem as *autoridades radicais* nas quais se confia incondicionalmente. Por exemplo,

- `VeriSign`, `GeoTrust`, `Comodo`, ... são grandes empresas certificadoras estado-unidenses;

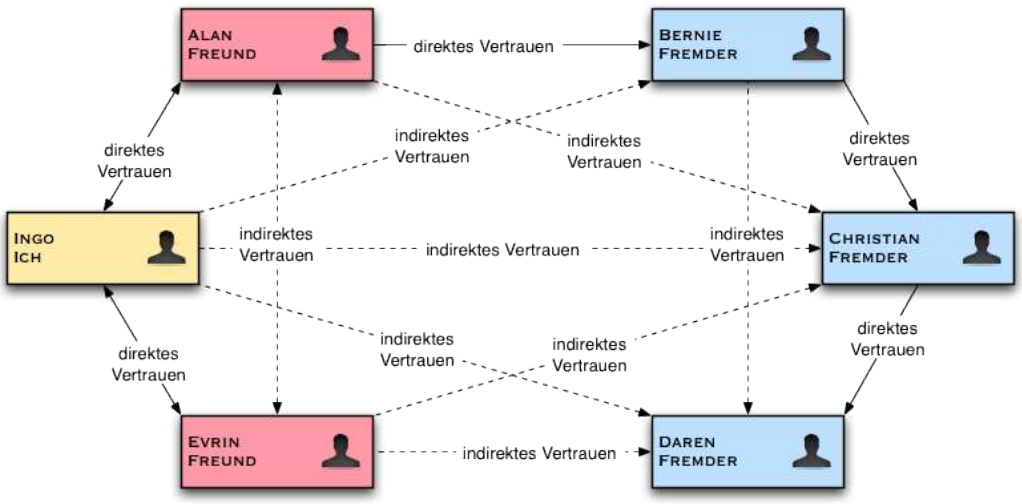


Figura 42: teia de confiança

- recentemente surgiu a autoridade (intermediaria estado-unidense) `Let's encrypt` sem fins lucrativos;
- uma olhada na pasta `/etc/ssl/certs` na distribuição de Linux openSUSE revela que existe

  – uma autoridade alemã (`T-TeleSec` da Deutsche Telekom AG, a antiga operadora de telecomunicação nacional),
  – uma autoridade holandesa (`Staat der Nederlanden`),
  – três espanholas (`Firmaprofesional`, `ACCVRAIZ1` — Agencia de Tecnología y Certificación Electrónica, e `ACC RAIZ FNMT` — Fábrica Nacional de Moneda y Timbre),

  entretanto a maioria é estado-unidense; (Quanto aos navegadores, os certificados do `Firefox` jazem na base-de-dados `cert8.db` na pasta do perfil do usuário [por exemplo, `%APPDATA%/Mozilla/Firefox/<perfil>/cert8.db` em `Microsoft Windows` > 7], mas precisa compila o programa `certutil` para lê-lo.)
- no Brasil o `ICP`, Instituto Nacional de Tecnologia da Informação da Casa Civil da Presidência da República, cuida da infraestrutura de chaves públicas. Em particular, credencia as autoridades radicais: Na lista destas autoridades, figuram, entre outras,

  – o `SERPRO` - Serviço Federal de Processamento de Dados, primeira

Autoridade Certificadora credenciada pela ICP-Brasil
  - Caixa Econômica Federal, a única instituição financeira credenciada como Autoridade Certificadora da ICP-Brasil
  - AC Digitalsign, uma empresa portuguesa
  - Ministério das Relações Exteriores, responsável exclusivamente pelo certificado digital do novo passaporte brasileiro aderente ao Public Key Directory — PKD da Organização da Aviação Civil Internacional — ICAO, agência especializada das Nações Unidas.
  - SERASA Experian, que fornece certificados digitais para quase todos os grupos financeiros participantes do Sistema de Pagamentos Brasileiro — SPB.

Consta, porém, que nenhum certificado destas autoridades

- figura na pasta /etc/ssl/certs, ou
- é aceite pelo navegador Chrome ou Firefox.

Teia de Confiança. Na teia de confiança, os donos de chaves privadas não se distinguem de um ao outro.

A ausência de autoridades radicais, entidades incondicionalmente confiáveis, é compensada pela

- confiança estabelecida por
  - ter obtido a chave pública pessoalmente (por exemplo, em *key-sign parties*, encontros em que os participantes trocam e assinam as suas chaves públicas mutuamente), ou
  - por
    * ter obtido a chave por um canal (site, e-mail, ...) e
    * ter comunicado a sua soma de verificação por outro canal (telefone, SMS, mensageiro instantâneo, ...) e
- a qual se transfere de um ao outro, isto é, a confiança é transitiva: se Alice confia em Bob, e Bob confia em Carlos, então Alice confia em Carlos.

Padronização das Filosofias na Internet. Na internet,

- o sistema de confiança por autoridades hierárquicas foi padronizado pelo esquema `X.509`, com o seu maior uso de cifrar a comunicação entre o usuário e um site (frequentemente comercial), e
- o pela teia de confiança pelo esquema `OpenPGP` (e principalmente implementado pelo programa `GnuPG`), com o seu maior usa de cifrar os e-mails entre dois usuários. Este esquema radicalmente rejeita qualquer hierarquia: o usuário pode publicar uma chave pública com um endereço de e-mail em um servidor de chaves públicas (por exemplo, em pgp.mit.edu) sem sequer confirmar (por um e-mail de ativação) que tem acesso à conta deste endereço de e-mail.

Comparamos as vantagens e inconveniências entre as duas abordagens:

4.3 `X.509`

A assinatura de um certificado `X.509` é a cifração pela chave privada do hash da concatenação

$$[\mathrm{V}, \mathrm{SN}, \mathrm{AI}, \mathrm{CA}, \mathrm{TA}, \mathrm{A}, \mathrm{KA}]$$

de

- `V` = versão `X.509`,
- `SN` = número de série do certificado,
- `AI` = número identificador do algoritmo,
- `CA` = nome da autoridade certificadora,
- `TA` = tempo de intervalo de validade do certificado,
- `A` = nome do sujeito, e
- `KA` = chave pública do sujeito.

O sistema de *autoridades hierárquicas* foi criado para estabelecer a confiança através de máquinas e tem como grande vantagem

- o conforto que a troca de chave poder ser automatizada.

Porém, confiança é uma questão humana, e tem como calcanhar-de-aquiles a confiança (absoluta) na autoridade (radical):

- a confiança absoluta que a chave **pública** pertença à autoridade;

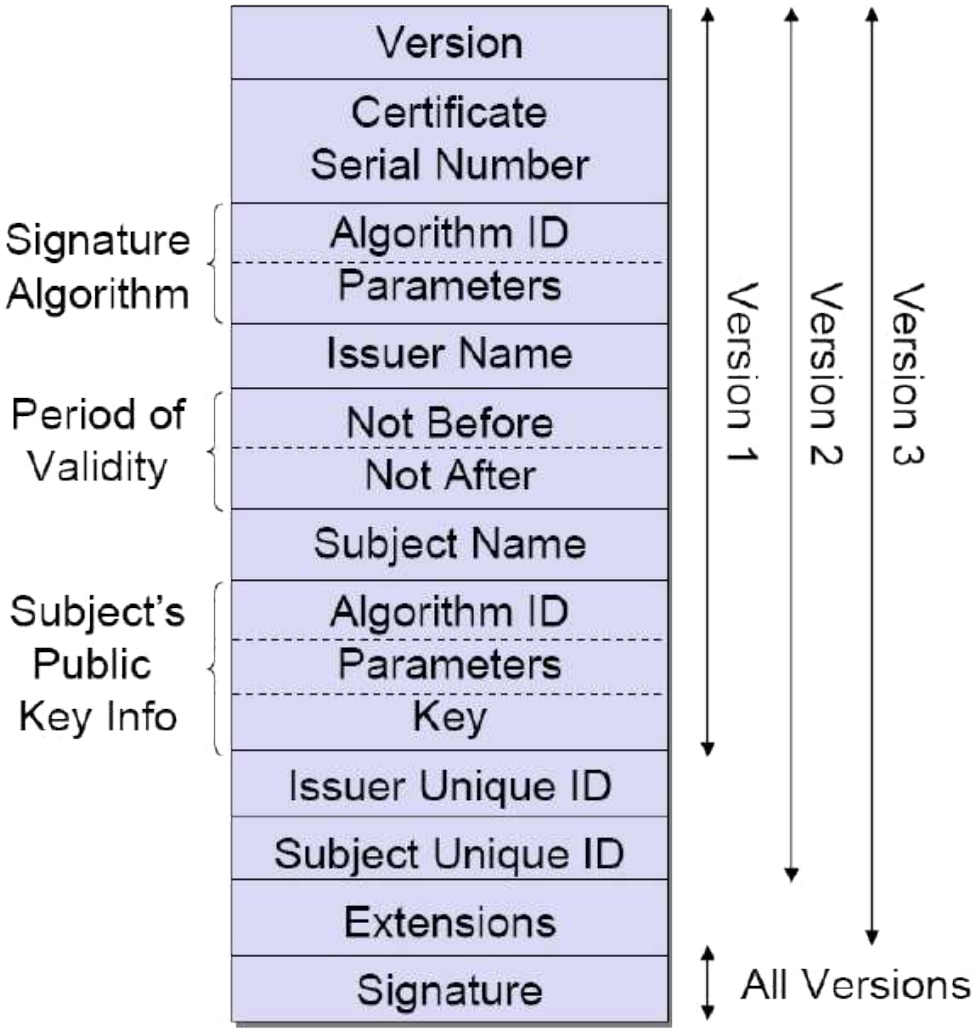


Figura 43: Estrutura de um certificado `X.509`

- a confiança absoluta que a chave **privada** da autoridade não seja comprometida;
- a confiança que a autoridade não **abuse** do seu poder. Por exemplo, cobrar indevidamente caro; (Esta observação levou a recente criação da autoridade livre Let's Encrypt que
  - fornece certificados gratuitos (e tem um orçamento de 3 Milhões $;

  - para comparar, VeriSign cobra por cada certificado 399$ por um ano),

- a confiança que a autoridade trabalhe **seriamente**, por exemplo, na verificação da identidade do terceiro pela autoridade. A este fim, cada certificadora radical é sujeita a pareceres de auditorias periódicos (que leva a questão recursiva se o mesmo vale para quem faz as auditorias?!). Além disto, o nível de segurança reflete-se pela forma do cadeado na barra-de-endereço do navegador:

  - Na emissão de um certificado comum, domain certificate, a verificação é completamente automatizada; nenhuma verificação fora de linha é feita. Basta ter acesso ao domínio para obter o certificado.
  - Na emissão de um extended certificate a verificação do proprietário do site é feita pessoalmente.

  Por isso, para ter certeza que o site pertença a quem pretende (por exemplo, evitar a confusão entre bancobrasil.com.br e bancobrazil.com.br), é importante verificar na barra-de-endereço que o cadeado indique um extended certificate; por exemplo, os navegadores Firefox e Chrome indicam-no pela cor verde do nome da entidade.

  Pelo cadeado comum que corresponde a um domain certificate, o usuário apenas tem certeza que comunique com o dono do domínio, mas não que ele pertence à empresa ou organização que o site aparenta representar. Quer dizer, o cadeado (ou certificado) comum,

  - sim, comprova que o dono da chave privada seja o dono do servidor neste endereço, mas
  - não que o servidor neste endereço pertença à pessoa (jurídica) que aparenta.

  Isto é, mesmo se o cadeado do certificado comum desperta confiança, a sua emissão automatizada

  - evita um ataque MITM por DNS Cache poisoning, em que o endereço do nome (por exemplo, bancobrasil.com.br) é resolvido ao endereço numérico IP de outro servidor, contudo
  - permite um ataque MITM pela confusão do usuário entre o endereço e a pessoa (jurídica).

Figura 44: Certificado Avançado

Figura 45: Certificado Comum

- o **respeito** à autoridade, isto é, o receptor do certificado tem de saber se pode confiar na chave pública da autoridade ou não.

  – não existe qualquer autoridade radical brasileira aceite pelos navegadores.
  – tem umas autoridades espanholas (como AC Raíz, aceite por Linux, pelo Chrome, porém, não pelo Firefox!),
  – por exemplo, uma alemã, uma holandesa, mas
  – quase todas são estado-unidenses.

  Por exemplo, nem o Chrome, nem o Firefox confiam na autoridade radical brasileira principal SERPRO; isto é, não inclui o seu certificado (= chave pública e auto-assinatura). Para o usuário, como o certificado auto-assinado

  é desconhecido, aparece o seguinte alerta:

  Por isso, por exemplo,

  – a Caixa Económica Federal,
    * em vez de usar um certificado assinado pela própria autoridade radical (ou, por exemplo, pelo SERPRO),
    * usa um certificado assinado pela autoridade radical estado-unidense COMODO RSA,
  – o DETRAN em Alagoas,
    * em vez de usar um certificado assinado pela autoridade radical da Receita Federal do Brasil (ou, por exemplo, pelo SERPRO),
    * usa um certificado assinado pela autoridade intermediaria gratuita estado-unidense Let's Encrypt.

4.4 O aperto de mão pelo X.509

Relatamos o maior uso do padrão X.509 na internet, o TLS Handshake:

- o *aperto de mão* na Transport Layer Security, Segurança da Camada de Transporte (antigamente Secure Sockets Layer — SSL, Protocolo de Camada Segura de Soquetes), o protocolo que cifra
- a comunicação pelo protocolo (de controle) de transmissão Transmission Control Protocol — TCP.

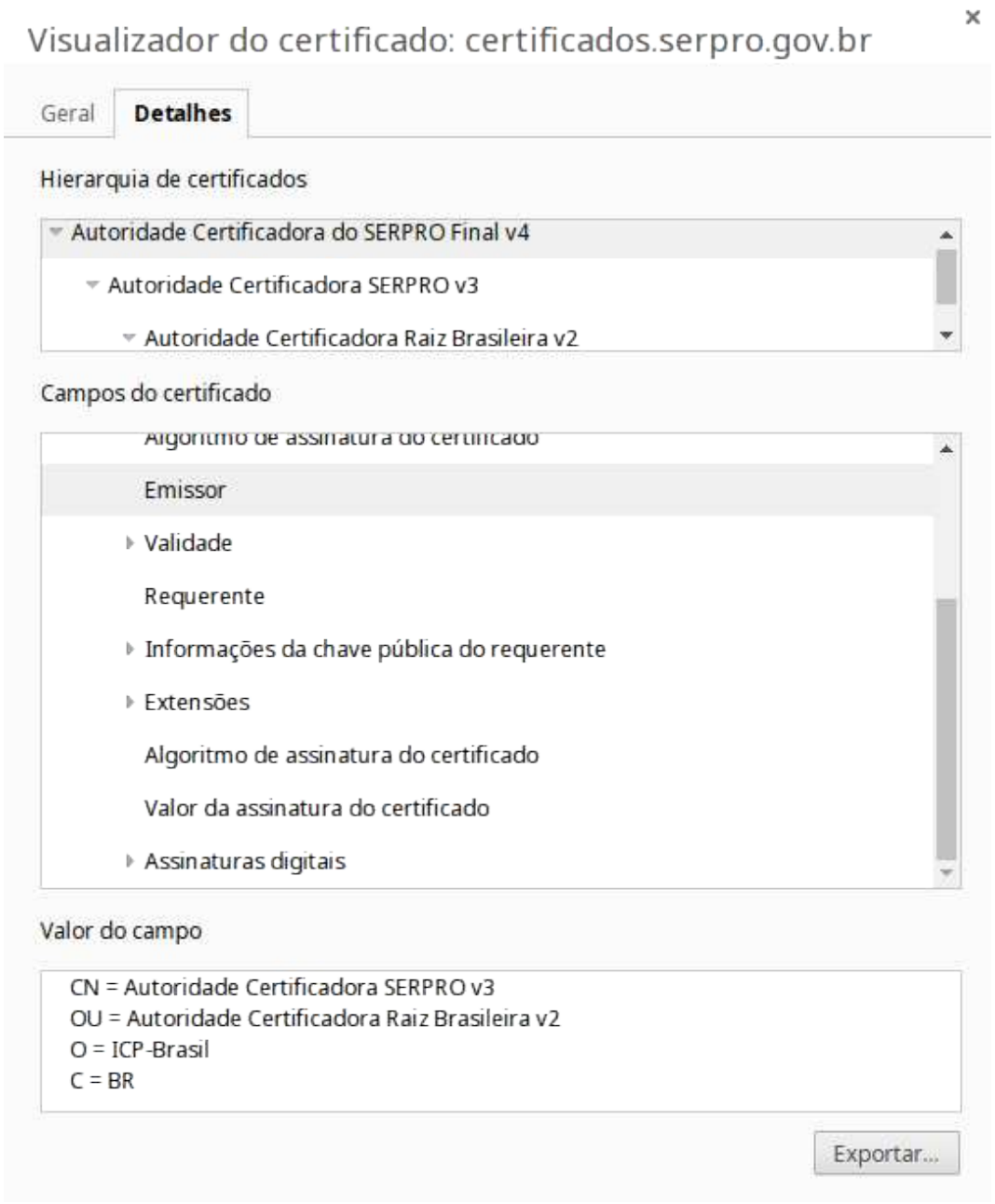


Figura 46: Certificado auto-assinado pelo SERPRO

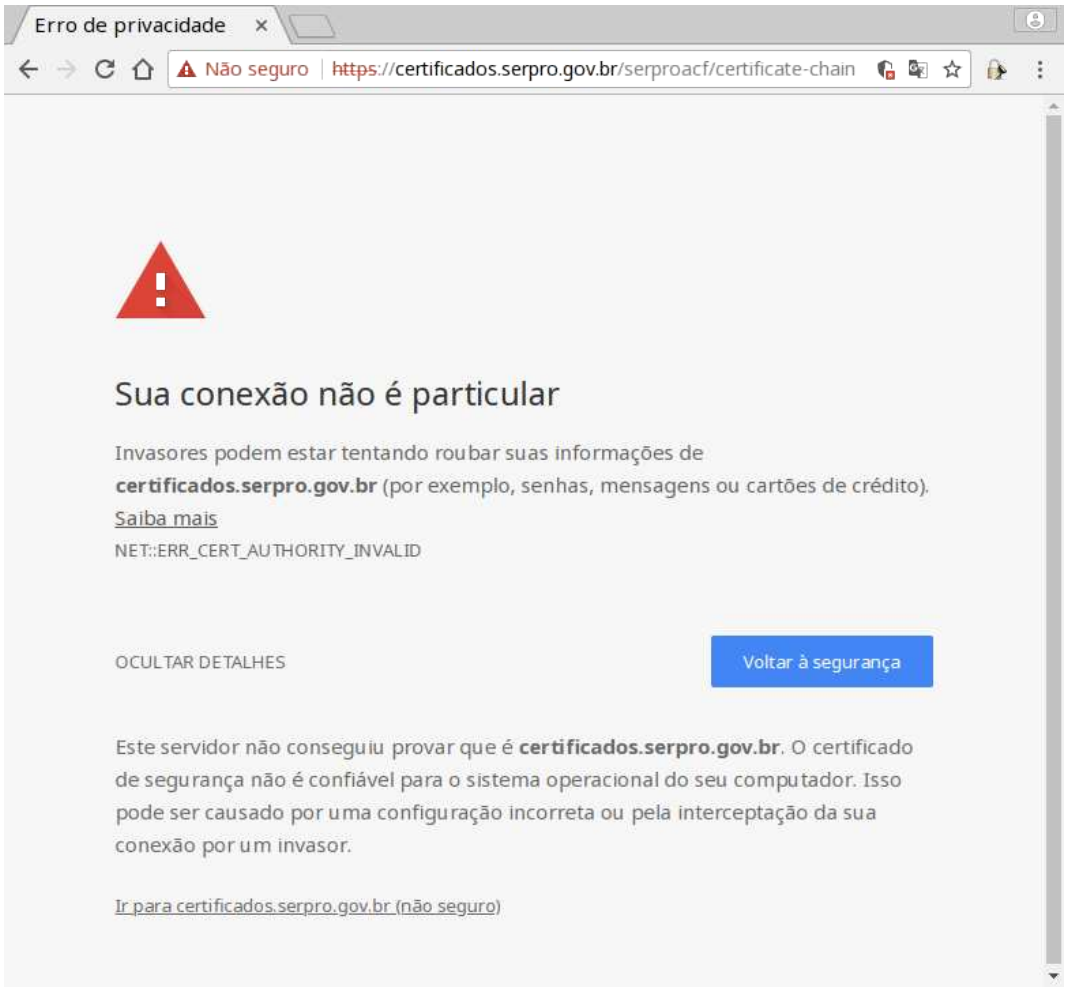


Figura 47: Alerta ao acessar um site certificado pelo SERPRO

A referência definitiva é a especificação RFC 5246. Uma ótima visão global é dada pela Conexão Ilustrada de TLS 1.3 com cada baite explicado.

São os primeiros passos entre um cliente e o servidor, por exemplo, um site de comércio eletrônico, para estabelecer uma conexão cifrada (por exemplo, para receber os dados do cartão de crédito do cliente).

Opcionalmente,

- além da autentificação obrigatória do servidor (por um certificado), o cliente se autentifica da mesma maneira (por um certificado);

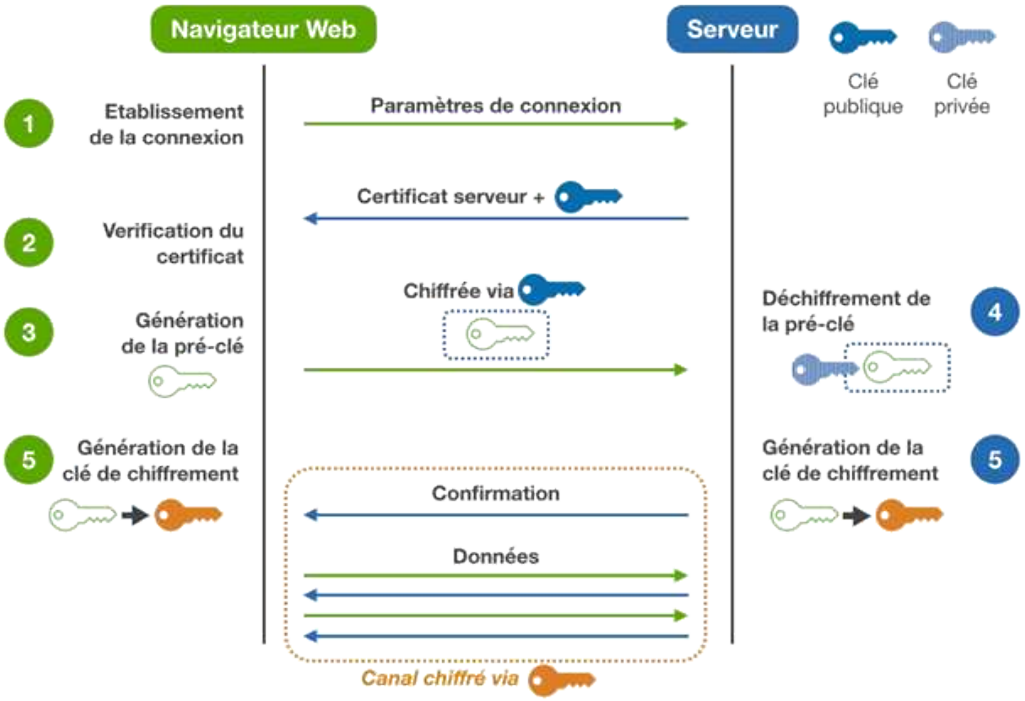


Figura 48: Aperto de Mão no TLS

- o servidor e cliente trocam uma chave efêmera *assimétrica* para mandar a chave simétrica;
  - enquanto a chave assimétrica permanente do servidor quase sempre usa o algoritmo `RSA`,
  - a chave assimétrica efêmera quase sempre usa um algoritmo que se baseia na troca de chaves de Diffie-Hellman (por exemplo, `ECC` ou `ElGamal`).

1. O "Alô" entre o cliente e o servidor, em que o cliente propõe, e o servidor escolhe, um *pacote criptográfico*; isto é, o conjunto dos algoritmos criptográficos,
   - para se autentificar,
     - uma soma de verificação criptográfica (`MD5`, `SHA`, ...), e
     - um algoritmo criptográfico assimétrico (`RSA`, ...),

- para trocar uma chave simétrica, um algoritmo criptográfico assimétrico (RSA, ECC, ...),
- para cifrar a comunicação, um algoritmo simétrico (AES, Camellia, RS4, ...). Frequentemente, é escolhido não o algoritmo simétrico mais seguro, mas, sim, o algoritmo mais econômico pelo servidor.

Por exemplo, o pacote criptográfico TLS_RSA_WITH_3DES_EDE_CBC_SHA (codificado 0x00 0x0a) usa

- RSA para autentificar e trocar as chaves,
- 3DES em modo CBC para cifrar a conexão, e
- SHA como hash criptográfico.

Além disto, cada um,

- o cliente,
- e o servidor

criam um nonce, isto é, um número de uso único, que contém

- 4 baites para marcar o horário, e
- 20 baites aleatórias,

que evita um ataque de repetição (replay attack); isto é, a reutilização das autentificações em outras sessões.

2. O servidor identifica e autentifica-se pelo

   - seu *certificado* X.509:

   Este contém (principalmente):

   - o endereço do servidor e a sua chave pública (usando o algoritmo assimétrico inicialmente combinado para a troca da chave simétrica)
   - para autentificar, um nome de uma autoridade (radical, por exemplo, VeriSign), e a sua assinatura digital (usando o hash criptográfico e o algoritmo assimétrico já combinados); isto é, a cifração pela chave privada da autoridade (de uma soma de verificação criptográfica)
     - do endereço do servidor, e
     - da chave pública do servidor.

   Na imagem, temos o servidor www.detran.al.gov

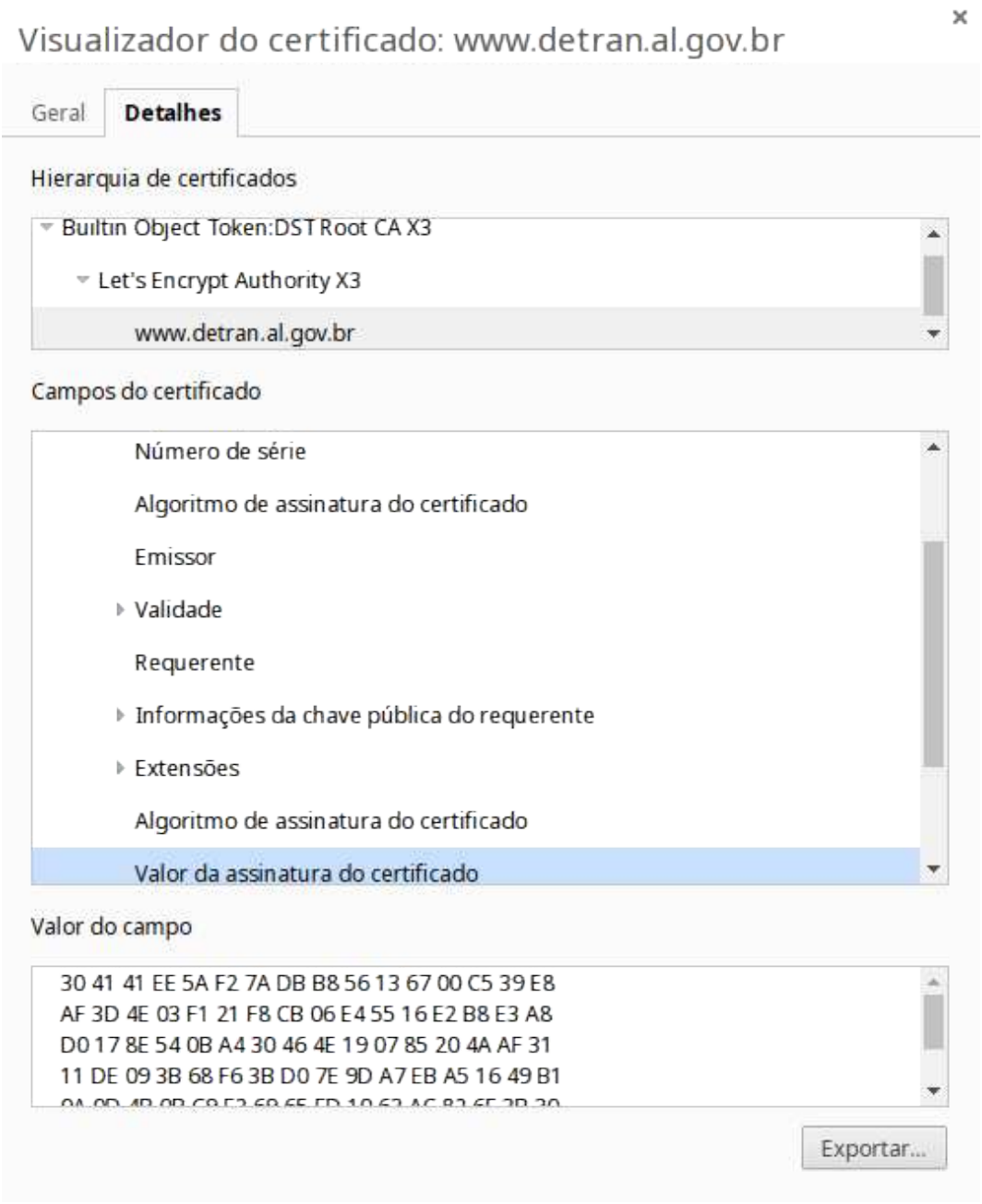


Figura 49: certificado `X.509`

- cujo certificado é assinado pela autoridade intermediária `Let's Encrypt Authority X3`
- cujo certificado é assinado pela autoridade radical `DST Root CA X3`
- cujo certificado é auto-assinado (isto é, assinado por ela mesma).

O cliente procura a chave pública da autoridade (radical) indicada no certificado (que é usualmente inclusa no navegador), e usa-a para decifrar esta assinatura digital. Se o resultado é a soma de verificação esperada (isto é, do endereço do servidor e da sua chave pública), então

- a assinatura digital provém verdadeiramente da autoridade indicada, e
- a autoridade confia neste servidor.

Como o cliente (ou, mais exatamente, o seu navegador) confia incondicionalmente nas autoridades radicais, a este ponto ele tem certeza que a chave pública pertença verdadeiramente ao servidor visado.

(Opcionalmente, neste ponto também o cliente se autentifica por um certificado.)

3. O cliente

   - cria um *pré-segredo* (`pré-master-secret`), um número (pseudo-)aleatório de 48 baites,
   - cifra-o pela chave pública (usando o algoritmo *assimétrico* combinado), e
   - manda-o ao servidor.

   O servidor

   - decifra o pré-segredo pela sua chave privada.

4. O cliente e o servidor calculam o *segredo* (`master secret`), um número de 48 baites, por uma função `PRF`, `master_secret = PRF(pre_master_secret, ClientHello.random + ServerHello.random)` que usa como variáveis

   - o pré-segredo, e
   - os `nonces`, comunicados no Alô,
     - do cliente, e

– do servidor.

O cliente e o servidor calculam quatro chaves simétricas (para o algoritmo combinado inicialmente, por exemplo, se é `AES`, cada uma de 16 baites) a partir do segredo; nomeadamente:

- `client_write_MAC_secret`,
- `server_write_MAC_secret`,
- `client_write_key`, e
- `server_write_key`.

Entre elas,

- as primeiras duas servem para verificar a integridade dos dados
- as últimas duas servem para cifrar os dados.

Que cada lado tem uma chave diferente se deve à melhor prática (best practice) de usar uma chave diferente para cada uso diferente.

4.5 OpenPGP

Vamos discutir:

1. as vantagens e inconveniências da teia de confiança:
2. as vantagens e inconveniências (das implementações) de `OpenPGP`:

Teia de Confiança. O sistema da *teia de confiança* modela o estabelecimento da confiança humana e tem como grande vantagem que

- é um sistema par-a-par; é independente de qualquer autoridade ou terceiro particular;

e como grande inconveniência que

- precisa de manutenção pessoal.

Além disso,

- expõe informação como

  - o grafo social de cada participante,
  - a hora e o lugar da interação entre os confidenciados.

- não escala bem; isto é, a quantidade de armazenagem para a teia de confiança do mundo inteiro seria imensa. Para o nível de autentificação que fornece, não faz sentido nestas escalas; infelizmente, confiamos com convicção na chave da outra pessoa quando confiamos em todos os terceiros que testemunham a sua autenticidade, isto é, quando conhecemo-los pessoalmente.
- quão séria é a confiança testemunhada pelos terceiros na teia? Por exemplo, as ditas `key-sign parties` comumente juntam desconhecidos que assinam as suas chaves um ao outro. Isto é o contrário de que confiança significa: Só deve ser passada quando conhecemos bem o dono da chave que assinamos!
- A teia é tão segura quanto a sua malha menos segura; isto é, uma malha da teia (facilmente) comprometida põe todos os seus fios em risco: Uma chave comprometida pode ser usada para assinar qualquer outra chave na teia de confiança.

  Por exemplo, imagina que Alice quer cifrar uma mensagem a Bob. Seja a chave de Carlos comprometida pelo `man-in-the-middle`.

  - O `man-in-the-middle`
    * cria uma chave no nome de Bob;
    * assina esta chave pela chave de Carlos;
    * manda esta chave ao remetente Alice;
  - Alice,
    * se confia em Carlos pela teia de confiança,
    * então confia em Bob (isto é, na chave do man-in-the-middle!).

  Mesmo se esta fraude for descoberta por outros (a ideia da teia de confiança é que é impossível enganar todo mundo), certo estrago já foi feito.

Na prática, em vez da teia de confiança, é mais viável estabelecer a confiança por outro canal impessoal (pela troca das impressões digitais das chaves públicas), por exemplo,

- por SMS,
- um mensageiro como Telegram ou WhatsApp,
- por telefone, ...

OpenPGP. Apesar da absurdidade (visto que confiança é uma questão humana e pessoal) da confiança automática do usuário prestada às certificadoras radicais (sobretudo empresas),

- o esquema X.509 é, pelo seu conforto para o usuário, ubíquo no comércio eletrônico,
- pouquíssimos usuários usam o esquema OpenPGP para comunicar por e-mail.

Avaliam principalmente o esforço para a instalação do programa e a manutenção das chaves (a qual inerentemente necessita da estimativa do usuário para a confiança nas chaves usadas) desproporcionalmente grande para o benefício (da maior privacidade e segurança) obtido. Infelizmente, justamente porque tão poucas pessoas usam OpenPGP, a usabilidade dos programas dedicados melhorou pouco nos últimos anos.

Por isso:

- o gerenciamento das chaves deve ser facilitado:
  - como citado, atualmente um servidor de chaves públicas (por exemplo, pgp.mit.edu) aceita qualquer chave sem sequer confirmar (por um e-mail de ativação) que o remetente tem acesso à conta deste endereço de e-mail;
    * Seria preferível que fizesse esta confirmação, isto é, uma autoridade central para as chaves públicas;
    * pode até ser pensado em uma autoridade central que gera as chaves privadas (para sincronizá-las entre os aparelhos do usuário). (Porém, qualquer solução centralizada que depende de um terceiro, comumente uma empresa, e esta pode ser forçada pelo governo a passar a armazenagem das chaves privadas.)
  - Surgiram recentemente várias soluções de Software que automatizam a troca de chaves no protocolo OpenPGP e que serão apresentados em Seção 4.6. Mesmo se usuário perde (em parte) o controle sobre a confiança, a ganha de conforto ajuda a divulgar este protocolo para

leigos. O tipo de segurança que oferecem chama-se Opportunistic Security: enquanto ninguém se interesse, é seguro; caso contrário, é vulnerável ao ataque pelo man-in-the-middle.

- O protocolo OpenPGP tem as suas insuficiências concepcionais:
  - a falta de Perfect Forward Secrecy (PFS); mesmo se o usuário típico quer guardar os e-mail escritos e lidos e prefere chifrá-los no seu computador pela mesma chave, é mais seguro cifrar o e-mail para o transporte (através do servidor do remetente e destinatário) por uma chave efêmera. (Consta-se, porém, que muitos usuários guardam localmente copias em texto claro das suas mensagens, o que compromete o esforço criptográfico no transporte.)
  - a vulnerabilidade da teia de confiança; (Porém, o estabelicmento de confiança por ela é opcional; é perfeitamente possível estabelecê-la por outros canais.)
  - uma assinatura não somente comprova ao destinatário a origem da mensagem, mas também a terceiros (que podem usar este comprovante contra o remetente; a assinatura de grupo foi concebida contra este abuso);
  - porque OpenPGP usa o protocolo de e-mail que não cifra o conteúdo, omite-se facilmente a cifração por mal uso do cliente de e-mail. (Pode ser pensado em usar um cliente de e-mail exclusivamente dedicado a comunicação cifrada.)

Recentemente, surgiu o programa opmsg (como alternativa a GPG que será apresentado em Seção 4.6) que implementa (muitos métodos d') o protocolo Off the Record — OTR (como alternativa a OpenPGP) que oferece:

- Perfect Forward Secrecy: o comprometimento das chaves privadas não compromete as conversas cifradas anteriores (porque uma chave efêmera é criada para o envio de cada mensagem e apagada após o recebimento)
- Autenticidade: garantia que o correspondente seja a pessoa certa (porque toda mensagem é assinada pelo remetente)
- Contestabilidade: impossibilidade de comprovar a origem das assinaturas após a correspondência (pela *assinatura de grupo* — ring signature, que compartilha a chave secreta para assinar entre todos os correspondentes; por este recurso surge a necessidade de usar uma chave para cada correspondente!)

Além disso

- recomenda (criar e) usar uma chave (= persona na terminologia de opmsg) para cada correspondente,
- insiste em verificar a impressão digital da chave pública por outro canal (e não implementa a teia de confiança)
- NÃO cifra as chaves privadas por uma senha (simétrica).

4.6 Exemplos de Programas OpenPGP

Apresentamos uns programas que usam a teia de confiança (o protocolo OpenPGP), como

- o programa de linha-de-comando GPG para criar chaves e (de)cifrar e assinar/autenticar por elas,
- a extensão Enigmail para o cliente de e-mail livre Thunderbird, que recentemente automatiza este processo pelo suporte a AutoCrypt,
- o programa Mailvelope para cifrar os e-mails nos sites como gmail.com e Hotmail.com, entre outros, e
- o mensageiro Delta-Chat para Android que tem pelo uso do protocolo de e-mail a vantagem de não depender de um servidor só da empresa fornecedora, como o WhatsApp.

GPG. O programa GnuPG é um programa de *linha-de-comando* e pouco acolhedor para principiantes. Serve para a funcionalidade criptográfica de muitos programas criptográficos com *interface gráfica*.

Objetivo do GPG. Oferecer ao grande público *aberta e gratuitamente* métodos criptográficos para cifrar dados eletrônicos confidenciais.

Funções Principais. Um programa de linha-de-comando para

- cifrar e decifrar de dados (por exemplo, e-mails), e
- criar e verificar assinaturas digitais (para garantir autenticidade e integridade dos dados).

Aceitação.

- Pré-instalado na maioria das distribuições de Linux, e
- disponível sob Mac OS e Microsoft Windows

História.

1997. O desenvolvimento do programa Gnu Privacy Guard (= GPG) pelo alemão *Werner Koch* começou (e até hoje não cessou) para ter uma alternativa livre ao programa de criptografia de e-mail comercial Pretty Good Privacy (= PGP) por Phil Zimmermann. Até hoje, é o desenvolvedor principal e depende de doações como única fonte de renda. No ínico de 2015, os seus recuros estavam acabando e pediu ajuda financeira, a qual recebeu amplamente.

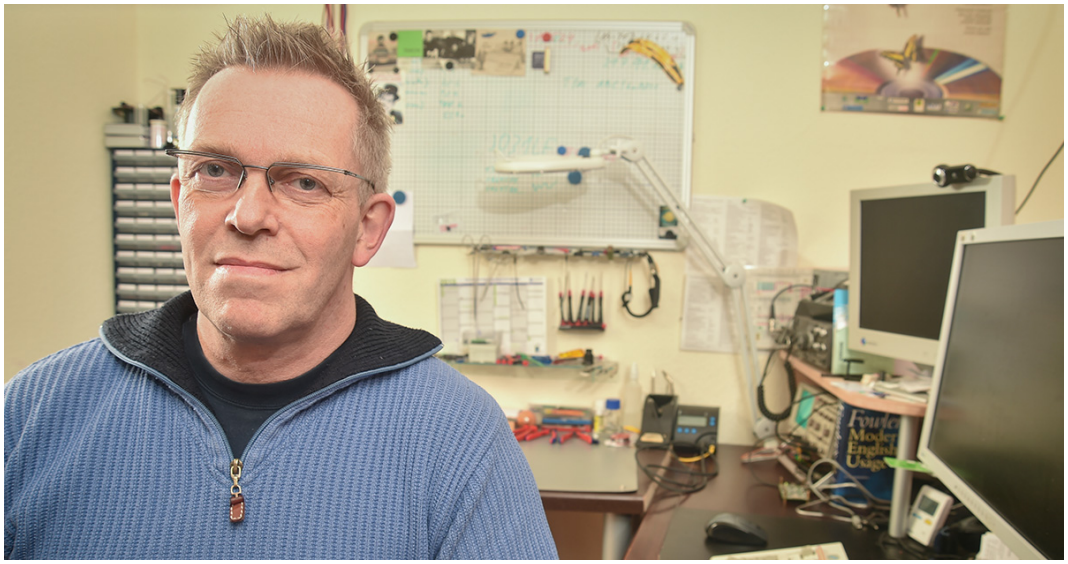


Figura 50: O criador e desenvolvedor principal Werner Koch

1999. A versão 1.0.0 foi anunciada.

2000. O Ministério Federal Alemão de Economia e Tecnologia patrocinou a portabilidade para o Microsoft Windows.

2006. A versão 2.0 foi anunciada e trouxe mudanças significativas na arquitetura do programa.

```
--- ~ » gpg2 --full-generate-key
Por favor selecione o tipo de chave desejado:
   (1) RSA and RSA (default)
   (2) DSA and Elgamal
   (3) DSA (apenas assinatura)
   (4) RSA (apenas assinatura)
Opção? 1
RSA keys may be between 1024 and 4096 bits long.
What keysize do you want? (2048)
O tamanho de chave pedido é 2048 bits
Por favor especifique por quanto tempo a chave deve ser válida.
         0 = chave não expira
      <n>  = chave expira em n dias
      <n>w = chave expira em n semanas
      <n>m = chave expira em n meses
      <n>y = chave expira em n anos
A chave é valida por? (0) 1y
Key expires at Ter 02 Abr 2019 13:11:59 -03
Is this correct? (y/N) y

GnuPG needs to construct a user ID to identify your key.

Nome completo: Enno
O nome deve ter pelo menos 5 caracteres
Nome completo: Enno Nagel
Endereço de correio eletrónico: biz.nagel@t-online.de
Comentário:
Você selecionou este identificador de utilizador:
    "Enno Nagel <biz.nagel@t-online.de>"

Mudar (N)ome, (C)omentário, (E)ndereço ou (O)k/(S)air? O
Precisamos gerar muitos bytes aleatórios. É uma boa ideia realizar outra
actividade (escrever no teclado, mover o rato, usar os discos) durante a
geração dos números primos; isso dá ao gerador de números aleatórios
uma hipótese maior de ganhar entropia suficiente.
_
```

Figura 51: Criação de um par de chaves no GPG na linha-de-comando

Funcionamento. Cria um par de chaves, uma pública e a outra privada, escolhendo

- o algoritmo (por exemplo, RSA),
- o tamanho (por exemplo, 2048 bites),
- a validade (por exemplo, um ano),
- uma senha para a chave privada, e
- a identidade: o nome e endereço de e-mail do dono.

A chave *pública* é destinada à divulgação. Ela serve

- a cifrar e
- a verificar assinaturas.

A chave *privada* é guardada e protegida por uma senha. Ela serve

- a decifrar e

- a assinar.

`Enigmail`.

- O programa `Enigmail` é uma extensão para o programa de e-mail gráfico `Thunderbird` que adiciona a este a função de

  – cifrar,
  – decifrar,
  – assinar e
  – verificar assinaturas de e-mails,

  pelo `GnuPG`, assim que o usuário pode aceder a estas funções por botões no próprio `Thunderbird`.

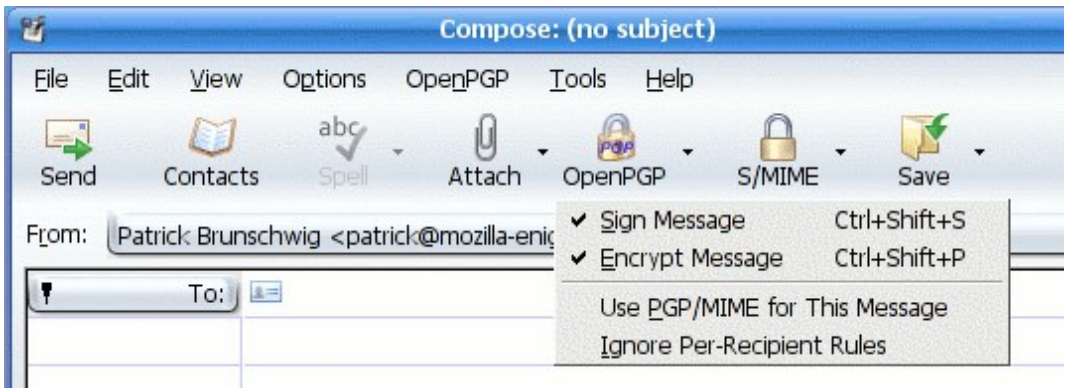


Figura 52: Enigmail

`Mailvelope`. O `Mailvelope` é uma extensão para os navegadores `Firefox` e `Chrome`, desenvolvido por uma empresa alemã, que adiciona recursos de criptografia e descriptografia à interface de provedores comuns de webmail como:

- `Gmail`
- `Hotmail.com`
- `Yahoo!`

Por exemplo, cifra e decifra mensagens (usando o padrão OpenPGP), arquivos em seu disco rígido e envie anexos de e-mail cifrados.

Mailvelope é de código aberto e com base em OpenPGP.js (https://openpgpjs.org), uma biblioteca OpenPGP para JavaScript.

Uma introdução é dada em https://casalribeiro.com/pt/email-encryption-mailvelope/ .

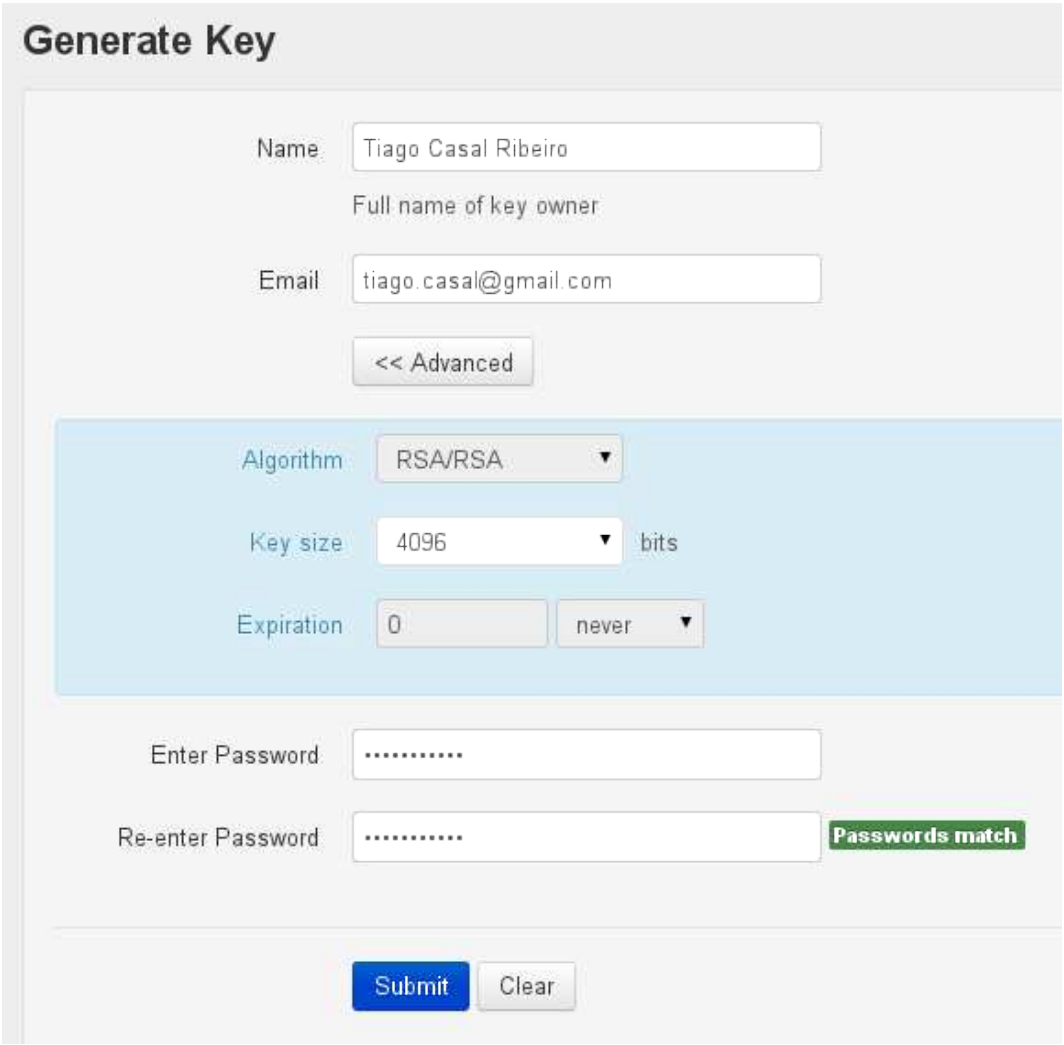


Figura 53: Criar uma chave pública pelo Mailvelope

É muito confortável, porém, este conforto vem ao detrimento da segurança; consta-se que

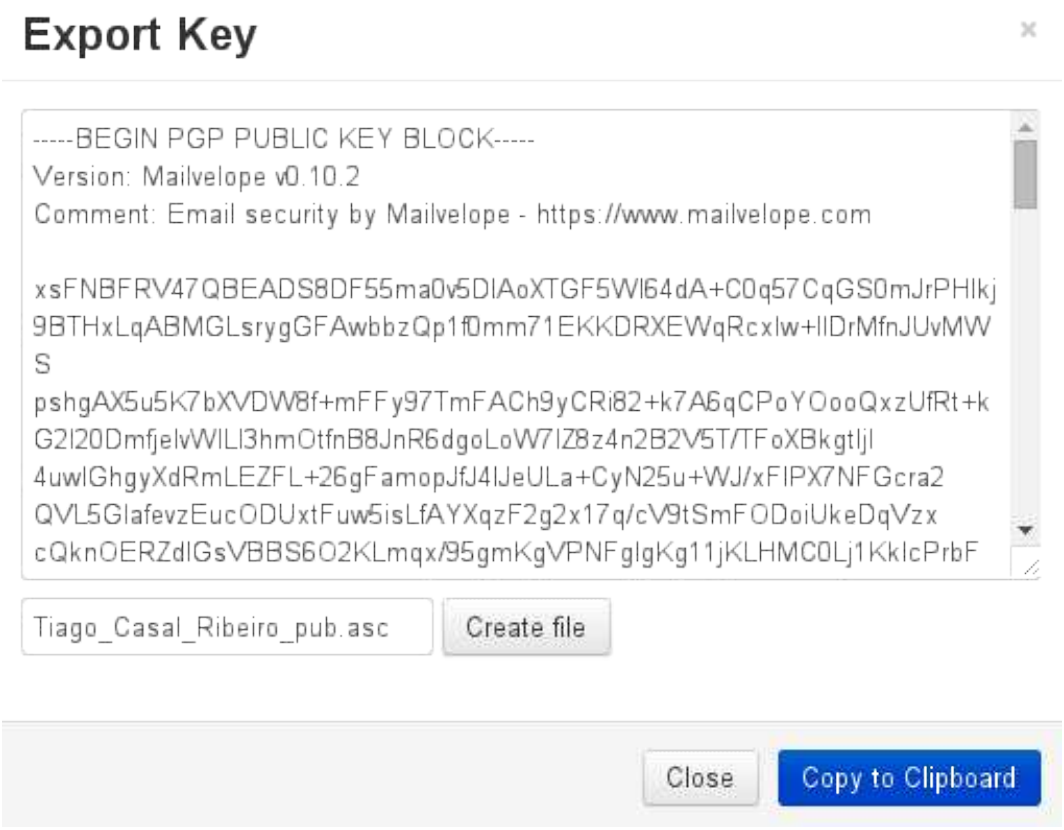


Figura 54: Uma chave pública criada pelo `Mailvelope`

- as chaves privadas são salvas na armazenagem do navegador que é insegura (porque vulnerável a ataques de Cross-Site Scripting Attack —`XSS`, em que um site acede aos dados locais armazenados para outro);
- a linguagem `Javascript` em que a extensão é escrita não é apropriada para uma cifração segura; entre outros, é suscetível às seguintes falhas :
  - outros scripts no navegador podem ler a chave secreta;
  - é impossível apagar a chave secreta da memória;
- com consenso do usuário, o provedor de e-mail tem acesso às chaves secretas do usuário (para sincronizá-las entre os aparelhos, o que é confortável, outra vez, significa igualmente um grande risco).

Por isso, é mais seguro usar um cliente de e-mail, como `Thunderbird` com `Enigmail`. Em seguida, para proteger as chaves da melhor forma, um cartão inteligente, como sera apresentado no fim deste capítulo.

Automatizar a Troca de Chaves. Todo programa que será apresentado (Autocrypt ou prettyeasyprivacy ou Delta-Chat) não é uma solução perfeita, mas conforme a RFC 7435 (Request for Comments, padrão livremente criado pelos usuários na internet) oferece apenas "Opportunistic Security: Some Protecion Most of the Time": Opportunistic Security significa uma proteção contra atacantes passivos, mas não contra atacantes ativos; isto é, a cifração da comunicação funciona só enquanto ninguém se interesse nela! Justamente pela falta da verificação do dono da chave privada que corresponde à chave pública, é vulnerável ao ataque "man-in-the middle" (MITM) em que o atacante se interpõe entre as duas partes que se comunicam; Por exemplo, é perfeitamente possível

- usar o nome de outrem numa conta de um e-mail, e
- usar o nome de outrem numa conta de WhatsApp.

A cifração da comunicação apenas impede que é lida por terceiros, mas não garante que a conta pertença à pessoa alegada. Para evitar este ataque, tem que verificar pessoalmente (ou por outro canal, por exemplo, por telefone) com o dono o "fingerprint" (= impressão digital = uma soma de verificação criptográfica) da sua chave pública.

Autocrypt. Na versão 2.0 a extensão Enigmail tem suporte para o programa Autocrypt que automatiza a troca de chaves públicas: Ele insere uma linha adicional no cabeçalho do e-mail (que é normalmente invisível para o usuário) que contém o certificado (nome, endereço de e-mail e referência à chave pública do usuário). Autocrypt foi iniciado por um projeto da União Europeia em resposta às revelações por Edward Snowden.

Em vez disto, o envio automático da chave pública de Alice, um melhor funcionamento seria:

1. O cliente de e-mail de Alice pede ao receptor Bob a cifração.
2. O cliente de e-mail de Bob põe-se automatizadamente com o cliente de Alice de acordo sobre as chaves públicas usadas.
3. Se Alice e Bob quiserem, podem verificar a impressão digital das chaves públicas por outro canal (por exemplo, por telefone).

Figura 55: Funcionamente de Autocrypt

prettyeasyprivacy. Outro projeto para mandar e-mails cifrados automaticamente como Autocrypt, fundado por uma iniciativa privada, que dá suporte a programas comerciais como o cliente de e-mail (e agendador, entre outros) Microsoft Outlook é prettyeasyprivacy.

Instala e lança mão de GPG(4Win), isto é, é uma interface gráfica no cliente de e-mail para este programa.

Uma ideia interessante é usar Safewords (palavras seguras) em vez da codificação hexadecimal para verificar a impressão digital de uma chave pública, isto é,

- em vez de transmitir muitos caracteres hexadecimais como 72F0 5CA5 0D2B BA4D 8F86 E14C 38AA E0EB,
- os interlocutores podem verificá-la por cinco palavras do seu idioma, por exemplo, Oceania contaminação arenas gogo ema; o que é bem mais apto para nos seres humanos.

`Delta-Chat`. Mais um projeto semelhante, para mandar mensagens instantâneas automaticamente cifradas para outros usuários, é o aplicativo `Delta-Chat` que usa a conta de e-mail do usuário.

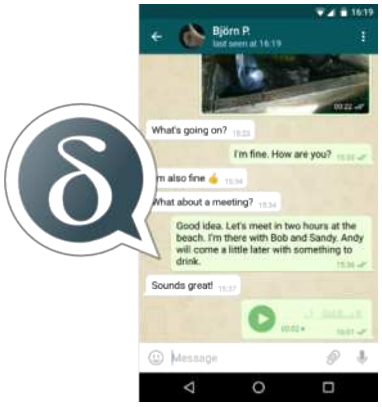


Figura 56: Delta-Chat

Como usa o mesmo protocolo aberto como o e-mail, em comparação a muitos outros aplicativos para mandar mensagens instantâneas automaticamente cifradas (como `WhatsApp`), não depende dos servidores de uma empresa específica. (Dizem que tem suporte a *federação*, por exemplo, um usuário de `hotmail.com` pode comunicar com outro de `gmail.com`.) Esta dependência traz os seguintes problemas:

- o lucro vem da compilação dos meta-dados dos usuários armazenados nos seus servidores (como `facebook.com`),
- o governo poderia suspender ou bloquear estes servidores (como no caso de `WhatsApp` aconteceu no Brasil e na China),
- os servidores da empresa podem ser comprometidos,
- a empresa muda de ideia sobre o modelo empresarial, e
- o usuário tem que ter confiança total nesta empresa que para quase todo mundo é uma grande entidade anônima.

Além disso, é compatível a destinatários que não usam o `Delta Chat` porque podem ser contatados por e-mail.

Isto dito, os meta-dados entre os servidores de e-mail não são cifrados. (O protocolo de e-mail `IMAP` é antigo e tem muitas deficiências; porém, é o estabelecido padrão, testado pelo tempo, com um grande ecossistema de programas.)

O mensageiro Conversations propõe um protocolo moderno XMPP que usa menos meta-dados que o protocolo IMAP. (Aparenta que o mensageiro Signal use menos meta-dados que Conversations, mas infelizmente quase todos os servidores são mantidos pela própria fornecedora Open Whisper Systems, em contraste ao Conversations. No fim das contas, a única solução segura é rodar o seu próprio servidor!)

É uma solução bastante completa quanto à segurança; infelizmente pouco estabelecida; por exemplo, há muitos servidores de e-mail (isto é, que comunicam pelo protocolo IMAP); porém, pouquíssimos que comunicam por XMPP.

5 Teoria dos Números Elementar

Explicamos a utilidade da aritmética modular na criptografia: Lembremo-nos que a função alçapão é facilmente computável, porém o seu inverso (por exemplo, a radiciação no RSA) é quase incomputável!

É esta dificuldade de calcular o inverso que corresponde à dificuldade da decifração, de inverter a cifração.

Os algoritmos criptográficos assimétricos como o RSA usam

- como função de cifração, que deve ser facilmente computável, a potenciação, e
- como função de decifração, que deve ser dificilimamente computável, pelo menos sem conhecimento da chave, o seu inverso, a radiciação.

Os algoritmos criptográficos, como o RSA, usam a *aritmética modular* para dificultar a computação da função inversa da potenciação, a radiciação. Já conhecemos a aritmética modular ou *circular*! Pela aritmético do relógio, onde um número fixo m, por exemplo $m = 12$ no caso do relógio, é considerado igual a 0. Como o indicador recomeça a contar a partir de 0 após cada volta, por exemplo, 3 horas após 11 horas são 2 horas:

$$11 + 3 = 14 = 12 + 2 = 2.$$

Sobre este domínios, chamados de *anéis finitos*, (os gráficos d') estas funções tornam-se irregulares e praticamente incomputáveis, pelo menos sem conhecimento de um atalho, a chave.

A seguir,

1. convencemo-nos da plausibilidade desta dificuldade (em comparação ao caso do domínio dos números reais),
2. construimos este domínio, e
3. explicamos como calcular o inverso sobre ele.

5.1 Aritmética Modular como Randomização

A dificuldade de calcular o inverso corresponde à dificuldade da decifração, de inverter a cifração. Na criptografia assimétrica,

- a *facilidade* de cifrar (um número), e
- a *dificuldade* de decifrar (um número)

baseiam-se em uma função invertível tal que

- ela seja **facilmente** computável, mas
- a sua função inversa seja **dificilmente** computável.

Por exemplo, os inversos das funções alçapão

- da *exponenciação* $x \mapsto g^x$ (no algoritmo `Diffie-Hellman`), e
- da *potenciação* $x \mapsto x^e$ (no algoritmo `RSA`)

são dados pela

- a radiciação $x \mapsto x^{1/n}$, e
- o logaritmo $\log_g$).

Observação. Observamos que as funções usadas são ambas *algébricas*, isto é, exprimem-se por uma fórmula (de somas, produtos e potências) finita. Funções *analíticas*, isto é, somas infinitas convergentes, por exemplo, o seno, cosseno, ..., parecem impraticáveis pelos erros que surgem do necessário arredondamento.

Eles são

- facilmente computáveis sobre os números reais (por exemplo, pelo método da bisseção para funções contínuas graças a conexidade de $\mathbb{R}$),
- porém, sobre os seus domínios criptográficos, são quase incomputáveis:

Introduzamos estes domínios:

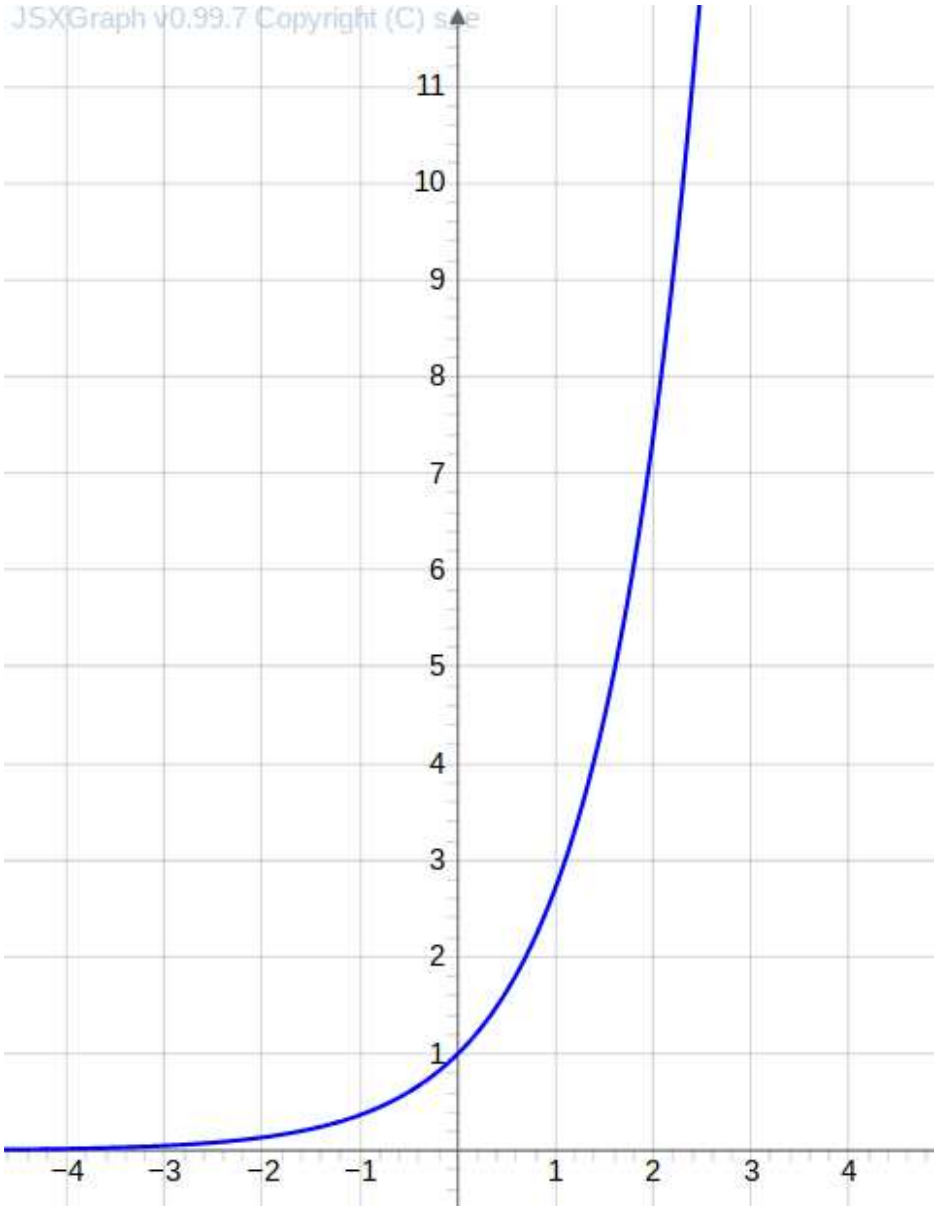


Figura 57: Exponencial $x \mapsto e^x$

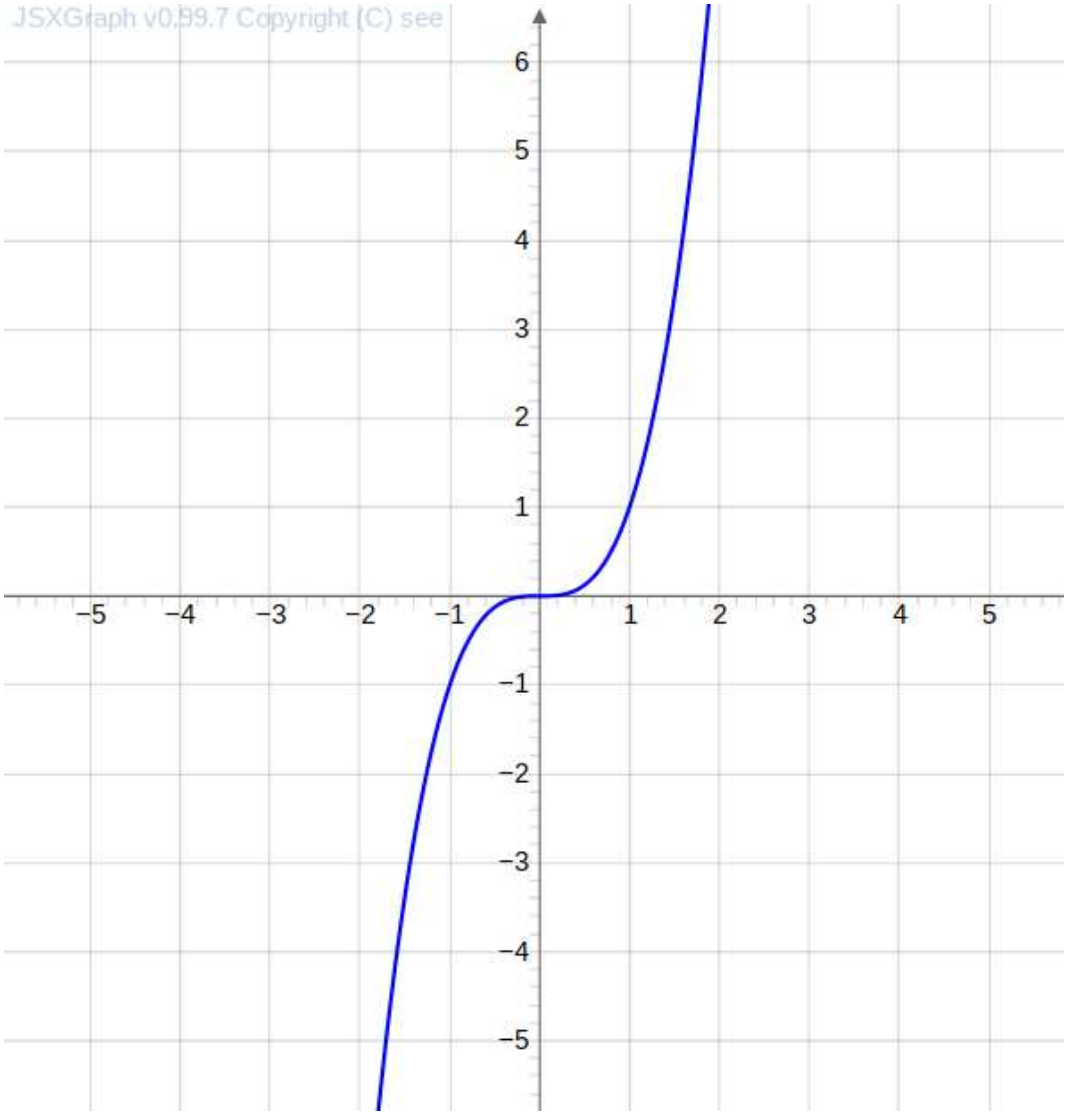


Figura 58: A parábola da função cúbica $x \mapsto x^3$

Funções sobre Conjuntos Discretos. O seu domínio **não** é o dos números inteiros $\mathbb{Z}$ (ou o dos números reais $\mathbb{R}$ que os contém), porque ambas as funções, exponenciação e potenciação, são **contínuas** sobre $\mathbb{R}$:

Se o domínio destas funções fosse $\mathbb{R}$, os seus *inversos* poderiam ser *aproximados* sobre $\mathbb{R}$, por exemplo, pelo **Método da Bisseção**: Dado y_0, encontrar um x_0 tal que $f(x) = y_0$ equivalha encontrar um **zero** x_0 da função

$$x \mapsto f(x) - y_0.$$

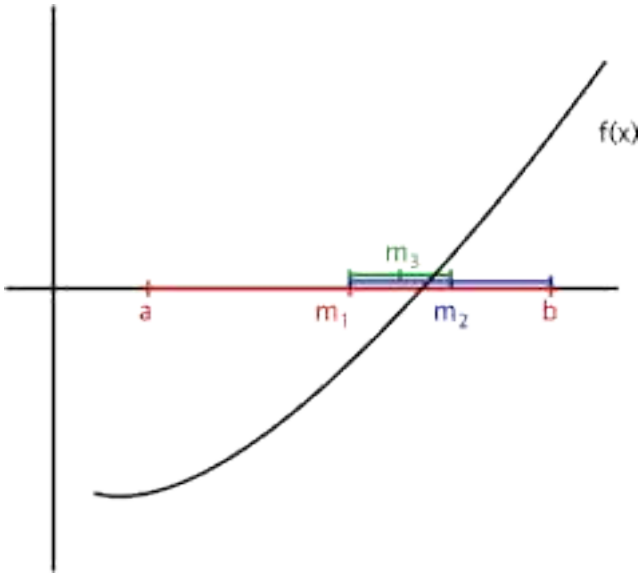


Figura 59: Bisseção de uma função contínua

1. (*Inicialização*) Escolha um intervalo $[a, b]$ tal que

$$f(a) < 0 \quad \text{e} \quad f(b) > 0.$$

2. (*Recalibração*) Calcule o seu meio $m := \frac{a+b}{2}$. Temos

   - ou $f(m) = 0$, então $m = x_0$,
   - ou $f(m) < 0$, então substituta a borda esquera a por m,
   - ou $f(m) > 0$, então substituta a borda direita b por m.

   e itere com o intervalo dado pelas novas bordas.

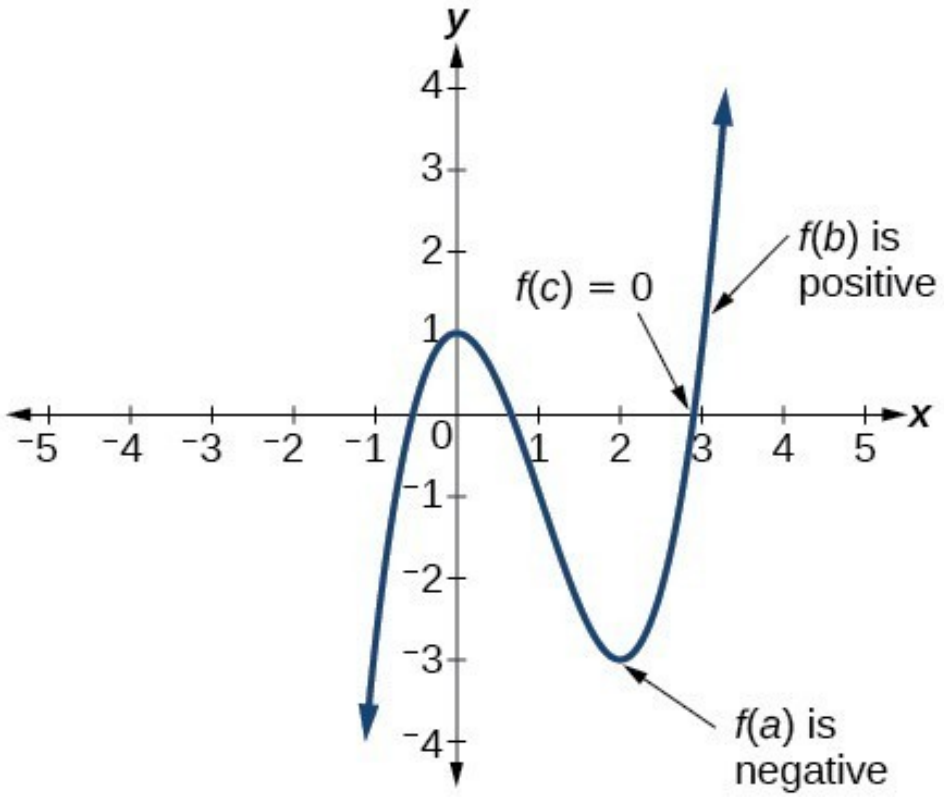


Figura 60: Teorema do Valor Intermédiario

Pelo **Teorema do Valor Intermediário**, o zero é garantido de estar no intervalo, que a cada passo diminui e converge à interseção.

Bisseção do polinômio $F(x) = x^3 + 3x^2 + 12x + 8$ com pontos iniciais $x_1 = -5$ e $x_2 = 0$

Step	x	F(x)	$\|x(i) - x(i-1)\|$
x_2	o	8	5
x_3	−2.5	−18.875	2.5
x_4	−1.25	−4.26563	1.25
x_5	−0.625	1.42773	0.625
x_6	−0.9375	−1.43726	0.3125
x_7	−0.7813	−0.02078	0.15625
x_8	−0.7031	0.69804	0.07813
x_9	−0.7422	0.33745	0.03906

Step	x	F(x)	$\|x(i) - x(i-1)\|$
x_{10}	−0.7617	0.15806	0.01953
x_{11}	−0.7715	0.06857	0.00977
x_{12}	−0.7764	0.02388	0.00488
x_{13}	−0.7783	0.00154	0.00244
x_{14}	−0.7800	−0.00962	0.00122

Anéis Finitos. Para evitar a iterada aproximação ao zero e assim **dificultar** a computação da função inversa (além de facilitar a computação da função mesma), o domínio de uma função alçapão é um **anel finito** (= principalmente um conjunto finito que contém 0 e 1 e sobre que uma soma + é definida)

$$\mathbb{Z}/m\mathbb{Z} = \{0, 1, \ldots, m-1\}$$

para um número natural m. Neles, vale $m = 1 + \cdots + 1 = 0$ e **toda adição (e logo toda multiplicação e toda potenciação) tem resultado** $< m$. Assim $\mathbb{Z}/m\mathbb{Z}$ é como anel **não** incluso em $\mathbb{R}$. Por exemplo, para $m = 7$,

$$2^2 = 2 \cdot 2 = 4 \quad \text{e} \quad 3^2 = 3 \cdot 3 = 7 + 2 = 0 + 2 = 2.$$

Introduziremos estes anéis finitos primeiro pelos exemplos $\mathbb{Z}/12\mathbb{Z}$ e $\mathbb{Z}/7\mathbb{Z}$ (= os anéis dos números das horas do relógio e dos dias da semana), depois para m qualquer.

Quando olhamos as funções, cujos gráficos são tão regulares em $\mathbb{R}$, no anel finito $\mathbb{Z}/101\mathbb{Z}$,

observamos que

- tanto a exponenciação com base 2
- quanto à parábola

é inicialmente tão regular sobre $\mathbb{Z}/101\mathbb{Z}$ quanto sobre $\mathbb{Z}$, mas

- a partir de $x = 7$ (porque $2^7 = 128 > 100$)
- respectivamente a partir de $x = 11$ (porque $11^2 = 121 > 100$)

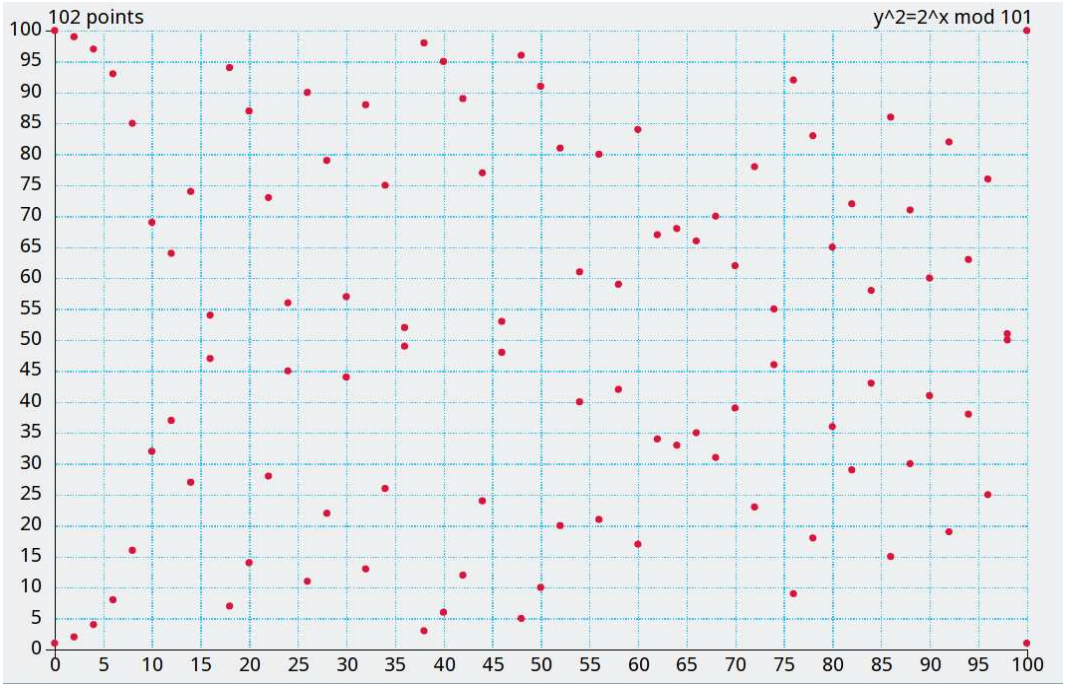


Figura 61: A exponencial com base 2 em $\mathbb{Z}/101\mathbb{Z}$

começa a comportar-se erraticamente (exceto a formosa simetria da parábola no eixo central $x = 50,5$ pela igualdade $(-x)^2 = x^2$).

Invitamos o caro leitor a experimentar com este plotter em linha de gráficos de funções discretos (de que origina a imagem acima), uma cortesia de Sascha Grau.

Observação: Observamos que dependente da base g da exponenciação $x \mapsto g^x$ e do expoente E da potência x^{E}, a função pode só ser invertível sobre subconjuntos do seu domínio e contra-domínio, isto é, dos seus argumentos e valores.

Para invertê-la, na sua *imagem* (= o conjunto dado por todos os seus valores), ela precisa de ser *injetora* (isto é, manda argumentos diferentes a valores diferentes).

A imagem da exponenciação nunca contém 0 e é possivelmente menor ainda do que $\mathbb{Z}/101\mathbb{Z}^* := \mathbb{Z}/101\mathbb{Z} - \{0\}$. Para $g = 2$ sobre $\mathbb{Z}/101\mathbb{Z}$, a sua imagem é máximo, quer dizer, é $\mathbb{Z}/101\mathbb{Z}^*$; em outras palavras, todos os números (diferentes de zero) são potências de g. (Diz-se que g *gera* $\mathbb{Z}/101\mathbb{Z}^*$.) Porém, por exemplo $g = 4 = 2^2$ gera sobre o mesmo domínio só a metade de $\mathbb{Z}/101\mathbb{Z}^*$, isto

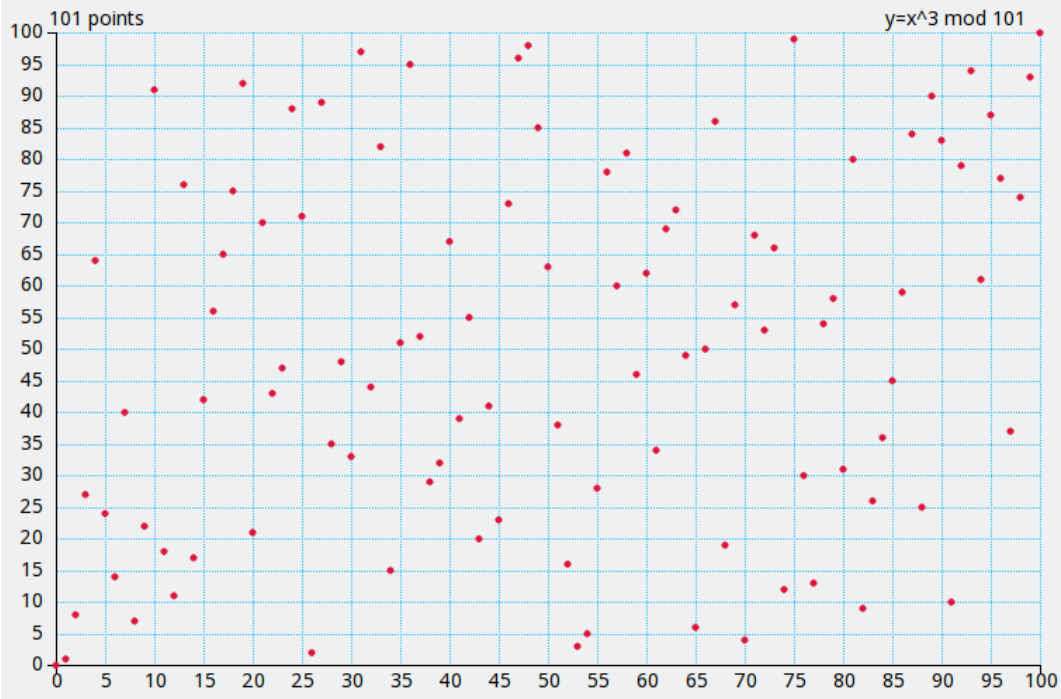


Figura 62: Os pontos da função cúbica $Y = X^3$ em $\mathbb{Z}/101\mathbb{Z}$

é, um conjunto de 50 elementos. Seção 7.2 mostrará que existe um *gerador* de $\mathbb{Z}/m\mathbb{Z}^*$ se, e tão-somente se,

- ou $m = 1, 2, 4$,
- ou $m = p^e$ respetivamente $m = 2p^e$ para um número primo $p > 2$.

Se o expoente E é par, $E = 2e$ para um inteiro e, então a potenciação $x \mapsto x^E$ satisfaz $(-x)^E = (-x)^{2e} = (-x)^{2^e} = x^{2^e} = x^{2e} = x^E$, isto é, manda $-x$ e x ao mesmo valor. Em particular, não é injetora. Observamos esta simetria para $x \mapsto x^2$ sobre $\mathbb{Z}/101\mathbb{Z}$ em Figura 63 ao longo do eixo central $x = 50,5$ (e também que a sua restrição a $\{0, \ldots, 50\}$ é injetora).

Dado um módulo m, Seção 7.2 mostrará para quais expoentes E a potenciação $x \mapsto x^E$ é injetora sobre $\mathbb{Z}/m\mathbb{Z}$:

- Para $m = p$ primo, são os sem divisor comum com $p - 1$. Por exemplo, para $m = 101$, o expoente $E = 3$;

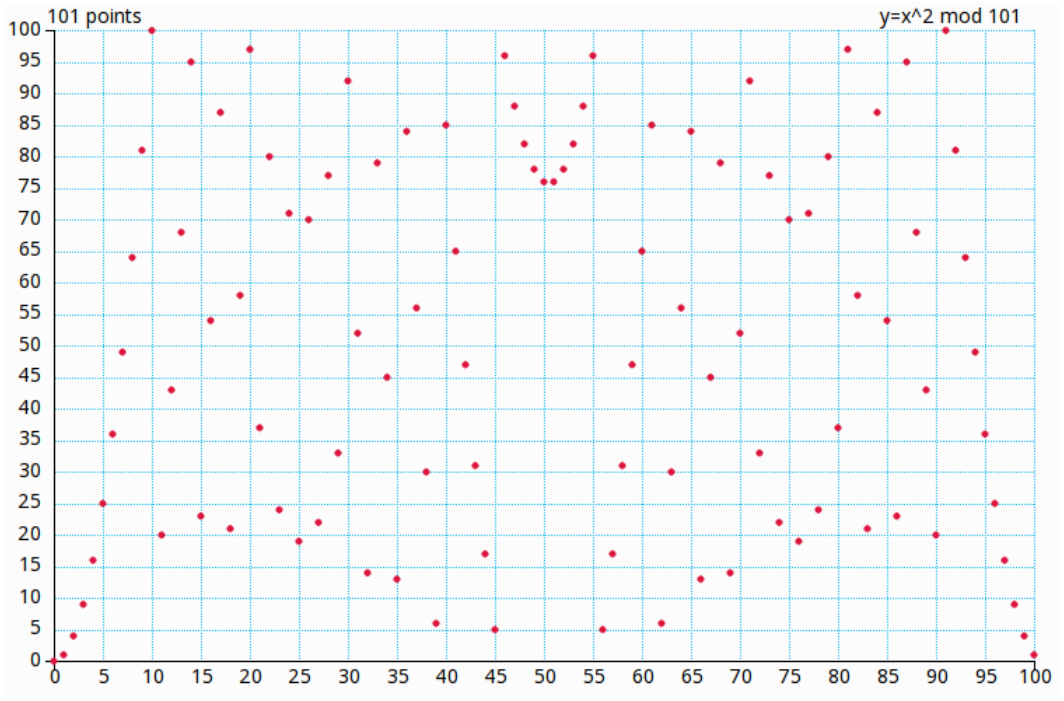


Figura 63: Os pontos da função quadrática $Y = X^2$ em $\mathbb{Z}/101\mathbb{Z}$

- Para $m = pq$ com p e q primo (é o tipo de módulo usado pelo RSA), são os sem divisor comum nem com $p - 1$, nem com $q - 1$. Por exemplo, para $m = 21 = 3 \cdot 7$, o expoente $E = 5$.

Vale a pena verificá-lo pelo plotter.

5.2 Aritmética Modular

Introduzamos esses domínios finitos $\mathbb{Z}/m\mathbb{Z}$ (*o anel dos inteiros módulo m*, para um número natural (comumente primo) m) das funções alçapão na criptografia assimétrica; são eles que dificultam tanto a nossa vida quando tentamos invertê-la (em comparação a $\mathbb{R}$ ou $\mathbb{Z}$).

Veremos que já os conhecemos e usamos no dia-a-dia para $m = 12$, na aritmética do relógio, e para $m = 7$, nos dias da semana.

Recordemo-nos breve da *Divisão com Resto*:

Definição. *Sejam a e b números inteiros positivos. Que a dividido por b tem* ***resto*** *r significa que existe um inteiro q tal que*

$$a = b \cdot q + r \quad \text{com } 0 \leq r < b.$$

Exemplo. Para $a = 230$ e $b = 17$, obtemos $230 = 17 \cdot 13 + 9$. Isto é, o resto de 230 dividido por 17 é 9.

Para m um número inteiro qualquer, definiremos o *anel* finito $\mathbb{Z}/m\mathbb{Z}$ (= para nós sobretudo um conjunto finito em que podemos somar) como segue:

- como conjunto, é $\{0, 1, ..., m-1\}$, e
- a soma de x e y em $\mathbb{Z}/m\mathbb{Z} = \{0, 1, ..., m-1\}$ será igualmente denotada por $x + y$, porém com outro significado sobre $\mathbb{Z}/m\mathbb{Z} = \{0, 1, ..., m-1\}$ do que sobre $\mathbb{Z}$! Para a soma $x + y$ em $\mathbb{Z}/m\mathbb{Z}$ não sair deste conjunto, isto é, $x + y < m$, calculemos *o resto* da soma $x + y$ em $\mathbb{Z}$ dividido por m. Logo, no relógio, isto é, para $m = 12$, vale, por exemplo, $7 + 8 = 3$.

Aritmética Modular no dia-a-dia. Damos exemplos da aritmética modular do nosso dia-a-dia (como o relógio, os dias da semana, ou até o alfabeto). A propriedade comum entre estes exemplos é a sua circularidade (de onde a nomenclatura "anel").

Relógio. O exemplo protótipo de aritmética modular é a aritmética do relógio em que o ponteiro volta após 12 horas no início; isto é, vale a equação

$$12 = 0,$$

que implica, entre outros, as equações

$$9 + 4 = 1 \quad \text{e} \quad 1 - 2 = 11; \quad (*)$$

Quer dizer, 4 horas depois das 9 horas é 1 hora, e 2 horas antes da 1 hora são 11 horas. Podemos ir mais longe:

$$9 + 24 = 9, \quad (**)$$

quer dizer se agora são 9 horas, então 24 horas (= um dia) mais tarde também.

Figura 64: Relógio como Anel dos números $1, 2, \ldots, 11, 12 = 0$

Dias da Semana. Além das horas, outro exemplo do aritmética modular no dia-a-dia são os dias da semana: tendo passado 7 dias, os dias da semana recomeçam. Se numeramos `sábado`, `domingo`, `segunda`, `terça`, `quarta`, `quinta` e `sexta-feira` por 0, 1, 2, 3, 4, 5, 6 então vale a equação

$$7 = 0,$$

e que implica, entre outros, as equações

$$4 + 4 = 1 \quad \text{e} \quad 1 - 2 = 5;$$

Quer dizer, 4 dias depois da quarta-feira é domingo, e 2 dias antes do domingo é sexta-feira. Podemos ir mais longe: $5 + 14 = 5$, quer dizer se agora é `quinta-feira`, então daqui em 14 dias (= duas semanas) também.

Figura 65: A circularidade semanal de tomar comprimidos

`Meses.` Além das semanas, outro exemplo do aritmética modular no dia-a-dia são os meses do ano: tendo passado 12 meses, os meses do ano recomeçam. Se numeramos `janeiro`, `fevereiro`, ... por 1, 2, ... então vale, como no relógio,

a equação

$$12 = 0,$$

e que implica, entre outros, as equações

$$10 + 4 = 1 \quad \text{e} \quad 1 - 2 = 11;$$

Quer dizer, um trimestre depois outubro recomeça o ano, e 2 meses antes de janeiro é novembro. Podemos ir mais longe: $5 + 24 = 5$, quer dizer se agora é `maio`, então daqui em 2 anos também.

A cifração de César. Na cifração de César, trasladamos cada letra do alfabeto por uma distancia t fixa; por exemplo, para $t = 3$, obtemos

$$\mathrm{A} \mapsto \mathrm{D}, \mathrm{B} \mapsto \mathrm{E}, \ldots, \mathrm{W} \mapsto \mathrm{Z}$$

Para trasladarmos as últimas $t = 3$ letras X, Y e Z do alfabeto, consideramos o alfabeto como anel, isto é:

Assim,

$$\mathrm{X} \mapsto \mathrm{A}, \ldots, \mathrm{Z} \mapsto \mathrm{C}.$$

Se identificamos cada letra do alfabeto romano com a sua posição,

$$\mathrm{A} \mapsto 0, \mathrm{B} \mapsto 1, \ldots, \mathrm{X} \mapsto 23, \mathrm{Y} \mapsto 24, \mathrm{Z} \mapsto 25.$$

então vale $23 + 3 \mapsto 0$, $24 + 3 = 1$ e $25 + 3 = 2$; isto é, $26 = 0$.

Formalização. Formalmente, derivamos as equações em $(*)$ e $(**)$ das igualdades

$$9 + 4 = 13 = 12 + 1 = 0 + 1 = 1 \quad \text{e} \quad 1 - 2 = -1 = -1 + 0 = -1 + 12 = 11.$$

e

$$9 + 24 = 9 + 2 \cdot 12 = 9 + 2 \cdot 0 = 9.$$

Em geral, para quaisquer a e x em $\mathbb{Z}$,

$$a + 12 \cdot x = a$$

Figura 66: O ciclo do ano

ou, equivalentemente, para quaisquer a e b em $\mathbb{Z}$,

$$a = b \quad \text{se } 12 \mid a - b.$$

Em palavras, a e b deixam o mesmo resto divido por 12.

As mesmas observações valem para os dias da semana.

Não há nada de especial sobre o número $m = 12$ (horas até meio-dia), $m = 7$ (dias da semana) ou $m = 26$ (letras do alfabeto latino). Por exemplo, as mesmas

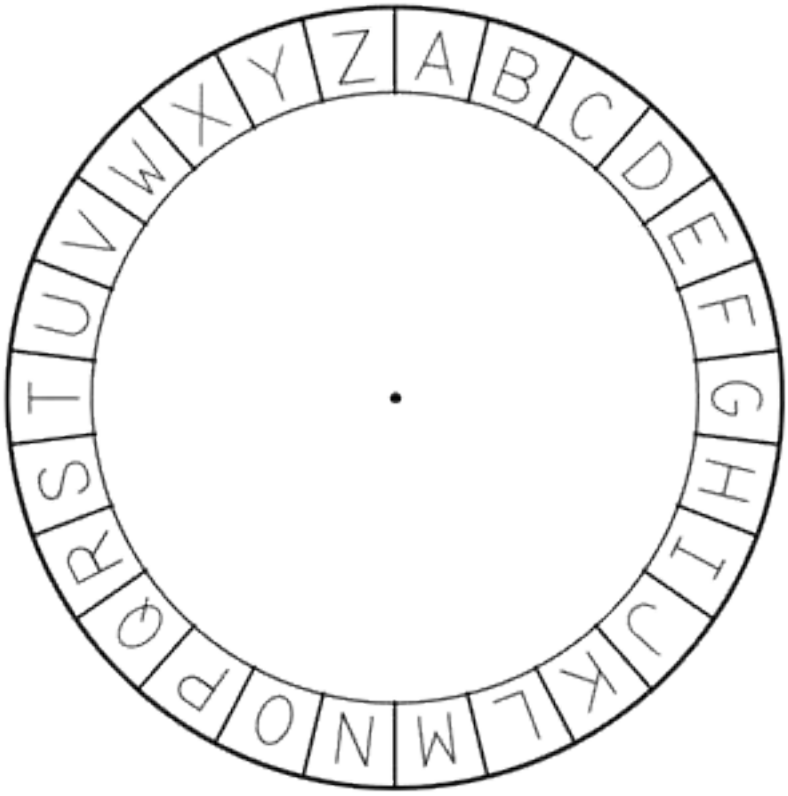


Figura 67: Roda das Letras do Alfabeto Latino

observações valeriam se o relógio indicasse $m = 15$ horas (como no planeta Netuno em que o dia, a rotação completa em torno do próprio eixo, dura 16 horas):

Definição. *Seja $m \geq 1$ um inteiro. Os números inteiros a e b são* ***congruentes módulo*** *m ou, em fórmulas,*

$$a \equiv b \bmod m.$$

se $m \mid a - b$, isto é se a sua diferença $a - b$ é divisível por m. Em outras palavras, se a e b deixam o mesmo resto divido por m. O número m é o ***módulo****.*

Exemplo: Para a cifração de César, sejam

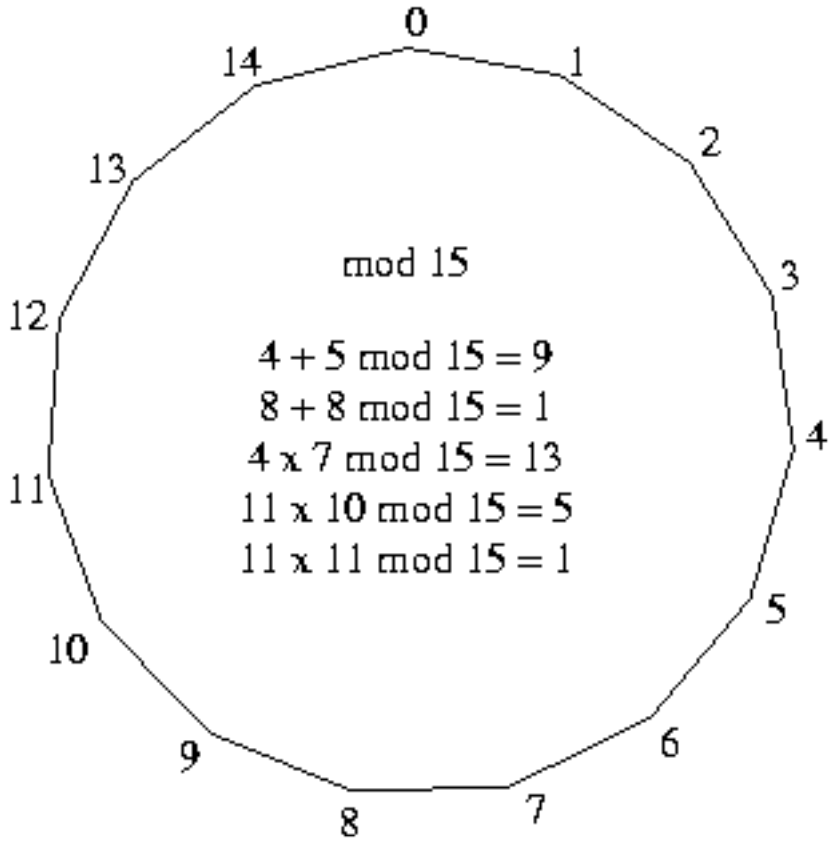


Figura 68: O relógio com 15 horas

- t a chave (= o *traslado*),
- i a (posição da) letra *inicial* e c a (posição da) letra *cifrada.*

A cifração e decifração se descrevem pelo aritmética modular por

$$c \equiv i + t \bmod 26 \quad \text{e} \quad i \equiv c - t \bmod 26$$

O anel finito das classes de resíduos. Dado um inteiro $m \geq 1$, queremos construir um conjunto $\mathbb{Z}/m\mathbb{Z}$ com 0 e 1 e sobre que podemos somar (isto é, existe uma operação +) tal que nele valha

$$m = 1 + \cdots + 1 = 0.$$

(Já observamos que, para esta igualdade valer, + sobre $\mathbb{Z}/m\mathbb{Z}$ tem de ser definida diferente de + sobre $\mathbb{Z}$.) Além do conjunto, queremos construir uma aplicação (ou talvez melhor *identificação*) $\bar{\cdot}\colon \mathbb{Z} \to \mathbb{Z}/m\mathbb{Z}$ tal que $\bar{x} = \bar{y}$ se, e tão-somente se, $x \equiv y \bmod m$. Isto é, $\bar{\cdot}$ identifica x e y se têm o mesmo resto divido por m.

Um tal conjunto chama-se *anel* (comutativo com 1). Mais exatamente, é

- um conjunto que contém 0 (= o elemento neutro da adição) e 1 (= o elemento neutro da multiplicação),
- com duas operações, a adição + (e o seu inverso −) e a multiplicação ·,
- que satisfazem a lei associativa, comutativa e distributiva.

Queremos construir de duas formas, prática e teórica,

- o *menor* anel, denotado por $\mathbb{Z}/m\mathbb{Z}$,
- com uma aplicação
$$\bar{\cdot}\colon \mathbb{Z} \to \mathbb{Z}/m\mathbb{Z}$$
que
  - respeita a adição, isto é $x \bar{+} y = \bar{x}+\bar{y}$ (e consequentemente $x \bar{\cdot} y = \bar{x}\cdot\bar{y}$, em particular, $\bar{0} = 0$ e $\bar{1} = 1$), e
  - identifica os múltiplos de m com 0, isto é,
$$x \mapsto 0 \iff m \mid x \qquad (\dagger)$$
Isto é, $x \equiv y \bmod m$ se, e tão-somente se, $\bar{x} = \bar{y}$ em $\mathbb{Z}/m\mathbb{Z}$.

Para o matematicamente interessado, a *propriedade definidora* do anel $\mathbb{Z}/m\mathbb{Z}$ é que ele seja

- o *menor* anel (minimidade)
- com uma aplicação que satisfaça (†).

Observamos que, por exemplo,

- por um lado, o anel $0 = \{0\}$ é menor que $\mathbb{Z}/m\mathbb{Z}$ e a aplicação $\mathbb{Z} \mapsto 0$ satisfaz $m \mapsto 0$ mas não somente $m\mathbb{Z} \mapsto 0$, isto é, (†) não vale;
- por outro lado, sobre o anel polinomial $\mathbb{Z}/m\mathbb{Z}[\mathrm{X}]$ a aplicação $\mathbb{Z} \to \mathbb{Z}/m\mathbb{Z} \hookrightarrow \mathbb{Z}/m\mathbb{Z}[\mathrm{X}]$ satisfaz (†), mas ele é maior que $\mathbb{Z}/m\mathbb{Z}$.

Matematicamente, a propriedade definidora da minimidade do anel $\mathbb{Z}/m\mathbb{Z}$ é uma *propriedade universal*: Qualquer aplicação $\mathbb{Z} \to \mathrm{A}$ que satisfaz $m \mapsto 0$ passa pela aplicação $\mathbb{Z} \to \mathbb{Z}/m\mathbb{Z}$; isto é, sempre há uma aplicação $\mathbb{Z}/m\mathbb{Z} \to \mathrm{A}$ tal que

$$\mathbb{Z} \to \mathbb{Z}/m\mathbb{Z} \to \mathrm{A} \quad = \quad \mathbb{Z} \to \mathrm{A}.$$

Construção Teórica. Esta é a construção encontrada em livros de matemática (universitária).

Equação (†) diz que exatamente $m\mathbb{Z} \mapsto 0$, e consequentemente, para x em $\mathbb{Z}$ qualquer, exatamente

$$x + m\mathbb{Z} \mapsto \bar{x}.$$

O que nos leva a pôr

- como conjunto

  $$\mathbb{Z}/m\mathbb{Z} := \{x + m\mathbb{Z} : x \text{ em } \mathbb{Z}\}$$

  e como aplicação $\bar{\cdot}\colon \mathbb{Z} \to \mathbb{Z}/m\mathbb{Z}$

  $$x \mapsto x + m\mathbb{Z};$$

- como elementos neutros da adição e multiplicação

  $$0 = 0 + m\mathbb{Z} = m\mathbb{Z} \quad \text{e} \quad 1 = 1 + m\mathbb{Z};$$

- como operações + e ·

  $$(x+m\mathbb{Z})+(y+m\mathbb{Z}) := (x+y)+m\mathbb{Z} \quad \text{e} \quad (x+m\mathbb{Z})\cdot(y+m\mathbb{Z}) := (x\cdot y)+m\mathbb{Z}.$$

  (Diante da definição da identificação $\bar{\cdot} = \cdot + m\mathbb{Z}$, esta definição de + é requerida pela igualdade $x \mathbin{\bar{+}} y = \bar{x} + \bar{y}$.)

Construção Prática. Esta é a construção encontrada em livros de ciências que aplicam a matemática (por exemplo, a ciência da computação).

Pomos

- como conjunto

  $$\mathbb{Z}/m\mathbb{Z} := \{0, \ldots, m-1\},$$

  e como aplicação $\bar{\cdot}\colon \mathbb{Z} \to \mathbb{Z}/m\mathbb{Z}$

  $$x \mapsto \mathrm{r}(x) \quad \text{onde } \mathrm{r}(x) \text{ é o resto de } x \text{ dividido por } m$$

- como elementos neutros da adição e multiplicação

$$0 \quad \text{e} \quad 1;$$

- como operações + e ·

$$x + y = \mathrm{r}(x + y) \qquad \text{onde } \mathrm{r}(x) = \text{ o resto de } x + y \text{ dividido por } m$$

e

$$x \cdot y = \mathrm{r}(x \cdot y) \qquad \text{onde } \mathrm{r}(x \cdot y) = \text{ o resto de } x \cdot y \text{ dividido por } m.$$

Isto é, para adicionar e multiplicar em $\mathbb{Z}/m\mathbb{Z}$,

1. calculamos a soma ou produto como em $\mathbb{Z}$, e
2. calculamos o seu resto divido por m.

Pela definição da identificação $\bar{\cdot} = \mathrm{r}$ como resto divido por m, a igualdade $x \bar{+} y = \bar{x} + \bar{y}$ requer que a soma dos restos seja definida pelo resto da soma.

Por exemplo, para $m = 4$ obtemos as tabelas de adição e multiplicação

+	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

e

*	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	0	2
3	0	3	2	1

Exercício. Mostra que um número inteiro é divisível por 3 (respectivamente 9) se, e tão-somente se, a soma dos seus algarismos decimais é divisível por 3

(respectivamente 9).

Exemplos em Python. Em `Python`, o operador modular é denotado pelo símbolo percentual `%`. Por exemplo, na shell interativa, obtemos:

```
>>> 15 % 12
3
>>> 210 % 12
6
>>> 20 % 10
0
```

Potenciação Rápida. A função alçapão

- no algoritmo `RSA` é uma potenciação $x \mapsto x^{\mathrm{E}}$, e
- a no algoritmo `ElGamal` (baseado na troca de chaves de Diffie-Hellman) é a exponenciação $x \mapsto g^x$

em $\mathbb{Z}/m\mathbb{Z}$ para um número natural m. Vemos como calculá-la eficazmente:

Algoritmo. Dados uma base b e um expoente e em $\mathbb{Z}$, para calcular

$$b^e \text{ em } \mathbb{Z}/\mathrm{M}\mathbb{Z},$$

1. Expande o expoente **binariamente**, isto é

$$e = e_0 + e_1 2 + e_2 2^2 + \cdots + e_s 2^s \text{ com } e_0, e_1, \ldots, e_s \text{ em } \{0,1\},$$

2. Calcula

$$b^1, b^2, b^{2^2}, \ldots, b^{2^s} \bmod \mathrm{M}.$$

Como $b^{2^{n+1}} = b^{2^n \cdot 2} = (b^{2^n})^2$, isto é, cada potência é o **quadrado da anterior** (e no máximo M), cada uma, sucessivamente, é facilmente computável. Obtemos

$$b^e = b^{e_0 + e_1 2 + e_2 2^2 + \cdots + e_s 2^s} = b^{e_0} (b^2)^{e_1} (b^{2^2})^{e_2} \cdots (b^{2^s})^{e_s}$$

Isto é, apenas as potências com expoentes $e_0, e_1, \ldots, e_s$ iguais a 1 importam, as outras podem ser omitidas.

Este algoritmo leva $2\log_2(e)$ multiplicações módulo M.

Exemplos. Para calcular 3^5 módulo 7, expandimos

$$5 = 1 + 0 \cdot 2^1 + 1 \cdot 2^2$$

e calculamos

$$3^1 = 3, 3^2 = 9 \equiv 2, 3^{2^2} = (3^2)^2 \equiv 2^2 = 4 \bmod 7,$$

obtendo

$$3^5 = 3^{1+2^2} = 3^1 \cdot 3^{2^2} = 3 \cdot 4 \equiv 5 \bmod 7.$$

Para calcular 3^{11} módulo 5, expandimos

$$11 = 1 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3$$

e calculamos

$$3^1 = 3, 3^2 = 9 \equiv 4, 3^{2^2} = (3^2)^2 \equiv 4^2 = 1 \text{ e } 3^{2^3} = 3^{2^2 \cdot 2} = (3^{2^2})^2 \equiv 1^2 = 1 \bmod 5,$$

obtendo

$$3^{11} = 3^{1+2^1+2^3} = 3^1 \cdot 3^{2^1} \cdot 3^{2^3} = 3 \cdot 4 \cdot 1 = 12 \equiv 2 \bmod 5.$$

6 Troca de Chaves segundo Diffie-Hellman

Para já nos convencermos da utilidade da aritmética modular, olhamos o protocolo para construir uma chave secreta mútua através de um canal inseguro, apresentado pela primeira vez em Diffie e Hellman (1976).

Isto não é exatamente criptografia assimétrica, ou criptografia de chave pública, porque a chave secreta é conhecida a *ambos os correspondentes*. Os alogritmos assimétricos que se baseiam neste protocolo (por exemplo, `ElGamal` e `ECC`) geram uma chave *de uso único* para *cada* mensagem. Esta geração tem de ser suficientemente aleatória para ser imprevisível.

6.1 Troca de Chaves

Observação. Denote daqui por diante, num algoritmo de criptografia assimétrica,

- uma letra redonda *maiúscula* exclusivamente um número *público*, e
- uma letra redonda *minúscula* preferivelmente um número *secreto* .

Para Alice e Bob construírem uma chave secreta através de um canal inseguro, combinam primeiro

- um número primo p *apropriado* (o *módulo*), e
- um número natural g *apropriado* (a *base*).

e depois

1. Alice, para gerar **uma metade** da chave, escolhe um número a,
   - calcula $\mathrm{A} \equiv g^a \bmod p$, e
   - transmite A ao Bob.
2. Bob, para gerar **outra metade** da chave, escolhe um número b,
   - calcula $\mathrm{B} \equiv g^b \bmod p$, e
   - transmite B à Alice.
3. A chave secreta **mútua** entre Alice e Bob é

$$c := \mathrm{A}^b = (g^a)^b = g^{ab} = g^{ba} = (g^b)^a = \mathrm{B}^a \bmod p.$$

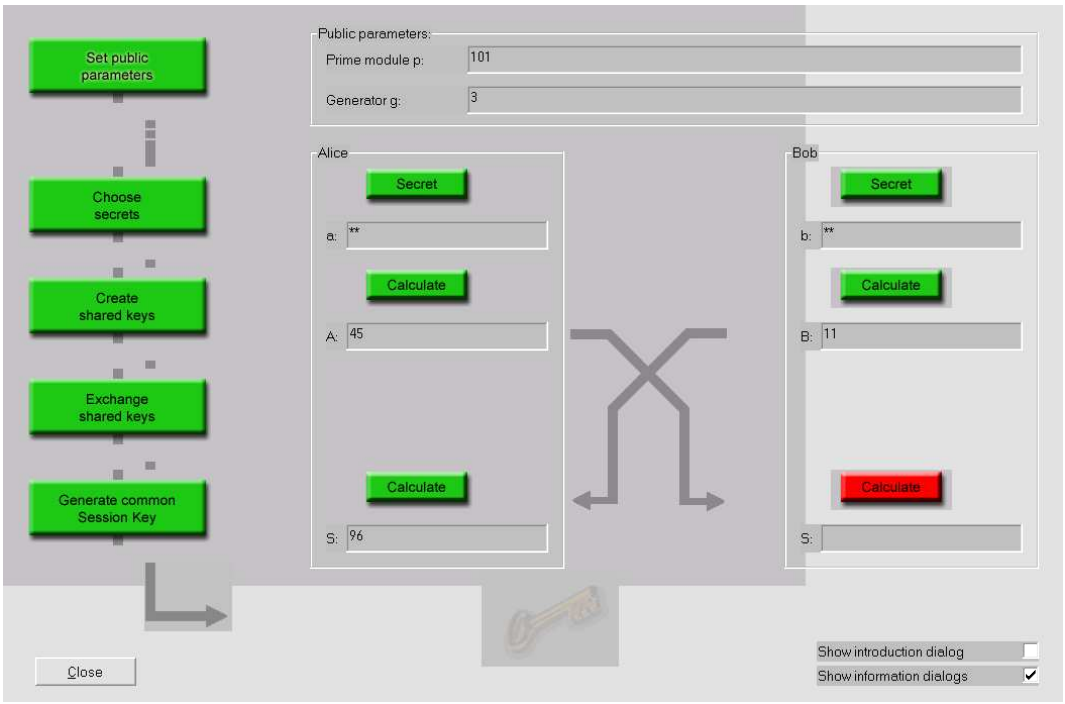


Figura 69: Os passos do `Diffie-Hellman` no `CrypTool 1`

O `CrypTool 1` oferece no Menu `Individual Procedures -> Protocols` uma entrada para experimentar com os valores das chaves no `Diffie-Hellman`.

Em Seção 8, o algoritmo `ElGamal` mostra como cifrar uma mensagem pela chave mútua (que constrói uma chave mútua efêmera por cada mensagem cifrada).

6.2 Segurança

A segurança da troca de chaves de Diffie-Hellman reside na dificuldade de computar o logaritmo módulo p.

Um olheiro obteria a chave secreta $\mathrm{A}^b = \mathrm{B}^a$ a partir de A e B, se pudesse computar

$$a = \log_g \mathrm{A} \quad \text{ou} \quad b = \log_g \mathrm{B} \bmod p;$$

isto é, o *logaritmo* $\log_g$ inverte a *exponenciação* $x \mapsto g^x = y$,

$$\log_g y = x \quad \text{com } x \text{ tal que } g^x = y.$$

Enquanto a exponenciação é facilmente computável, o **logaritmo é dificilimamente computável** para escolhas de p e g **apropriadas**, isto é:

- o número primo p
  - seja **grande** e
  - tal que haja um número primo **grande** q que divida $p-1$ (na melhor das hipóteses, $p-1=2q$ com q primo; neste caso p é chamado de um *primo seguro*);
- as potências da base g gerem um conjunto **grande** (isto é, a sua cardinalidade é um múltiplo de q)

$$\{g, g^2, g^3, \ldots\}.$$

Concluímos que, como A e B são públicos, a segurança computacional de `Diffie-Hellman` baseia-se unicamente na dificuldade da computação do logaritmo módulo p.

Mais exatamente, a segurança da Troca de Diffie-Hellman baseia-se no problema de Diffie-Hellman Computacional. Os três problemas da computação do logaritmo modular são, em ordem descendente da sua dificuldade:

- O problema do Logaritmo Discreto: Dado y, calcula x tal que $g^x = y$.

  Uma solução seria um algoritmo que leva tempo polinomial em (o número de bits de) y para calcular x.

- O problema de Diffie-Hellman *Computacional:* Dados $y' = g^{x'}$ e $y'' = g^{x''}$ (mas não x' e x''), calcula $Y = g^{x' \cdot x''}$.

  Uma solução seria um algoritmo que leva tempo polinomial em (o número de bits de) y' e y'' para calcular Y.

- O problema de Diffie-Hellman *Decisório:* Dados y', y'' e Y, decide se $y' = g^{x'}$, $y'' = g^{x''}$ e $Y = g^{x' \cdot x''}$ para alguns x' e x'' (que não precisam ser encontrados) ou não.

  Uma solução seria um algoritmo `BPP` (= bounded error probabilistic polynomial time) que leva tempo polinomial em (o número de bits de) y', y'' e Y para decidir com uma probabilidade de erro $< 1/2$ se $y' = g^{x'}$, $y'' = g^{x''}$ e $Y = g^{x' \cdot x''}$ para alguns x' e x''. (Pela repetição do algoritmo e a escolha do resultado mais frequente, a probabilidade de erro pode ser diminuída a um valor arbitrariamente pequeno > 0 em tempo polinomial.)

O primeiro problema é mais difícil do que o segundo, e o segundo e mais difícil que o terceiro; isto é: A solução do primeiro problema implica a do segundo, e a do segundo implica a do terceiro. Ou, equivalentemente, a hipótese da solução do terceiro problema implica a do segundo, e a do segundo implica a do primeiro.

Se o grupo que contém g é *genérico* (isto é, satisfaz apenas os axiomas de um grupo, mas nada além) e de cardinalidade prima ou desconhecida, então se demonstra que não existe solução do problema de Diffie-Hellman Decisório, logo, tampouco para os problemas de Diffie-Hellman Computacional e o Logaritmo Discreto. Porém, isto não exclui que exista solução em grupos específicos tal como $\mathbb{Z}/p\mathbb{Z}$ das unidades multiplicativas módulo um primo p; o que de fato ocorre!

Vice-versa: Toda solução conhecida do segundo problema se baseie na solução do primeiro; logo, supõe-se que, de fato, estes dois problemas sejam equivalentes, isto é, a solução do primeiro problema implique a do segundo, e reciprocamente. Existem evidências: Joux e Nguyen (2003) constrói uma família de grupos onde eles são equivalentes e, supostamente, irresolúveis (como definido acima). Antes, em 2001, Maurer e Wolf (1999) já demonstrou esta equivalência para grupos genéricos (baseado em observações para curvas elípticas finitas no início dos anos 90) com certas restrições nos divisores primos da cardinalidade do grupo.

Ao contrário, o terceiro problema é para muitos grupos mais fácil do que o segundo, isto é, existem grupos conhecidos em que o problema Diffie-Hellman Decisório é resolúvel, mas não o problema de Diffie-Hellman Computacional. (Este é um fenómeno não tão desconhecido: Por exemplo, enquanto leva tempo exponencial (no número de bites) para calcular os fatores primos de um número composto n, existem algoritmos que levam tempo polinomial para decidir se n é composto sem revelar os seus fatores. De fato, conjetura-se que todo algoritmo BPP, cuja saída pode ser verifica em tempo polinomial com probabilidade de erro $< 1/2$, é um algoritmo em P, os cuja saída pode ser verifica em tempo polinomial. Foi demonstrado em 2002 por Agrawal, Kayal e Saxena, um algoritmo polinomial existir para a verificação se um número é primo ou não, isto é, ser em P; antes, por exemplo, o Teste de Rabin-Miller, já mostrou que o problema é em BPP. [Na prática, o Teste de Rabin-Miller permanece o algoritmo preferido porque é muito mais rápido e a probabilidade de erro arbitrariamente pequena.])

O exemplo clássico e importante, de um grupo em que o problema de Diffie-Hellman Decisório é *resolúvel:* se g gera todo o grupo $\mathbb{Z}/p\mathbb{Z}^*$ (isto é, $g^e \equiv 1 \bmod p$ para $e = p-1$ pela primeira vez) então o valor de $g^a \bmod p$ em $\{\pm 1\}$ revela se a é par ou ímpar. Isto é, dados g^a, g^b e g^{ab}, calculam-se em tempo polinomial (no número de bits) os bits menos significativos de a, b e ab; logo, consegue-se pela sua comparação distinguir g^{ab} de um número qualquer em tempo polinomial com probabilidade de erro $< 1/2$. Isto é, o problema é resolúvel segundo a definição dada acima.

O exemplo clássico, de um grupo em que o problema de Diffie-Hellman Decisório é supostamente *irresolúvel* segundo muitos criptologistas, é um subgrupo de cardinalidade prima de $\mathrm{Z}_p^\times$ para p primo. Por exemplo, se $(p-1)/2$ é primo, (isto é, p é um primo *seguro*), então o grupo de quadrados módulo p é um tal grupo.

Observação. A *criptografia baseada em emparelhamentos* (pairing-based cryptography) usa (os grupos de) curvas elípticas finitas especialmente criadas em que o problema de Diffie-Hellman Computacional seja considerado difícil, mas o de Diffie-Hellman Decisório seja fácil. Isto é, outra indicação da suposta inequivalência entre eles.

Ele foi popularizada por Antoine Joux em 2002, e usa certo *emparelhamento de Weil* e na curva elíptica para, por exemplo, estabelecer um segredo comum entre *três* participantes (enquanto a Troca de Chave segundo Diffie-Hellman é restrita a dois): O emparelhamento é *bilinear*, isto é, $e(g^a, h^b) = e(g,h)^{ab}$. Se a Alice partilha $\mathrm{A} = g^a$, o Bob partilha $\mathrm{B} = g^b$ e o Charles partilha $\mathrm{C} = g^c$ para um inteiro aleatório e secreto c, então

- Alice calcula $e(g,g)^{abc} = e(g^b, g^c)^a = e(\mathrm{B},\mathrm{C})^a$,
- Bob calcula $e(g,g)^{abc} = e(\mathrm{A},\mathrm{C})^b$ e
- Charles calcula $e(g,g)^{abc} = e(\mathrm{A},\mathrm{B})^c$.

Pela bilinearidade, todos os valores são iguais. (Nota que g denota um ponto na curva, um *par* de números e a potenciação é a iterada adição deste ponto no grupo como explicado em Seção 9.3.)

Números Apropriados. **Teorema de Euclides**.

$$\#\{ \text{ números primos } \} = \infty$$

Demonstração: Suponhamos o contrário, que haja só um número finito $p_1, ..., p_n$ de números primos. Considere $q = p_1 ... p_n + 1$. Como q é maior que $p_1, \ldots, p_n$, não é primo. Seja então p um número primo que divida q. Pela hipótese, p em $\{p_1, ..., p_n\}$. Porém, pela sua definição, q tem o resto 1 divido por qualquer primo $p_1, \ldots, p_n$. Contradição!

O Teorema de Euclides garante que haja números primos arbitrariamente **grandes** (> 1024 bites). Em Seção 7.3, veremos como encontrá-los. Graças a Deus, quase todo número primo p satisfaz que

- haja um número primo q **grande** (> 768 bites) que divide $p - 1$.

O Teorema da *Raiz Primitiva* (que será demonstrado em Seção 7.2) garante que sempre haja um número g em $\mathbf{F}_p^*$ tal que

$$\{g, g^2, g^3, ..., g^{p-1}\}, = \mathbf{F}_p^* (= \{1, 2, 3, ..., p - 1\})$$

Em particular, a cardinalidade de $\{g, g^2, g^3, ..., g^{p-1}\}$ é um múltiplo de qualquer divisor primo q de $p - 1$.

Na prática, os números p e g são adotados duma fonte confiável.

Computação do Logaritmo. Como inicialmente os valores g^x sobre $\mathbb{Z}/p\mathbb{Z}$ igualam os valores de g^x sobre $\mathbb{Z}$, mais exatamente para $x < \log_g p$ (por exemplo, em Figura 61 com $g = 2$ e $p = 101$ para $x < 7$), os números secretos a e b devem ser suficientemente grande, mais exatamente, maior do que $\log_g p$. Para garantir isto na prática, estes números são artificialmente aumentados (por exemplo, se obtidos a partir de uma mensagem demasiada curta, ela é artificialmente enchida).

Presentemente, o algoritmo mais rápido para calcular o logaritmo x a partir de g^x, é uma modificação do *campo de número de peneira geral* de Gordon (1993). A grosso modo, o número de operações para calcular o logaritmo de um número inteiro de n bites é

$$\exp(\log n^{1/3}).$$

Módulos Arbitrários. Observemos para módulos que não são primos, isto é, um produto de fatores primos, que a dificuldade aumenta *linearmente* no número dos fatores, ao contrário dela aumentar *exponencialmente* no número dos bites de cada fator:

Produto de Primos Diferentes. Se o módulo $m = pq$ é produto de dois fatores p e q sem fator comum, então o logaritmo modular

$$\log_g \bmod m$$

pode ser computado, pelo Teorema Chinês dos Restos, pelos logaritmos

$$\log_g \bmod p \quad \text{e} \quad \log_g \bmod q$$

Mais exatamente, existem inteiros a e b, computados (em tempo linear no número dos bites de p e q) pelo Algoritmo de Euclides (estendido), tais que $ap + bq = 1$ e

$$\log_g \bmod m = a(\log_g \bmod p) + b(\log_g \bmod q).$$

Potência de um Primo. Se o módulo $m = p^e$ é uma potência de um primo p, então Bach (1984) mostra como o logaritmo modular módulo m para uma base g

$$\log_g : \mathbb{Z}/m\mathbb{Z}^* \to \mathbb{Z}/\phi(m)\mathbb{Z}$$

pode ser computado em tempo polinomial a partir do logaritmo módulo p. Exponhamos os passos para um número primo $p > 2$:

1. Recordemo-nos da subsecção sobre a existência da raiz primitiva para qualquer módulo de Seção 7.2 que $\mathbb{Z}/p^e\mathbb{Z}^*$ é cíclico de ordem $(p-1)p^{e-1}$. Logo, existe uma aplicação multiplicativa

   $$\mathbb{Z}/p^e\mathbb{Z}^* \to \mu_{p-1} \times U_1$$

   dada por

   $$x \mapsto x^{p^{e-1}}, x/x^{p^{e-1}} \quad (*)$$

   onde

   $$\mu_{p-1} = \{\zeta \in \mathbb{Z}/p^e\mathbb{Z}^* : \zeta^{p-1} = 1\}$$

   denote o grupo das $(p-1)$-ésimas raízes da unidade e

   $$U_1 = 1 + p\mathbb{Z}/p^e\mathbb{Z}$$

   o das unidades unitárias.

2. Temos o isomorfismo
$$\mu_{p-1} \to \mathbb{F}_p^*$$
dado por $x \mapsto x \bmod p$ e o seu inverso por $X \mapsto X^{p^{e-1}}$ para qualquer X em $\mathbb{Z}/p^e\mathbb{Z}$ tal que $X \equiv x \bmod p$. (Observa que a restrição do homomorfismo
$$\mathbb{Z}/p^e\mathbb{Z}^* \to U_1$$
dado por $x^{1-p^{e-1}}$ a U_1 é a identidade porque a ordem de U_1 é p^{e-1}.)
3. Temos o logaritmo para a base g
$$\log_g \colon \mathbb{F}_p^* \to \mathbb{Z}/(p-1)\mathbb{Z}$$
e temos o logaritmo natural
$$\log \colon U_1 \to p(\mathbb{Z}/p^e\mathbb{Z})$$
que é calculado em tempo polinomial pela fórmula
$$u \mapsto [x^{p^e} - 1]/p^e; \tag{2}$$
e o qual fornece o logaritmo $\log_g \colon U_1 \to p(\mathbb{Z}/p^e\mathbb{Z})$ para a base g pelo escalamento
$$\log_g = \log \cdot / \log g.$$
4. Pelo Teorema Chinês dos Restos, temos o isomorfismo
$$\mathbb{Z}/(p-1)\mathbb{Z} \times \mathbb{Z}/p^{e-1}\mathbb{Z} \to \mathbb{Z}/(p-1)p^{e-1}\mathbb{Z}$$
dada pelo produto e o seu inverso dado por $y \mapsto (ay \bmod p, by \bmod p^{e-1})$ onde a e b satisfazem $a(p-1)+b(p^{e-1}) = 1$ e foram obtidos pelo Algoritmo de Euclides (estendido).

Concluímos que, dado

- o número y em $\mathbb{Z}/p^e\mathbb{Z}$ e
- o seu valor $\log_g(y)$ sob $\log_g \colon \mathbb{F}_p^* \to \mathbb{Z}/(p-1)\mathbb{Z}$,

o valor $\log_g(y)$ de $\log_g \colon (\mathbb{Z}/p^e\mathbb{Z})^* \to \mathbb{Z}/(p-1)p^{e-1}\mathbb{Z}$ é computado em tempo polinomial.

Observação. Para facilitar a computação, ao invés da projeção
$$\mathbb{Z}/p^e\mathbb{Z}^* \to U_1$$

dada em $(*)$ por $x \mapsto x^{1-p^{e-1}}$, é mais rápido usar a dada por $\pi\colon x \mapsto x^{p-1}$. Porém, a sua restrição a U_1 não é a identidade. Logo, é preciso usar ao invés de

$$\log g\colon \mathrm{U}_1 \to p\mathbb{Z}/p^e\mathbb{Z}$$

o logaritmo escalado

$$(p-1)^{-1}\log_g$$

para obter

$$\log_g = (p-1)^{-1}\log_g \circ\pi = (\log(g)p-1)^{-1}\log\circ\pi\colon \mathrm{U}_1 \to p\mathbb{Z}/p^e\mathbb{Z}.$$

Logaritmo Discreto Para uma Potência de um Primo. Fundamentemos a Equação 2 que define o logaritmo $\log\colon \mathrm{U}_1 \to p(\mathbb{Z}/p^e\mathbb{Z})$: recordemos a definição da exponencial sobre $\mathbb{R}$ pelos juros compostos

$$\exp(x) = \lim\left(1+\frac{1}{n}\right)^n,$$

a qual leva à definição da função inversa

$$\log(x) = \lim n(x^{\frac{1}{n}}-1) = \lim\frac{1}{\epsilon}(x^{\epsilon}-1)$$

onde $\epsilon = \frac{1}{n} \to 0$.

Ora, em $\mathbb{Z}/p^e\mathbb{Z}$, temos $1, p, p^2, \ldots, p^e = 0$, isto é, $p^n \to 0$, o que talvez motive a ideia de considerar p como pequeno. Logo, o bom análogo sobre U_1 é

$$\log(x) = \lim\frac{1}{p^{e-1}}(x^{p^{e-1}}-1).$$

Com efeito, sobre U_1

$$\log(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \ldots$$

é um valor bem-definido em $p\mathbb{Z}/p^e\mathbb{Z}$, porque se p divide x, então nenhum denominador da fração cortada é divisível por p e todos os números indivisíveis por p são invertíveis em $\mathbb{Z}/p^e\mathbb{Z}$. Da mesma maneira, sobre $p\mathbb{Z}/p^e\mathbb{Z}$,

$$\exp(x) = \sum_{n\geq 0}\frac{x^n}{n!}$$

é um valor bem definido em $1 + p\mathbb{Z}/p^e\mathbb{Z}$, porque se p divide x, então nenhum denominador cortado é divisível por p e todos os números indivisíveis por p são invertíveis em $\mathbb{Z}/p^e\mathbb{Z}$.

De interesse particular é a base e^p do logaritmo natural em $1 + p\mathbb{Z}/p^e\mathbb{Z}$, isto é, o argumento y tal que $\log y = 1$. Por exemplo, para $p = 7$ e $e = 4$, calculamos

$$\exp(p) = \sum_{n \geq 0} \frac{p^n}{n!} = 1 + p + \frac{p^2}{2} + \frac{p^3}{3!} = 1 + 7 \cdot 127 = 1 + 1 \cdot 7 + 4 \cdot 7^2 + 2 \cdot 7^3.$$

7 Multiplicação e Divisão Modular

Estudamos a multiplicação nos anéis finitos $\mathbb{Z}/n\mathbb{Z}$. Interessamo-nos em particular por quais números podemos dividir neles. Será o Algoritmo de Euclides que nos computa a reposta.

7.1 O Algoritmo de Euclides

A chave privada

- no algoritmo `RSA`, ou
- da cifração da mensagem no algoritmo `ElGamal` (baseado na troca de chaves de Diffie-Hellman),

define uma função que é o inverso da função definida pela chave pública. Este inverso é computado via o *maior divisor comum* entre os dois números. Este, por sua vez, é computado pelo *Algoritmo de Euclides*, uma iterada divisão com resto.

Introduzimos então

- a noção do *maior divisor comum* de dois números inteiros, e
- como calculá-lo pela divisão com resto, o dito *algoritmo de Euclides.*

Sejam a e b números inteiros. Denotamos por $a \mid b$ que a divide b, isto é, que b é um múltiplo de a (existe um inteiro q tal que $b = qa$). Recordemo-nos de umas implicações básicas sobre a divisibilidade entre números inteiros:

Proposição 1. *Sejam a, b, c números inteiros.*

(a) *Se $a|b$ e $b|c$, então $a|c$.*
(b) *Se $a|b$ e $b|a$, então $a = \pm b$.*
(c) *Se $a|b$ e $a|c$, então $a|b \pm c$.*

Definição. *Um **divisor comum** de dois números inteiros a e b é um número natural que divide os dois. **O maior divisor comum** de dois números inteiros a e b é o maior número natural que divide os dois.* Denote $\mathrm{mdc}(a, b)$ o maior divisor comum de a e b,

$$\mathrm{mdc}(a, b) = \text{o maior número natural que divide } a \text{ e } b$$

Exemplo. O maior divisor comum de 12 *e* 18 *é* 6.

Definição. *Os números inteiros a e b são **relativamente primos** se* $\mathrm{mdc}(a,b) = 1$, isto é, se nenhum número inteiro > 1 divide a e b.

Para quaisquer números inteiros a e b, os números inteiros a/g e b/g para $g = \mathrm{mdc}(a,b)$ são relativamente primos.

A *divisão com resto* ajuda-nos a construir um algoritmo eficiente para calcular o maior divisor comum. Recordemos-nos, mais uma vez *Divisão com Resto*:

Definição. *Sejam a e b números inteiros positivos. Que a dividido por b tem **quociente** q e **resto** r significa*

$$a = b \cdot q + r \quad \text{com } 0 \leq r < b. \tag{3}$$

Exemplo. Para $a = 19$ e $b = 5$, obtemos $19 = 5 \cdot 3 + 4$. Isto é, o resto de 19 dividido por 5 é 4.

Uma **combinação linear** (ou **soma de múltiplos**) de dois números inteiros a e b é uma soma

$$s = \lambda a + \mu b$$

para números inteiros λ e μ.

Exemplo. Para $a = 15$ e $b = 9$, uma soma de múltiplos deles é

$$s = 2 \cdot a + (-3) \cdot b = 2 \cdot 15 - 3 \cdot 9 = 3.$$

É importante observar pela Proposição 1.C a seguinte implicação: se um número inteiro d divide a e b, então d divide toda combinação linear s de a e b.

Algoritmo de Euclides. Em particular, olhando à divisão com resto em Equação 3, para um número inteiro d, observamos:

- se d divide a e b, então a sua combinação linear $r = a - q \cdot b$, e, igualmente,
- se d divide b e r, então a sua combinação linear a.

Isto é, d divide a e b se, e tão-somente se, d divide b e r. Quer dizer, os divisores comuns de a e b são os mesmos que os de b e r. Em particular,

$$\mathrm{mdc}(a,b) = \mathrm{mdc}(b,r).$$

Dividindo com resto os números b e r (que é $< b$), obtemos

$$b = r \cdot q' + r' \quad \text{com } 0 \leq r' < r$$

e

$$\mathrm{mdc}(b, r) = \mathrm{mdc}(r, r').$$

Iterando, e assim diminuindo o resto, chegamos a $s = r'^{\ldots'}$ e $r'^{\ldots''}$ com $r'^{\ldots''} = 0$, isto é

$$\mathrm{mdc}(a, b) = \ldots = \mathrm{mdc}(s, 0) = s.$$

Em palavras, o maior divisor comum é o *penúltimo* resto (ou o último diferente de 0).

Exemplo. Para calcular $\mathrm{mdc}(748, 528)$*, obtemos*

$748 = 528 \cdot 1 + 220$

$528 = 220 \cdot 2 + 88$

$220 = 88 \cdot 2 + 44$

$88 = 44 \cdot 2 + 0$

logo $\mathrm{mdc}(528, 220) = 44$.

O `CrypTool 1`, na entrada do menu `Indiv. Procedures -> Number Theory Interactive -> Learning Tool for Number Theory`, Seção `1.3`, página 15, mostra uma animação deste algoritmo:

Formalizemos o que acabamos de observar:

Teorema. *(Algoritmo de Euclides) Sejam a e b números inteiros positivos com $a \geq b$. O seguinte algoritmo calcula* $\mathrm{mdc}(a, d)$ *em um número finito de passos:*

(início) Ponha $r_0 = a$ e $r_1 = b$, e $i = 1$.

(divisão) Divida r_{i-1} por r_i com resto para obter

$$r_{i-1} = r_i q_i + r_{i+1} \quad \text{com } 0 \leq r_{i+1} < r_i.$$

(distinção) Distinga entre:

- ou $r_{i+1} > 0$, então ponha $i := i + 1$ e continue no passo (divisão),
- ou $r_{i+1} = 0$, então $r_i = \mathrm{mdc}(a, b)$ e o algoritmo termina.

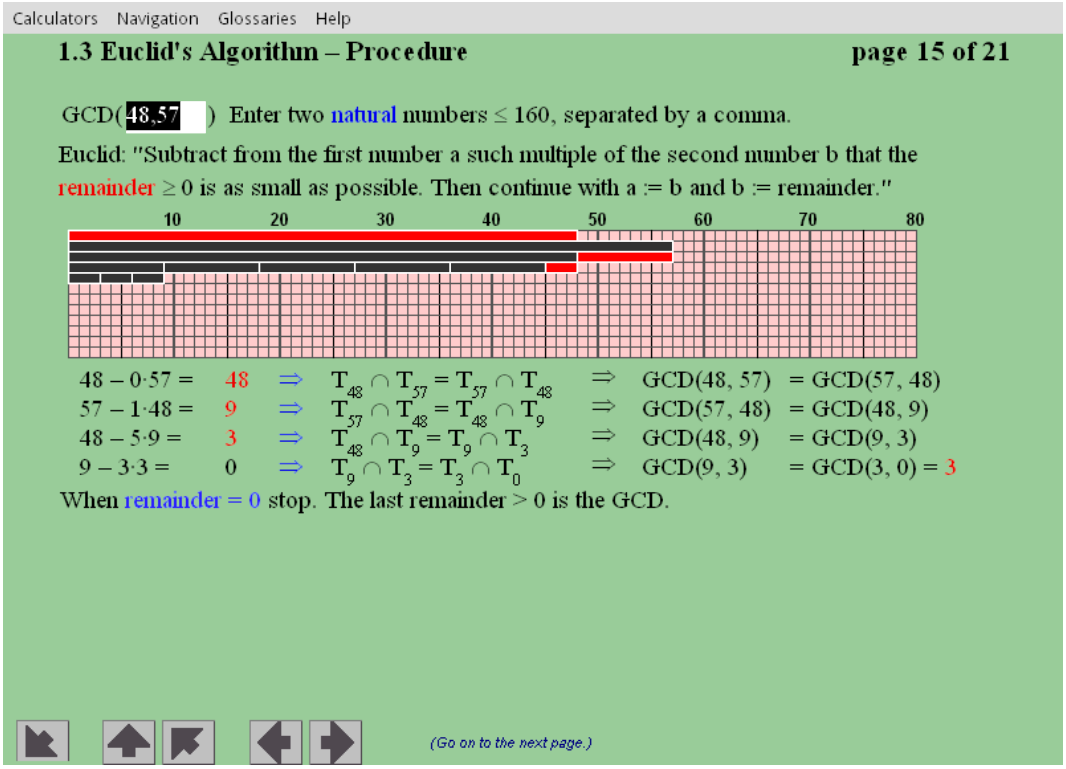


Figura 70: O algoritmo de Euclides no CrypTool 1

Demonstração: Precisamos de demonstrar que o algoritmo termina com o maior divisor comum de a e b:

- Como $r_0 > r_1 > \ldots$, finalmente $r_I = 0$ para I suficientemente grande, e o algoritmo termina.
- Pela Equação 3, temos

$$\text{mdc}(r_{i-1}, r_i) = \text{mdc}(r_i, r_{i+1}) \quad \text{para todos os } i = 1, 2, \ldots$$

  Como ultimamente $r_{I+1} = 0$ para I suficientemente grande, temos

$$\text{mdc}(a,b) = \text{mdc}(r_0, r_1) = \ldots = \text{mdc}(r_i, r_{i+1}). = \text{mdc}(r_I, 0) = r_I$$

  Isto é, $r_I = \text{mdc}(a,b)$.

Observação: Bastam $2 \cdot \log_2 b + 1$ divisões com resto para o algoritmo terminar.

Demonstração: Demonstramos

$$r_{i+2} < 1/2 \cdot r_i. \quad (\dagger)$$

Temos

- ou $r_{i+1} \leq 1/2 \cdot r_i$, e então $r_{i+2} < r_{i+1} \leq r_i$,
- ou $r_{i+1} > 1/2 \cdot r_i$.

Neste último caso, segue

$$r_i = r_{i+1} \cdot 1 + r_{i+2}$$

e então

$$r_{i+2} = r_i - r_{i+1} < r_i - 1/2 \cdot r_i = 1/2 \cdot r_i.$$

Por (†) obtemos iterativamente que bastam $2 \cdot \log_2 b + 1$ divisões com resto para o algoritmo terminar.

Com efeito, revela-se que já bastam no máximo $1.45 \log_2(b) + 1.68$ divisões com resto, e em média já $0.85 \log_2(b) + 0.14$.

Algoritmo de Euclides Estendido. Para a computação do inverso modular, precisamos de mais informações do que o maior divisor comum (calculado pelo Algoritmo de Euclides).

Com efeito, observa-se que em cada passo do Algoritmo de Euclides o maior divisor comum $\mathrm{mdc}(x, m)$ de x e m é uma *combinação linear* (ou *soma de múltiplos*) de x e m, isto é,

$$\mathrm{mdc}(x, m) = \lambda x + \mu m \quad \text{para inteiros } x \text{ e } m.$$

O inverso de x módulo m é um destes múltiplos. Vamos apresentar o Algoritmo de Euclides *Estendido* que preserva esta informação.

Teorema. *(Algoritmo de Euclides Estendido) Para quaisquer números inteiros positivos a e b, o seu maior divisor comum* $\mathrm{mdc}(a, b)$ *é uma combinação linear de a e b; isto é, há números inteiros u e v tais que*

$$au + bv = \mathrm{mdc}(a, b).$$

Demonstração: Como $r_0 = a$, $r_1 = b$, e $r_2 = r_0 - q_1 r_1$, segue que r_2 é uma combinação linear de a e b. Em geral, como r_{i-1} e r_i são combinações lineares de a e b, primeiro $q_i r_i$ é uma combinação linear de a e b, e assim

$$r_{i+1} = r_{i-1} - q_i r_i$$

é uma combinação linear de a e b. Em particular, se $r_{I+1} = 0$ então $r_I = \text{mdc}(r_I, r_{I+1}) = \text{mdc}(a, b)$ é uma combinação linear de a e b.

O CrypTool 1, na entrada do menu `Indiv. Procedures -> Number Theory Interactive -> Learning Tool for Number Theory`, Seção 1.3, página 17, mostra uma animação deste algoritmo:

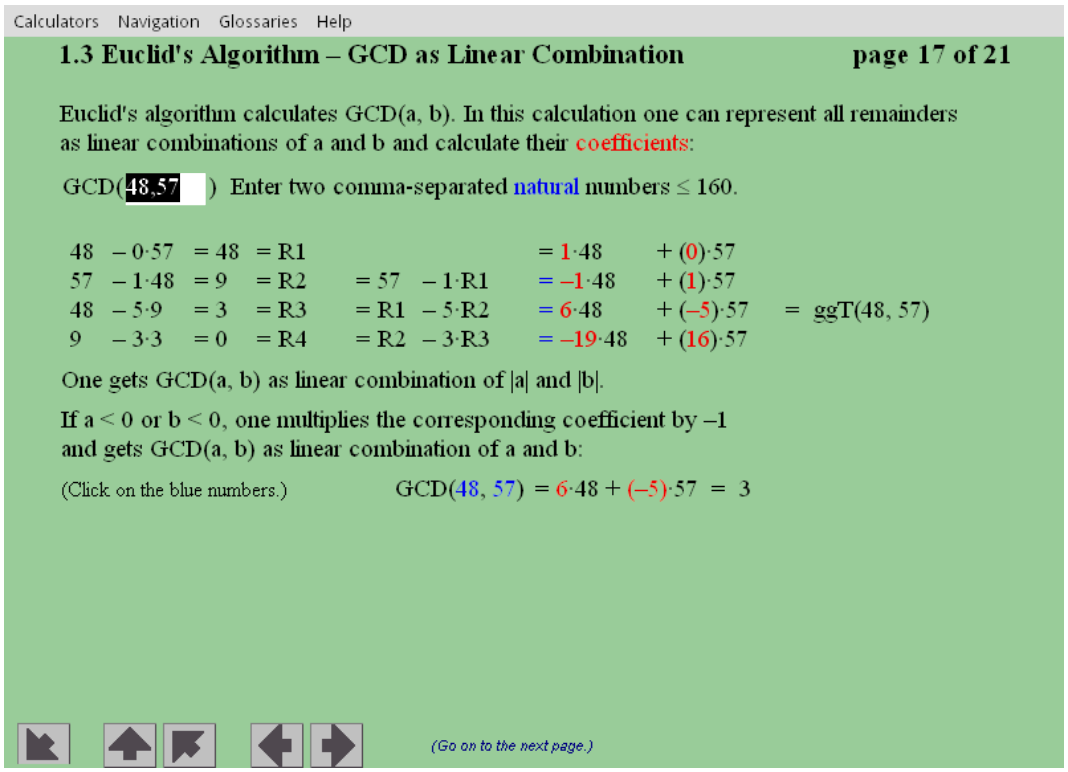


Figura 71: O algoritmo de Euclides estendido no CrypTool 1

Exemplo. Revenhamos à calculação do maior divisor comum de $a = 748$ e $b = 528$. O algoritmo de Euclides deu:

$748 = 528 \cdot 1 + 220$

$528 = 220 \cdot 2 + 88$

$220 = 88 \cdot 2 + \mathbf{44}$

$88 = 44 \cdot 2 + 0,$

o que fornece as combinações lineares

$220 = 748 - 528 \cdot 1 = a - b$

$88 = 528 - 220 \cdot 2 = b - (a - b) \cdot 2 = 3b - 2a$

$\mathbf{44} = 220 - 88 \cdot 2 = (a - b) - (3b - 2a) \cdot 2 = 5a - 7b.$

Com efeito,

$$44 = 5 \cdot 748 - 7 \cdot 528.$$

Implementação em Python. Para implementar o algoritmo de Euclides, vamos usar atribuição múltipla em Python:

```
>>> spam, eggs = 42, 'Hello'
>>> spam
42
>>> eggs
'Hello'
>>> a, b, c, d = ['Alice', 'Bob', 'Carol', 'David']
>>> a
'Alice'
>>> b
'Bob'
>>> c
'Carol'
>>> d
'David'
```

Os nomes das variáveis e os seus valores são alistados à esquerda respectivamente à direita de =.

Algoritmo de Euclides. Aqui é uma função que implementa o algoritmo de Euclides em Python; ela devolve o maior divisor comum mdc(a,b) de dois inteiros a e b.

```
def gcd(a, b):
    while a != 0:
        a, b = b % a, a
    return b
```

Por exemplo, no shell interativo:

```
>>> gcd(24, 30)
6
>>> gcd(409119243, 87780243)
6837
```

Algoritmo de Euclides Estendido. O operador // vai figurar na implementação do algoritmo de Euclides estendido; ele divide dois números e arredonda para baixo. Isto é, devolve o maior inteiro igual ou menor que o resultado da divisão. Por exemplo, no shell interativo:

```
>>> 41 // 7
5
>>> 41 / 7
5.857142857142857
>>> 10 // 5
2
>>> 10 / 5
2.0
```

Optamos pela seguinte implementação do algoritmo de Euclides estendido:

```
def egcd(a, b):
    x,y, u,v = 0,1, 1,0
    while a != 0:
        q, r = b//a, b%a
        m, n = x-u*q, y-v*q
        b,a, x,y, u,v = a,r, u,v, m,n
    gcd = b
    return gcd, x, y
```

Por exemplo,

```
print egcd(1432,123211)
```

7.2 Divisibilidade Modular

A chave privada é calculada pelo *inverso multiplicativo* na aritmética modular a partir da chave pública, tanto no algoritmo `RSA` quanto no algoritmo `ElGamal`.

Acabamos de apreender como calcular o maior divisor comum pelo Algoritmo de Euclides Estendido; agora apreendamos como ele é usado para calcular este *inverso multiplicativo.*

Unidades. Enquanto em $\mathbb{Q}$ podemos dividir por qualquer número (exceto 0), em $\mathbb{Z}$ unicamente por ± 1! Os números pelos quais pode se dividir são chamadas *invertíveis* ou *unidades.* A quantidade de números invertíveis em $\mathbb{Z}/m\mathbb{Z}$ depende do módulo m. De modo grosso, quanto menos fatores primos em m, tanto mais unidades em $\mathbb{Z}/m\mathbb{Z}$.

Caracterização. **Definição**. *O elemento $\bar{x}$ em $\mathbb{Z}/m\mathbb{Z}$ é uma **unidade** (ou **invertível**) se existe $\bar{y}$ em $\mathbb{Z}/m\mathbb{Z}$ tal que $\bar{y}\bar{x} = 1$. O elemento $\bar{y}$ é o **inverso** de $\bar{x}$ e denotado por $\bar{x}^{-1}$. O conjunto das unidades (em que podemos multiplicar e dividir) é denotado por*

$$(\mathbb{Z}/m\mathbb{Z})^* := \{ \text{ as unidades em } \mathbb{Z}/m\mathbb{Z}\}.$$

***A função totiente de Euler** Φ é*

$$m \mapsto \#(\mathbb{Z}/m\mathbb{Z})^*;$$

isto é, dado m, conta quantas unidades $\mathbb{Z}/m\mathbb{Z}$) tem.

Observações:

- O elemento neutro da adição 0 nunca é uma unidade. (Mas possivelmente, dependendo de m, todos os outros elementos.)
- Enquanto em $\mathbb{Z}$ as únicas unidades são ± 1, em $\mathbb{Z}/m\mathbb{Z}$ possivelmente todos os seus elementos, exceto 0, são.

Exemplos:

- No relógio, isto é, para $\mathbb{Z}/12\mathbb{Z}$ a multiplicação $v \cdot h$ de uma hora h por v corresponde a iterar v vezes o caminho percorrido pelo indicador em h (a partir de $0 = 12$.) Observamos que para $h = 1, 5, 7$ e 11 existe uma iteração do caminho que leva o indicador a 1 (mais exatamente, feita 1, 5, 7 e 11 vezes), enquanto para todos os outros números, esta iteração leva o indicador a 0. Estas possibilidades são mutuamente exclusivas, e concluímos

  $$(\mathbb{Z}/12\mathbb{Z})^* = \{1, 5, 7, 11\}.$$

  Isto é, $\Phi(12) = 11$.

- Nos dias da semana, isto é, para $\mathbb{Z}/7\mathbb{Z}$, obtemos

  $$(\mathbb{Z}/7\mathbb{Z})^* = \{1, 2, 3, 4, 5, 6\}.$$

  Isto é, o número de unidades é tão grande quanto possível, isto é, $\Phi(7) = 6$, todos os números exceto 0.

- Para $\mathbb{Z}/4\mathbb{Z}$, a tabela de multiplicação

*	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	0	2
3	0	3	2	1

  mostra

  $$(\mathbb{Z}/4\mathbb{Z})^* = \{1, 3\}$$

  pois $1 \cdot 1 = 1$ e $3 \cdot 3 = 1$ em $\mathbb{Z}/4\mathbb{Z}$. Ao contrário, $2 \cdot 2 = 0$ em $\mathbb{Z}/4\mathbb{Z}$, em particular 2 não é uma unidade. (Mas *um divisor de zero*; de fato, cada elemento em $\mathbb{Z}/m\mathbb{Z}$ é ou uma unidade, ou um divisor de zero.) Logo $\Phi(4) = 2$.

- Para $\mathbb{Z}/5\mathbb{Z}$, a tabela de multiplicação

*	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4

*	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

revela as unidades $(\mathbb{Z}/5\mathbb{Z})^* = \{1,2,3,4\}$.

Proposição. *Seja m em $\mathbb{N}$ e $\bar{x}$ em $\mathbb{Z}/m\mathbb{Z}$, isto é, $\bar{x}$ em $\{0,1,\ldots,m-1\}$. O número $\bar{x}$ é uma unidade em $\mathbb{Z}/m\mathbb{Z}$ se, e tão-somente se,* $\mathrm{mdc}(x,m) = 1$.

Demonstração: Observamos que cada divisor comum de $\bar{x}$ e m divide cada soma de múltiplos $s = u\bar{x} + vm$ de $\bar{x}$ e m; em particular, se $s = 1$, então o maior divisor comum de $\bar{x}$ e m é 1.

Pelo Algoritmo de Euclides Estendido, há u e v em $\mathbb{Z}$ tais que

$$u\bar{x} + vm = \mathrm{mdc}(\bar{x}, m).$$

Logo, pela observação acima, $\mathrm{mdc}(\bar{x}, m) = 1$ se, e tão-somente se, há u em $\mathbb{Z}$ tal que

$$u\bar{x} \equiv 1 \bmod m.$$

Equivalentemente,

$$\bar{u}\bar{x} = 1 \quad \text{em } \mathbb{Z}/m\mathbb{Z}.$$

para $\bar{u}$ em $\mathbb{Z}/m\mathbb{Z}$ o resto de u dividido por m. Isto é, $\bar{x}$ é uma unidade em $\mathbb{Z}/m\mathbb{Z}$ cujo inverso é $\bar{x}^{-1} = \bar{u}$.

Observação. Concluímos que para x em $\{0,\ldots,m-1\}$ com $\mathrm{mdc}(x,m) = 1$, obtemos pelo Algoritmo de Euclides Estendido u e v em $\mathbb{Z}$ tais que

$$ux + vm = 1$$

O inverso x^{-1} de x em $\mathbb{Z}/m\mathbb{Z}$ é dado pelo resto de u dividido por m.

Em `Python`, podemos então calcular o inverso de a em $\mathbb{Z}/m\mathbb{Z}$ por

```
def ModInverse(a, m):
    if gcd(a, m) != 1:
        return None # no mod. inverse exists if a and m not rel. prime
    else
        gcd, x, y = egcd(a,m)
        return x % m
```

Corpos Finitos. **Corolário 7.1**. *Se p é um número primo, então*

$$(\mathbb{Z}/p\mathbb{Z})^* = \{1, ..., p-1\}.$$

Isto é, todos os elementos, exceto 0, são unidades. Em particular, $\Phi(p) = p-1$.

Demonstração: Se p é primo, então $\mathrm{mdc}(x,p) = 1$ para qualquer x em $\{1,...,p-1\} = \mathbb{Z}/p\mathbb{Z}$.

Um anel cujos elementos, exceto 0, são todos unidades é um *corpo*. Isto é, em um corpo, dado um elemento a, além de $-a$, o inverso de a sob *adição*, sempre há a^{-1}, o inverso de a sob *multiplicação*. Em vez de ba^{-1}, se escreve também b/a. Em outras palavras, além de podermos subtrair, podermos dividir. Os exemplos mais comuns são $\mathbb{Q}$, $\mathbb{R}$ e $\mathbb{C}$, todos sendo infinitos.

O corpo com p elementos (isto é, $\mathbb{Z}/p\mathbb{Z}$) é denotado por $\mathbb{F}_p$.

Observação. Para um módulo do tipo $m = pq$ para p e q primo, vale pelo *Teorema Chinês dos Restos* $\mathbb{Z}/m\mathbb{Z} = \mathbb{Z}/p\mathbb{Z} \times \mathbb{Z}/q\mathbb{Z}$, e conforme

$$\mathbb{Z}/m\mathbb{Z}^* = \mathbb{Z}/p\mathbb{Z}^* \times \mathbb{Z}/q\mathbb{Z}^*.$$

Em particular, como os dois fatores são corpos, $\phi(m) = (p-1)(q-1)$. Esta observação é (implicitamente, neste curso) usada no algoritmo RSA.

Existência da raiz primitiva para um módulo primo. **Teorema**. *(Existência da **raiz primitiva** em um corpo finito)* Se p é um número primo, então existe α em $\mathbb{F}_p^* = \{1, 2, ..., p-1\}$ tal que

$$\mathbb{F}_p^* = \{\alpha, \alpha^2, ...\}.$$

Demonstração: Como (pela iterada divisão com resto) um polinômio de grau m tem $\leq m$ raízes, e $x^m = 1$ se, e tão-somente se, $P(x) = 0$ para $P(X) = X^m - 1$, vale

$$\#\{x \text{ em } \mathbb{F}_p^* \text{ tal que } x^m = 1\} \leq m \quad (\dagger)$$

Mostramos que qualquer tal grupo G é *cíclico*, isto é, gerado por um elemento: Seja x um elemento em G de *ordem* (= o menor número $m > 0$ tal que $x^m = 1$) máxima. Se $m < \#G$, então há por (†) um y em G cuja ordem não divide m. Então a ordem de $z = xy$ é $> m$, em *contradição à escolha* de m.

Existência da Raiz Primitiva para um módulo qualquer. Para explicarmos porque usualmente só módulos primos são usados para Diffie-Hellman, estabeleçamos para quais módulos m as unidades de $\mathbb{Z}/m\mathbb{Z}$ têm um único gerador:

Proposição. Se p é um número primo e e um inteiro positivo, então α em $\mathbb{Z}/p^e\mathbb{Z}$ é uma unidade se, e tão-somente se, p não divide α.

Demonstração: O número α em $A = \mathbb{Z}/p^e\mathbb{Z}$ é uma unidade se, e tão-somente se, a multiplicação $\alpha\cdot$ sobre A é sobrejetora, isto é, todo elemento em A é produto de α. Como A é finito, pelo princípio do pombal, uma aplicação sobre A é sobrejetora se, e tão-somente se, é injetora, isto é, manda dois argumentos diferentes a dois valores diferentes.

Como a multiplicação $\phi = \alpha\cdot$ sobre A satisfaz $\phi(a \cdot b) = \phi(a) \cdot \phi(b)$, ela é injetora se, e tão-somente se, para todo x em A, se $\phi(x) = 0$, então $x = 0$.

Por contraposição, existe x em A não-nulo com $\phi(x) = \alpha x = 0$, isto é, p^e não divide x, mas divide αx se, e tão-somente se, p divide α. *q.e.d.*

Em particular, notemos o valor da Função Totiente de Euler

$$\Phi(p^e) = \#\mathbb{Z}/p^e\mathbb{Z}^* = (p-1)p^{e-1}.$$

Recordemo-nos de que a *ordem* e de um elemento g do grupo $\mathbb{Z}/m\mathbb{Z}^*$ é o mínimo expoente e tal que $g^e = 1$.

Teorema *(Existência da **raiz primitiva** para um módulo que é a potência de* 2*)* Seja e um inteiro positivo. Existe α em $\mathbb{Z}/2^e\mathbb{Z}$ tal que

$$\mathbb{Z}/2^e\mathbb{Z}^* = \{\alpha, \alpha^2, \ldots\}.$$

se, e tão-somente se, $e = 0, 1$ ou 2.

Demonstração: Se $e = 0, 1$ ou 2, então 0, 1 e 3 geram $\mathbb{Z}/2^e\mathbb{Z}^*$. Se $e = 3$, então, nenhum dos dois números $x = 3$ ou 7 tais que $x \equiv 3 \bmod 2^2$ geram $\mathbb{Z}/2^3\mathbb{Z}$; logo, não existe gerador. Se $e > 3$ e $g^x \equiv a \bmod 2^e$, então $g^x \equiv a \bmod 2^3$. Logo, tampouco existe gerador. *q.e.d.*

Teorema *(Existência da **raiz primitiva** para um módulo que é a potência de um número primo ímpar)* Se p é um número primo > 2 e e um inteiro positivo, então existe α em $\mathbb{Z}/p^e\mathbb{Z}$ tal que

$$\mathbb{Z}/p^e\mathbb{Z}^* = \{\alpha, \alpha^2, \ldots\}.$$

Demonstração: O caso $e = 1$ já foi demonstrado. Mostremos primeiro o caso $e = 2$: Seja $g = g_1$ um gerador de $\mathbb{Z}/p\mathbb{Z}$ e $g_2 = g + kp$ em $\mathbb{Z}/p^2\mathbb{Z}$. Temos

$$g_2^{p-1} \equiv g^{p-1} \equiv 1 \bmod p,$$

isto é, $g_2^{p-1} \equiv 1 + rp \bmod p^2$ para algum r inteiro.

Se $r \neq 0$, então a ordem de $g_2 > p-1$. Como $\#\mathbb{Z}/p^2\mathbb{Z}^* = (p-1)p$, pelo Teorema de Lagrange, a ordem de g_2 é $(p-1)p$; isto é, g_2 gera $\mathbb{Z}/p^2\mathbb{Z}^*$.

Temos

$$g_2^{p-1} = (g + kp)^{p-1} \equiv 1 \bmod p^2$$

se, e tão-somente se,

$$(g + kp)^p \equiv g + kp \bmod p^2.$$

Como $(g + kp)^p = g^p \bmod p^2$, se, e tão-somente se,

$$g^p - g \equiv kp \bmod p^2.$$

Isto é, se $k \not\equiv [g^p - g]/p \bmod p^2$, então g_2 gera $\mathbb{Z}/p^2\mathbb{Z}$.

Observemos que se

$$g^t \equiv 1 + kp^s \bmod p^{s+1},$$

então

$$g^{pt} \equiv 1 + kp^{s+1} \bmod p^{(s+1)+1}.$$

Logo, se a ordem de g módulo p^{s+1} é $> t$, então a ordem de g módulo módulo $p^{(s+1)+1}$ é $> tp$. Em particular, se g gera $\mathbb{Z}/p^{s+1}\mathbb{Z}$, isto é, a sua ordem é $(p-1)p^s$ em $\mathbb{Z}/p^{s+1}\mathbb{Z}$, então a ordem de g é $(p-1)p^{s+1}$ em $\mathbb{Z}/p^{(s+1)+1}\mathbb{Z}$, isto é, g gera $\mathbb{Z}/p^{(s+1)+1}\mathbb{Z}$.

Por indução, o primeiro passo o caso $s = 1$ já demonstrado acima, obtemos que se g gera $\mathbb{Z}/p^2\mathbb{Z}^*$, então gera $\mathbb{Z}/p^s\mathbb{Z}^*$ para todo s. *q.e.d.*

Teorema Chinês dos Restos Sejam n e m inteiros. Se n e m tem nenhum divisor comum, então a aplicação natural

$$\mathbb{Z}/nm\mathbb{Z} \to \mathbb{Z}/n\mathbb{Z} \times \mathbb{Z}/m\mathbb{Z}$$

dada por $x \mapsto (x \bmod n, x \bmod m)$ é bijetora.

Demonstração:

Se $x \equiv 0 \bmod n$ e $x \equiv 0 \bmod m$, então, como m e n têm nenhum divisor comum, $x \equiv 0 \bmod mn$; logo, a aplicação é injetora.

Seja (y, z) em $\mathbb{Z}/n\mathbb{Z} \times \mathbb{Z}/m\mathbb{Z}$. Como m e n têm nenhum divisor comum, existem, pelo Algoritmo de Euclides Estendido, inteiros a e b tais que $am + bn = 1$. Logo, $x = ay + bz$ satisfaz $x \equiv y \bmod m$ e $x \equiv z \bmod n$. *q.e.d.*

Corolário. O grupo $\mathbb{Z}/m\mathbb{Z}$ é *cíclico*, isto é, gerado por um único elemento, se, e tão-somente se,

- ou m em $\{1, 2, 4\}$,
- ou $m = p^e$ ou $2p^e$ para um primo $p > 2$.

Demonstração: Pelo Teorema Chinês dos Restos, se $m = p_1^{e_1} \cdots p_2^{e_n}$ para primos $p_1, \ldots, p_n$, então

$$\phi\colon \mathbb{Z}/m\mathbb{Z} \to \mathbb{Z}/p_1^{e_1}\mathbb{Z} \times \cdots \times \mathbb{Z}/p_n^{e_n}\mathbb{Z}$$

é bijetora e respeita +, isto é, $f(x + y) = f(x) + f(y)$; logo respeita $\cdot$, isto é, $f(x \cdot y) = f(x) \cdot f(y)$. Em particular, $\phi(1) = 1$, e para todo a em $\mathbb{Z}/m\mathbb{Z}^*$ existe b com $ab \equiv 1 \bmod m$ se, e tão-somente se, existem $b_1, \ldots, b_n$ tais que $ab_1 \equiv 1 \bmod p_1^{e_1}, \ldots, ab_n \equiv 1 \bmod p_n^{e_n}$. Consequentemente

$$\mathbb{Z}/m\mathbb{Z}^* \to \mathbb{Z}/p_1^{e_1}\mathbb{Z}^* \times \cdots \times \mathbb{Z}/p_n^{e_n}\mathbb{Z}^*$$

é bijetora. Logo, como $\mathbb{Z}/p^n\mathbb{Z}$ é ciclico para um primo $p > 2$, o produto é cíclico se, e tão-somente se, existe um único fator não-trivial.

Pequeno Teorema de Fermat. O Pequeno Teorema de Fermat é uma observação útil quanto à ciclicidade da multiplicação nos anéis finitos $\mathbb{Z}/p\mathbb{Z}$ para p um número primo:

Como $(\mathbb{Z}/p\mathbb{Z})^* = \{1, \ldots, p-1\}$ é finito (de cardinalidade $p - 1$), para qualquer número x diferente de 0, necessariamente $x^0, x^1, x^2, \ldots, x^p$ terá uma repetição, $x^n = x^{n+m}$ para algum expoente n e algum $m \leq p - 1$. Como em $\mathbb{Z}/p\mathbb{Z}$ todo número exceto 0 é invertível, podemos dividir por x^{n-1}; isto é, há $m \leq p - 1$ tal que $1 = x^m$. Pelo Teorema de Lagrange, veremos que não somente $m \leq p - 1$, mas mais exatamente $m \mid p - 1$.

Este teorema nos servirá, por exemplo,

- para detectar primos, e
- para encontrar o atalha da função alçapão no algoritmo RSA.

Observação. No entanto, não quer dizer que possivelmente já $x^m = 1$ para $m < p - 1$. Com efeito, há muitos X em $\mathbb{Z}/p\mathbb{Z}$ para que isto vale: Para qualquer x em $\mathbb{Z}/p\mathbb{Z}$ e divisor d de $p - 1$, a potência $X := x^d$ satisfaz $x^m = 1$ já para $m = (p-1)/d < p - 1$. No mesmo tempo, acabamos de demonstrar que sempre há uma raiz primitiva, um *gerador* x_0 de $\mathbb{Z}/m\mathbb{Z}^*$; isto é, necessariamente $x_0^m \neq 1$ para todo $m < p - 1$.

Quanto à segurança criptográfica, o que pesa é o maior fator primo q em $p - 1$; por isso, os números primos queridos são da forma $p = 2q + 1$ com q primo, os *primos seguros*. Para facilitar os cálculos (sem perda de segurança),

- em vez de trabalhar com uma *raiz primitiva*, isto é, um x_0 tal que $x_0^m = 1$ para $m = p - 1$ pela primeira vez,
- trabalha-se com um x_0 tal que $x_0^m = 1$ para $m = q$ pela primeira vez.

Figura 72: Juiz e Matemático leigo francês Pierre de Fermat (†1655)

Teorema. *(Pequeno Teorema de Fermat) Se p é um número primo, então, para qualquer número inteiro a,*

- ou $a^{p-1} \equiv 0 \bmod p$ se $p \mid a$,
- ou $a^{p-1} \equiv 1 \bmod p$ se $p \nmid a$.

Demonstração: Suponhamos que $p \mid a$. Vale $p \mid a$ se, e tão-somente se, $\bar{a} = 0$, e então $a^{\bar{p-1}} = \bar{a}^{p-1} = 0^{p-1} = 0$ em $\mathbb{Z}/m\mathbb{Z}$.

Suponhamos que $p \nmid a$. Como o grupo $(\mathbb{Z}/p\mathbb{Z})^*$ é finito, há n e $n+m$ tais que $\bar{a}^n = \bar{a}^{n+m}$. Como p é primo, $\#(\mathbb{Z}/p\mathbb{Z})^* = p-1$ pelo Corolário 7.1. Em particular, podemos dividir por todo número exceto 0, em particular, por a, e obtemos $\bar{a}^m = 1$ com $m \leq p-1$. Pelo *Teorema de Lagrange*, $m \mid p-1$, isto é, $p-1 = md$ para um divisor d. Logo $\bar{a}^{p-1} = \bar{a}^{md} = (\bar{a}^m)^d = 1^d = 1$.

Corolário. *Se p é um número primo, então, para qualquer número inteiro a vale* $a^p \equiv a \bmod p$.

Com efeito, o Corolário equivale ao Pequeno Teorema de Fermat, porque se p é primo, então pode se dividir por qualquer número a (diferente de zero) em $\mathbb{Z}/p\mathbb{Z}$ (diz-se que $\mathbb{Z}/p\mathbb{Z}$ é um *corpo*).

Para demonstrar o Teorema de Lagrange, precisamos da noção de um *grupo*:

Definição. Um ***grupo*** G *é um conjunto com uma operação binária* $\cdot\colon \mathrm{G} \times \mathrm{G} \to \mathrm{G}$ *tal que*

- (existência da identidade) há e em G tal que $eg = ge = g$ para todos os g em G,
- (existência do inverso) para todos os g em G há g^{-1} em G tal que $gg^{-1} = g^{-1}g = e$, e
- (associatividade) para todos os g, h e i em G vale $(gh)i = g(hi)$.

Teorema. *(Lagrange) Seja* G *um grupo. Se* H *é um subgrupo de* G*, então* #H | #G.

Demonstração: Define a relação $\sim$ sobre G por $g' \sim g''$ se há h em H tal que $g'' = hg'$. Como H é um grupo, ela é transitiva e uma relação de equivalência. Logo G é a união disjunta de classes de equivalência.

Seja g em uma tal classe de equivalência C. Por definição de $\sim$, a aplicação $g\colon \mathrm{H} \to \mathrm{C}$ é sobrejetora. Como g é invertível (por g^{-1}), ela é injetora, então uma bijeção.

Concluímos que todas as classes de equivalência têm cardinalidade #H e, pela união disjunta, que #H | #G.

7.3 Detectar Primos

Recordemo-nos do Teorema de Euclides que asserta que haja números primos **arbitrariamente grandes** (por exemplo, com ≥ 2048 dígitos binários para o RSA):

Teorema de Euclides. Existe uma **infinitude** de números primos.

Demonstração: Suponhamos o contrário, que haja só um número finito $p_1, ..., p_n$ de números primos. Considere

$$q = p_1 ... p_n + 1.$$

Como q é maior que $p_1, \ldots, p_n$, não é primo. Seja então p um número primo que divida q. Pela hipótese, p em $\{p_1, ..., p_n\}$. Mas por definição q tem resto 1 divido por qualquer $p_1, \ldots, p_n$.

Contradição! Logo, não existe um maior número primo. q.e.d.

Exemplos de Grandes Números Primos. Marin Mersenne (Oizé, 1588 — Paris, 1648) foi um padre mínimo francês que tentou encontrar, sem êxito, uma fórmula para todos os números primos. Motivada por uma carta de Fermat em que lhe sugeriu que todos os números $2^{2^p} + 1$, os *Números de Fermat* fossem primos, Mersenne estudou os números

$$2^p - 1 \quad \text{para } p \text{ primo .}$$

Em 1644 publicou o trabalho *Cogita physico-mathematica* que afirma que estes são primos para

$$p = 2, 3, 5, 7, 13, 17, 19, 31 \text{ e } 127.$$

(e erroneamente incluiu $p = 63$ e $p = 257$). Só um computador conseguiu mostrar em 1932 que $2^{257} - 1$ é composto.

Os números primos de Mersenne, da forma $2^p - 1$ para p primo, conhecidos são

2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607, 1279, 2203, 2281, 3217, 4253, 4423, 9689, 9941, 11213, 19937, 21701, 23209, 44497, 86243, 110503, 132049, 216091, 756839, 859433, 1257787, 1398269, 2976221, 3021377, 6972593, 13466917, 20996011, 24036583, 25964951, 30402457, 32582657, 37156667, 42643801, 43112609, 57885161, 74207281, 77232917 e 82589933

O número primo

$$2^{82\,589\,933} - 1$$

tem 24 862 048 algarismos. Foi encontrado no dia 8 de dezembro 2018 e é até hoje o **maior número primo conhecido**.

O `CrypTool 1`, na entrada do menu `Indiv. Procedures -> Number Theory Interactive -> Compute Mersenne Numbers` permite calcular uns números primos de Mersenne.

Testes. Um teste rápido se o número natural n é composto é o **Pequeno Teorema de Fermat** (formulado pela sua contra-posição): Se há um número natural a tal que

$$a^n \not\equiv a \bmod n$$

então n é composto.

Mas a implicação inversa, isto é, se n é primo, então há um número natural a tal que $a^n \not\equiv a \bmod n$, não vale: Há números n (que se chamam **Números de Carmichael**) que são compostos mas, para todo número natural a,

$$a^n \equiv a \bmod n.$$

O menor tal número n é 561 (que é divisível por 3).

O crivo de Eratóstenes. O algoritmo mais simples para verificar se um número é primo ou não é o crivo de Eratóstenes (285 – 194 a.C.).

Para exemplificá-lo, vamos determinar os números primos entre 1 e 30.

Inicialmente, determina-se o maior número pelo que dividiremos para verificar se o número é composto; é a raiz quadrada da cota superior arredondada para baixo. No caso, a raiz de 30, arredondada para baixo, é 5.

1. Crie uma lista de todos os números inteiros de 2 até o valor da cota: 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29 e 30.
2. Encontre o primeiro número da lista. Ele é um número primo, 2.
3. Remova da lista todos os múltiplos de 2 até 30: 2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27 e 29.

Figura 73: Eratóstenes, o terceiro bibliotecário-chefe da Biblioteca de Alexandria

O próximo número da lista é primo. Repita o procedimento:

- No caso, o próximo número da lista é 3. Removendo os seus múltiplos, a lista fica: 2, 3, 5, 7, 11, 13, 17, 19, 23, 25 e 29.
- O próximo número, 5, também é primo: 2, 3, 5, 7, 11, 13, 17, 19, 23 e 29.

Conforme determinado inicialmente, 5 é o último número pelo qual dividimos. A lista final 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 contém só números primos.

Segue uma implementação em Python:

```
def primeSieve(sieveSize):
    # Returns a list of prime numbers calculated using
    # the Sieve of Eratosthenes algorithm.

    sieve = [True] * sieveSize
    sieve[0] = False # zero and one are not prime numbers
```

```
    sieve[1] = False
    # create the sieve
    for i in range(2, int(math.sqrt(sieveSize)) + 1):
        pointer = i * 2
        while pointer < sieveSize:
            sieve[pointer] = False
            pointer += i
    # compile the list of primes
    primes = []
    for i in range(sieveSize):
        if sieve[i] == True:
            primes.append(i)
    return primes
```

O Teste Determinista AKS. O **teste de AKS** determina em tempo polinomial se n é composto ou primo (mais exatamente, em tempo $O(d)^6$ onde d = o número de dígitos d [binários] de n). Na prática, basta normalmente o **Teste de Miller-Rabin** probabilista que garante muito mais *testemunhas* (= números a que provam se n é composto ou não) do que o Pequeno Teorema de Fermat.

Com efeito, quando comparamos a duração entre os dois algoritmos para verificar se um número é primo num computador com um processador Intel Pentium-4 de 2 GHz, obtemos

número primo	Miller-Rabin	AKS
7309	0.01	12.34
9004097	0.01	23 : 22.55
$2^31 - 1$	0.01	6 : 03 : 14.36

O CrypTool 1 oferece no Menu Individual Procedures -> RSA uma entrada para experimentar com diferentes algoritmos para detectar números primos.

O Teste Probabilista Miller-Rabin. Os testes simplistas, para saber se um número n é primo ou não, são ineficientes porque calculam os fatores de n. Em vez deles, para saber apenas se é primo ou não, há o Teste de Miller-Rabin.

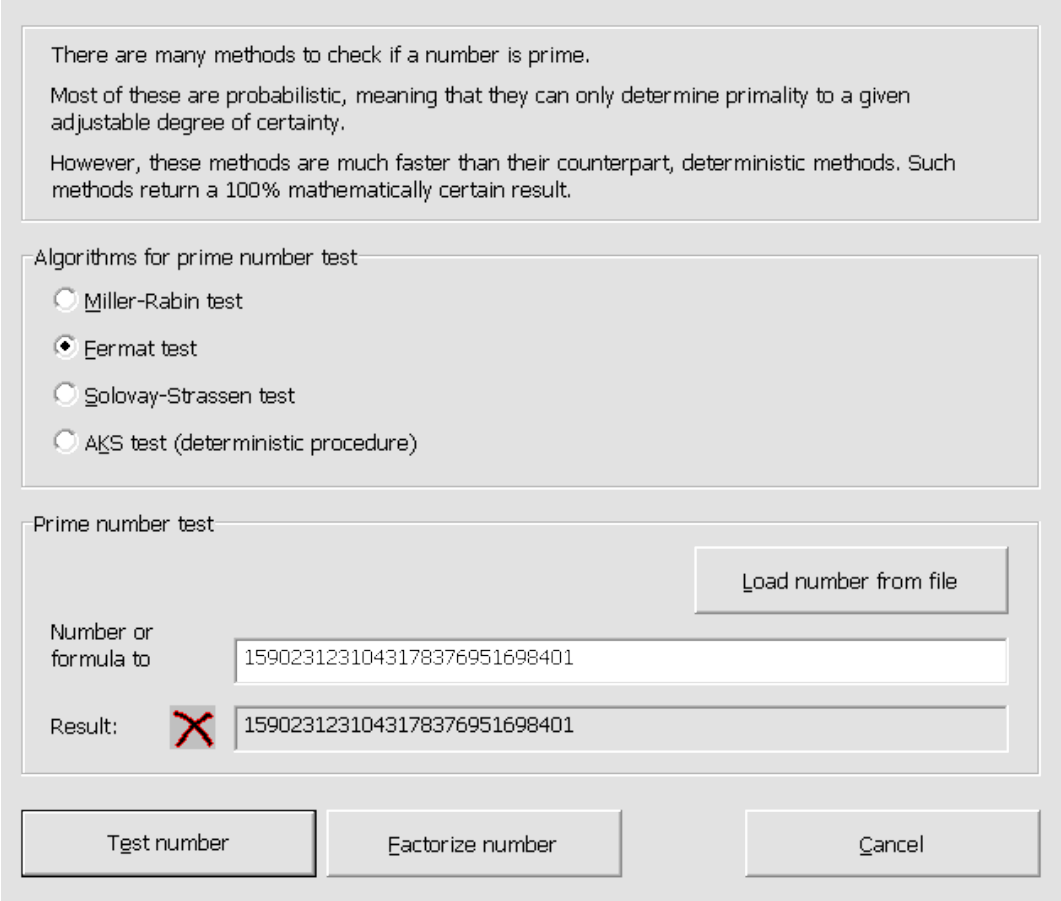


Figura 74: Os algoritmos para verificar se um número é primo no `CrypTool 1`

Após a sua demonstração, damos a sua contraposição; é nesta formulação que é aplicado.

O Teste de Miller-Rabin. Seja $p > 2$ um número primo, seja $n - 1 = 2^k q$ para números k e q (com q ímpar). Então, para qualquer número inteiro a indivisível por p vale

- ou $a^q \equiv 1$,
- ou existe r em $\{0, 1, \ldots, k-1\}$ tal que $a^{2^r q} \equiv -1 \bmod n$

Demonstração: Pelo Pequeno Teorema de Fermat

$$a^{p-1} = (a^d)^{2^k} \equiv 1 \bmod p$$

Extraindo iterativamente a raiz quadrada, obtemos

- ou $(a^q)^{2^r} \equiv 1 \bmod p$ para todo $r = 1, \ldots, k-1$; em particular $a^q \equiv 1 \bmod p$,
- ou existe r em $\{1, \ldots, k-1\}$ tal que $(a^q)^{2^r} \equiv -1 \bmod p$. *q.e.d.*

Se para um número ímpar, um número possivelmente primo, escrevemos $n-1 = 2^k q$, então pelo Teste de Fermat n não é primo se existe um inteiro a tal que $a^{2^k q} \not\equiv 1 \bmod n$. O Teste de Miller-Rabin explicita a condição $a^{q2^k} = (a^q)^{2^k} = ((a^q)^2)\cdots)^2 \not\equiv 1$:

O Teste de Miller-Rabin. *(Contraposição) Seja n ímpar e $n-1 = 2^k q$ para números k e q (com q ímpar). Um inteiro a relativamente primo a n é uma **Testemunha de Miller-Rabin** (para a divisibilidade) de n*, se

- $a^q \not\equiv 1 \bmod n$ e
- $a^{2q} a^{2^2 q}, \ldots, a^{2^{k-1} q} \not\equiv -1 \bmod n$.

Questão: Quais as chances que declaramos pelo Teste de Miller-Rabin acidentalmente um número primo, isto é, um número que é na verdade composto?

Teorema. *(Sobre a frequência das testemunhas)* Seja n ímpar e composto. Então pelo menos 75% dos números em $\{1, \ldots, n-1\}$ são Testemunhas de Miller-Rabin para n.

Logo, já depois de 5 tentativas a_1, a_2, ..., a_5 sem testemunha sabemos com uma chance $1/4^5 = 1/1024 < 0{,}1\%$, que o número é primo!

Implementação em Python. Vamos implementar

1. o algoritmo de Miller-Rabin,
2. um teste para um número primo, e
3. uma função para gerar números primos (grandes).

```
# Primality Testing with the Rabin-Miller Algorithm
# http://inventwithpython.com/hacking (BSD Licensed)

import random


def rabinMiller(num):
    # Returns True if num is a prime number.
```

```
    s = num - 1
    t = 0
    while s % 2 == 0:
        # keep halving s while it is even (and use t
        # to count how many times we halve s)
        s = s // 2
        t += 1

    for trials in range(5): # try to falsify num's primality 5 times
        a = random.randrange(2, num - 1)
        v = pow(a, s, num)
        if v != 1: # this test does not apply if v is 1.
            i = 0
            while v != (num - 1):
                if i == t - 1:
                    return False
                else:
                    i = i + 1
                    v = (v ** 2) % num
    return True

def isPrime(num):
    # Return True if num is a prime number. This function does a
    # quicker prime number check before calling rabinMiller().

    if (num < 2):
        return False # 0, 1, and negative numbers are not prime

    # About 1/3 of the time we can quickly determine if num is not
    # prime  by dividing by the first few dozen prime numbers.
    # This is quicker # than rabinMiller(), but unlike rabinMiller()
    # is not guaranteed to prove that a number is prime.
    lowPrimes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,
    47, 59,  61,  67,  71,  73,  79,  83,  89,  97,  101, 103, 107,
    109, 113, 127, 131, 137, 149, 151, 157, 163, 167, 173, 179, 181,
    191, 193, 197, 199, 211, 223, 227, 233, 239, 241, 251, 257, 263,
    269, 271, 277, 281, 283, 293, 307, 311, 313, 331, 337, 347, 349,
```

```
        353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 431, 433,
        439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509,
        523, 541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607,
        613, 617, 631, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691,
        701, 709, 719, 727, 739, 743, 751, 757, 761, 769, 773, 787, 797,
        809, 811, 821, 823, 827, 829, 853, 857, 859, 863, 877, 881, 883,
        887, 907, 911, 919, 929, 937, 941, 947, 967, 971, 977, 983, 991,
        997]

    if num in lowPrimes:
        return True

    # See if any of the low prime numbers can divide num
    for prime in lowPrimes:
        if (num % prime == 0):
            return False

    # If all else fails, call rabinMiller() to check if num is a prime.
    return rabinMiller(num)

def generateLargePrime(keysize = 1024):
    # Return a random prime number of keysize bits in size.
    while True:
        num = random.randrange(2**(keysize-1), 2**(keysize))
        if isPrime(num):
            return num
```

8 Algoritmos Assimétricos Clássicos

Apresentamos

1. o algoritmo RSA que se baseia na dificuldade de computar a fatoração em números primos, e
2. dois algoritmos que se baseiam nãoo protocolo de Diffie-Hellman (para construir uma chave secreta mútua através de um canal inseguro),

    1. ElGamal e
    2. ECC, criptografia com curvas elípticas

Ambos os algoritmos do tipo Diffie-Hellman, os algoritmosElGamal e ECC criam (em contraste ao RSA) uma chave efêmera para cada cifração e assinatura. É importante que esta chave seja imprevisível; caso contrário, esta informação permite a deduzir a chave privada. (Por exemplo, um aplicativo Bit Coin sob Android teve esta falha.)

Isto é, é preciso garantir que haja suficientemente entropia (isto é, desordem no sistema) na criação desta chave efêmera, usando como parâmetros particulares

- a chave secreta, e
- (um hash criptográfico d') a mensagem (que será cifrada ou assinada).

8.1 O Algoritmo RSA

Apresentamos o algoritmo RSA de Rivest, Shamir, e Adleman (1978) que cria

- uma chave pública para cifrar, e
- uma chave privada para decifrar.

Em comparação ao protocolo de Diffie-Hellman, ele tem a vantagem que é completamente assimétrico:

- não preciso de construir uma chave secreta mútua (e logo não expõe a chave secreto em dois lugares mas em um lugar só),

- mas um único correspondente tem acesso à chave secreta. Contudo, neste caso a comunicação é cifrada só em direção do dono da chave secreta.

  Para cifrar em ambas as direções,

  - ou cada correspondente cria uma chave assimétrica RSA,
  - ou o outro correspondente cifra e manda uma chave simétrica.

As chaves para cifrar e decifrar serão construídas via a *Fórmula de Euler* que por seu turno se baseia no *Pequeno Teorema de Fermat.*

Fórmula de Euler. **Pequeno Teorema de Fermat**. *Se p é um número primo, então, para qualquer número inteiro a,*

- ou $a^{p-1} \equiv 0 \bmod p$ se $p \mid a$,
- ou $a^{p-1} \equiv 1 \bmod p$ se $p \nmid a$.

Em particular, $a^p = a^{(p-1)+1} = a^{p-1}a = a$ para todo inteiro a. Se $p \nmid a$, então

- vale $a^{1+(p-1)} = a^1 \cdot a^{p-1} \equiv a^1 \cdot 1 = a \bmod p$,
- vale $a^{1+2(p-1)} = a^1 \cdot a^{2\cdot(p-1)} = a^1 \cdot (a^{p-1})^2 \equiv a^1 \cdot 1^2 = a \bmod p$,
- vale $a^{1+3(p-1)} = a^1 \cdot a^{3\cdot(p-1)} = a^1 \cdot (a^{p-1})^3 \equiv a^1 \cdot 1^3 = a \bmod p$, ...

Isto é, se o expoente $m \equiv 1 \bmod p-1$, então $a^m \equiv a \bmod p$.

Em particular, se $m = \mathrm{E}d$, isto é, E e d são tais que $\mathrm{E}d \equiv 1 \bmod p-1$, em outras palavras, $d \equiv 1/\mathrm{E} \bmod p-1$ então

$$a^{\mathrm{E}d} = (a^{\mathrm{E}})^d \equiv a \bmod p,$$

isto é,

$$a^d = a^{1/\mathrm{E}} = \sqrt[\mathrm{E}]{a}!$$

Isto é, a computação da E-ésima raiz $\sqrt[\mathrm{E}]{\cdot}$ iguala à da d-ésima potência $\cdot^d$, um grande atalho computacional!

Exemplo. Por exemplo, para $p = 5$, $\mathrm{E} = 3$ e $d = 3$ temos $p - 1 = 4$ e $\mathrm{E}d = 9 \equiv 1 \bmod 5$. Por exemplo, para $a = 2$ vale

$$2^3 = 8 \equiv 3 \bmod 5 \quad \text{e} \quad 3^3 = 27 \equiv 2 \bmod 5$$

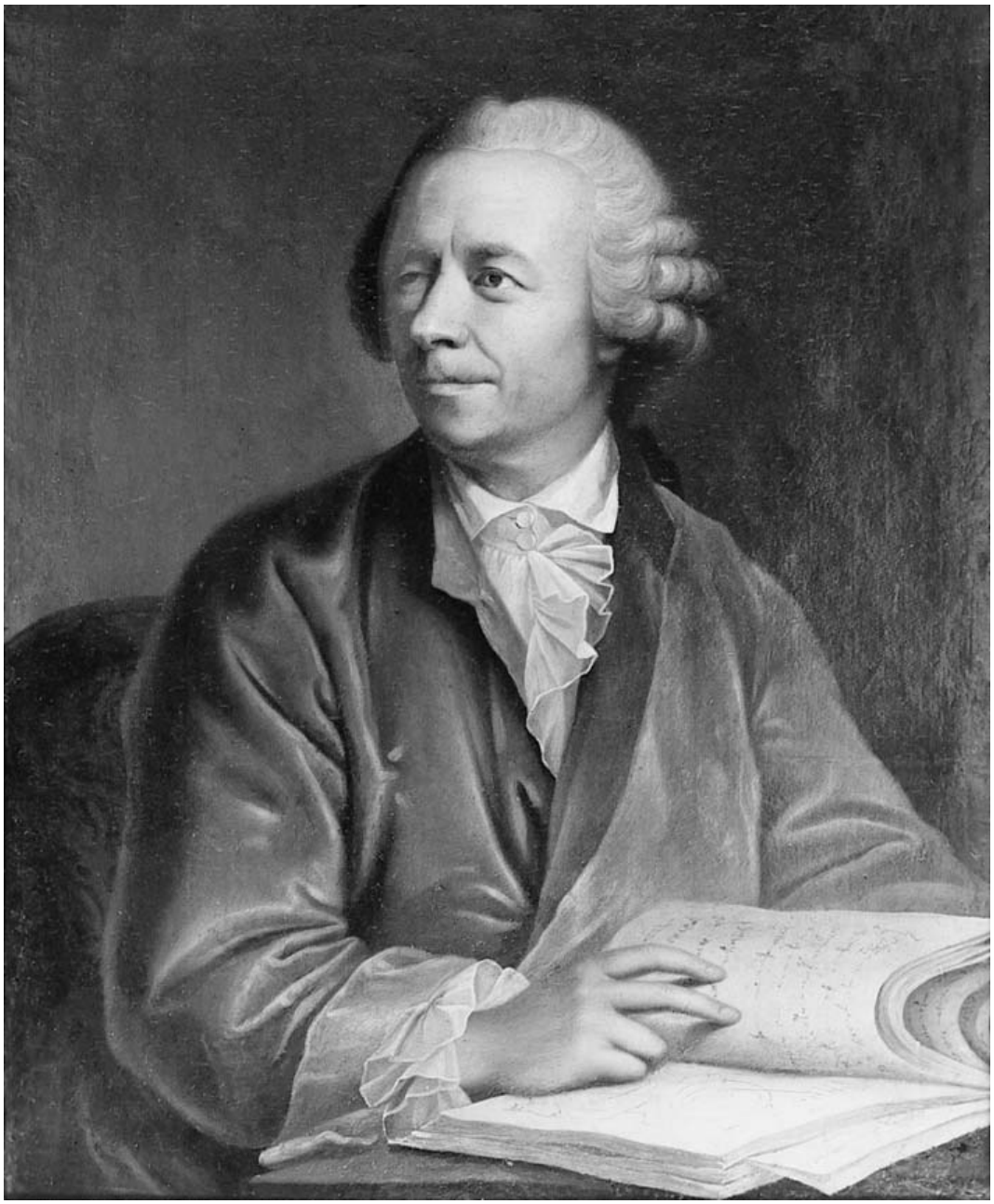


Figura 75: O Matemático suíço Leonhard Euler (1707 – 1783)

Esta computação de $\sqrt[E]{\cdot}$ por $\cdot^d$ é o atalho do RSA. Porém, para ser secreto, precisamos dificultar a computação de d a partir de E. Dado p e E, o número d tal que $Ed \equiv 1 \bmod p-1$ é diretamente calculado pelo algoritmo de Euclides.

No RSA, ao invés de um singelo primo p, usamos um produto $N = pq$ de dois números primos diferentes p e q que serão *desconhecidos*, cujo conhecimento é porém necessário (e dificílimo) para calcular d! Veremos primeiro como adaptar o *Pequeno Teorema de Fermat*, para um módulo p ao módulo $N = pq$, a **Fórmula de Euler**:

Teorema. *(Fórmula de Euler)* Sejam p e q números primos diferentes. Se a é divisível nem por p, nem por q, então

$$a^{(p-1)(q-1)} \equiv 1 \bmod pq.$$

Demonstração: Pelo pequeno teorema de Fermat,

$$a^{(p-1)(q-1)} = (a^{p-1})^{q-1} \equiv 1^{q-1} = 1 \bmod p$$

e

$$a^{(q-1)(p-1)} = (a^{q-1})^{p-1} \equiv 1^{p-1} = 1 \bmod q$$

isto é, p e q dividem $a^{(p-1)(q-1)} - 1$. Como p e q são primos diferentes, pq divide $a^{(p-1)(q-1)} - 1$, isto é, $a^{(p-1)(q-1)} \equiv 1 \bmod pq$.

Para números primos diferentes p e q, denote

$$N := pq \quad \text{e} \quad \phi(N) := (p-1)(q-1).$$

Observação. Recordemo-nos de que a função totem de Euler ϕ de Seção 7.2 que conta o número de unidades em $\mathbb{Z}/N\mathbb{Z}$. Verifica-se que $\Phi(N) = (p-1)(q-1)$.

Pela Fórmula de Euler $a^{\phi(N)} \equiv 1 \bmod N$. Logo, como para o módulo $p-1$ em vez de $\phi(N)$,

- vale $a^{1+\phi(N)} = a^1 \cdot a^{\phi(N)} \equiv a^1 \cdot 1 = a \bmod N$,
- vale $a^{1+2\phi(N)} = a^1 \cdot a^{2\cdot\phi(N)} = a^1 \cdot (a^{\phi(N)})^2 \equiv a^1 \cdot 1^2 = a \bmod N$,
- vale $a^{1+3\phi(N)} = a^1 \cdot a^{3\cdot\phi(N)} = a^1 \cdot (a^{\phi(N)})^3 \equiv a^1 \cdot 1^3 = a \bmod N$, ...

e geralmente:

Corolário. (Radiciação mod N) Sejam p e q números primos diferentes. Para qualquer expoente n tal que

$$n \equiv 1 \bmod \phi(\mathrm{N})$$

vale

$$a^n \equiv a \bmod \mathrm{N} \quad \text{para todo inteiro } a.$$

Demonstração: Como $n \equiv 1 \bmod (p-1)(q-1)$, isto é, há ν tal que $n - 1 = \nu(p-1)(q-1)$, vale pela Fórmula de Euler,

$$a^n = a^{\nu(p-1)(q-1)+1} = (a^{(p-1)(q-1)})^\nu \cdot a \equiv 1^\nu \cdot a = a \bmod \mathrm{N}.$$

Observação (crucial para o algoritmo RSA*).* Se $m \equiv 1 \bmod \phi(\mathrm{N})$, então pela *Fórmula de Euler* $a^m \equiv a \bmod \mathrm{N}$, isto é, a potenciação é a identidade,

$$\cdot^m \equiv \mathrm{id} \bmod \mathrm{N}$$

Em particular, se $m = \mathrm{E}d$ é o produto de dois números inteiros E e d, isto é,

$$\mathrm{E}d \equiv 1 \bmod \phi(\mathrm{N}),$$

então

$$a = a^m = a^{\mathrm{E}d} = (a^{\mathrm{E}})^d.$$

Isto é, a potenciação $\cdot^d = \cdot^{1/\mathrm{E}}$ inverte $\cdot^{\mathrm{E}}$ módulo N, a **extração da** E**-ésima radiciação é a** d**-ésima potência**,

$$\sqrt[\mathrm{E}]{\cdot} = \cdot^{1/\mathrm{E}} \equiv \cdot^d \bmod \mathrm{N}.$$

Calcular uma potência é muitíssimo mais fácil do que uma raiz!

Exemplos.

- Se $p = 3$ e $q = 5$, então

  $$\mathrm{N} = pq = 15 \quad \text{e} \quad \phi(\mathrm{N}) = (p-1)(q-1) = 8.$$

  Se $\mathrm{E} = 3$ e $d = 3$, então $n = \mathrm{E}d = 9 \equiv 1 \bmod 8$.

Por exemplo, para a base 2, verificamos

$$2^{\mathrm{E}} = 2^3 = 8 \equiv 8 \bmod \mathrm{N}$$

e

$$8^d = 8^3 = 512 \equiv= 2 \bmod \mathrm{N}.$$

Isto é,

$$\sqrt[\mathrm{E}]{8} = 2 = 8^d \bmod \mathrm{N}.$$

- Se $p = 3$ e $q = 11$, então

$$\mathrm{N} = pq = 33 \quad \text{e} \quad \phi(\mathrm{N}) = (p-1)(q-1) = 20.$$

Se $\mathrm{E} = 7$ e $d = 3$, então $n = \mathrm{E}d = 21 \equiv 1 \bmod 20$.

Por exemplo, para a base 2, verificamos

$$2^{\mathrm{E}} = 2^7 = 128 = 29 + 3 \cdot 33 \equiv 29 \bmod \mathrm{N}$$

e

$$29^d = 29^3 = (-4)^3 \equiv -64 = 2 - 2 \cdot 33 \equiv 2 \bmod \mathrm{N}.$$

Isto é,

$$\sqrt[\mathrm{E}]{29} = 2 = 29^d \bmod \mathrm{N}.$$

Resumamos: Para números primos diferentes p e q, denote

$$\mathrm{N} := pq \quad \text{e} \quad \phi(\mathrm{N}) := (p-1)(q-1).$$

Se

$$\mathrm{E}d \equiv 1 \bmod \phi(\mathrm{N}),$$

isto é, $d \equiv 1/\mathrm{E} \bmod \phi(\mathrm{N})$ então

$$\sqrt[\mathrm{E}]{\cdot} = \cdot^{1/\mathrm{E}} \equiv \cdot^d \bmod \mathrm{N}.$$

Recordemo-nos dos resultados de Seção 7.2:

Questão. Para quais E existe d tal que $\mathrm{E}d \equiv \bmod \phi(\mathrm{N})$?

Resposta: Para todo E que é **relativamente primo** a $\phi(\mathrm{N})$, isto é, todo E que tem nenhum divisor comum com $\phi(\mathrm{N})$.

Questão. Dado $\phi(\mathrm{N})$ e E que seja relativamente primo a $\phi(\mathrm{N})$, como calcular tal d?

Resposta: Pelo Algoritmo de Euclides.

Cifração. (Recordemo-nos de que caracteres *maiúsculos* tendem a designar números *públicos* e caracteres *minúsculos* tendem a designar números *secretos*.)

Para a Alice enviar a mensagem m secretamente ao Bob através de um canal inseguro,

1. Bob, para **gerar** a chave, escolhe

   - dois números primos p e q, e
   - um expoente E relativamente primo a $\phi(\mathrm{N}) := (p-1)(q-1)$, e

   Bob, para **transmitir** a chave, envia à Alice

   - o produto $\mathrm{N} := pq$ (o *módulo*) e o expoente E (a *chave pública*).

2. Alice, para **cifrar**,

   - calcula $\mathrm{M} = m^{\mathrm{E}} \bmod \mathrm{N}$, e
   - transmite M ao Bob.

3. Bob, para **decifrar**,

   - calcula (pelo Algoritmo de Euclides [estendido]) d tal que $\mathrm{E}d \equiv 1 \bmod (p-1)(q-1)$ (e o qual existe porque E é relativamente primo a $\phi(\mathrm{N})$),
   - calcula $\mathrm{M}^d = m^{\mathrm{E}d} = m \bmod \mathrm{N}$ (pela *Fórmula de Euler*).

A computação de d pela fatoração de N é o atalho de que Bob dispõe.

O `CrypTool 1` oferece no Menu `Individual Procedures -> RSA Cryptosystem` a entrada `RSA Demonstration` para experimentar com os valores das chaves e da mensagem no `RSA`.

Resumo. Observamos como

- a potenciação x^{E} cifra, com o expoente a chave pública, e
- a radiciação decifra.

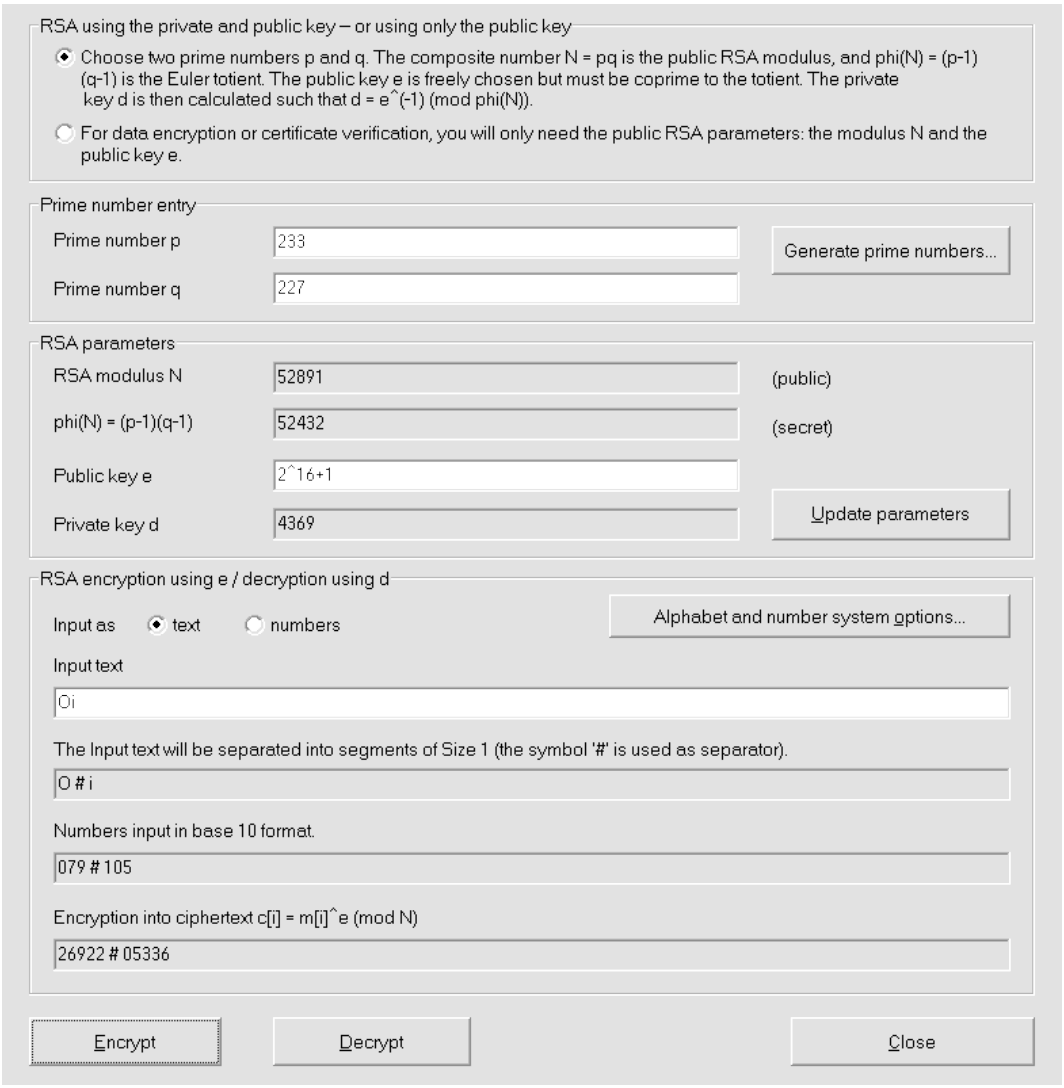


Figura 76: Os passos da cifração pelo RSA no CrypTool 1

Como tudo se calcula pelo resto módulo N (= pq, um produto de dois primos p e q), a radiciação é dificilimamente computável, não existem algoritmos muito mais rápidos do que o que prova todas as possibilidades e leva bilhões de anos.

Porém, pelo Teorema de Lagrange (ou, mais exatamente, a Fórmula de Euler), a radiciação de ordem E é igual à potenciação y^d para um número d que o Algoritmo de Euclides calcula a partir de E (e p e q, mais exatamente, de $\phi(N) = (p-1)(q-1)$). Logo, a chave secreta é d, ou, suficientemente, o conhecimento dos fatores primos p e q de N.

Implementação em Python. Vamos implementar o primeiro passo, a geração das chaves:

```
# RSA Key Generator
# http://inventwithpython.com/hacking (BSD Licensed)

import random, sys, os, rabinMiller, cryptomath


def main():
    # create a public/private keypair with 1024 bit keys
    print('Making key files...')
    makeKeyFiles('al_sweigart', 1024)
    print('Key files made.')

def generateKey(keySize):
    # Creates a public/private key pair with keys that are keySize
    # bits in size. This function may take a while to run.

    # Step 1: Create two primes p and q. Calculate n = p * q.
    print('Generating p prime...')
    p = rabinMiller.generateLargePrime(keySize)
    print('Generating q prime...')
    q = rabinMiller.generateLargePrime(keySize)
    n = p * q

    # Step 2: Create a number e that is relatively prime to (p-1)*(q-1).
```

```
    print('Generating e that is relatively prime to (p-1)*(q-1)...')
    while True:
        # Keep trying random numbers for e until one is valid.
        e = random.randrange(2 ** (keySize - 1), 2 ** (keySize))
        if cryptomath.gcd(e, (p - 1) * (q - 1)) == 1:
            break

    # Step 3: Calculate d, the mod inverse of e.
    print('Calculating d that is mod inverse of e...')
    d = cryptomath.ModInverse(e, (p - 1) * (q - 1))

    publicKey = (n, e)
    privateKey = (n, d)

    print('Public key:', publicKey)
    print('Private key:', privateKey)

    return (publicKey, privateKey)


def makeKeyFiles(name, keySize):
    # Creates two files 'x_pubkey.txt' and 'x_privkey.txt' (where x is
    # the # value in name) with the the n, e and d, e integers written
    # in them, delimited by a comma.

    # safety check to prevent us from overwriting our old key files:
    if os.path.exists('%s_pubkey.txt' % (name)) or
       path.exists('%s_privkey.txt' % (name)):
       sys.exit('WARNING: The file %s_pubkey.txt or %s_privkey.txt
       # already exists! Use a different name or delete these files
       # and re-run this program.' %(e, name))

    publicKey, privateKey = generateKey(keySize)

    print()
    print('The public key is a %s and a %s digit number.' %
(str(publicKey[0])), len(str(publicKey[1]))))
    print('Writing public key to file %s_pubkey.txt...' % (name))
```

```
    fo = open('%s_pubkey.txt' % (name), 'w')
    fo.write('%s,%s,%s' % (keySize, publicKey[0], publicKey[1]))
    fo.close()

    print()
    print('The private key is a %s and a %s digit number.' %
(str(publicKey[0])), len(str(publicKey[1]))))
    print('Writing private key to file %s_privkey.txt...' % (name))
    fo = open('%s_privkey.txt' % (name), 'w')
    fo.write('%s,%s,%s' % (keySize, privateKey[0], privateKey[1]))
    fo.close()


# If makeRsaKeys.py is run (instead of imported as a module) call
# the main() function.
if __name__ == '__main__':
    main()
```

A implementação dos dois outros passos, a cifração e decifração:

```
# RSA Cipher
# http://inventwithpython.com/hacking (BSD Licensed)

import sys

# IMPORTANT: The block size MUST be less than or equal to the key size!
# (Note: The block size is in bytes, the key size is in bits. There
# are 8 bits in 1 byte.)
DEFAULT_BLOCK_SIZE = 128 # 128 bytes
BYTE_SIZE = 256 # One byte has 256 different values.

def main():
    # Runs a test that encrypts a message to a file or decrypts a message
    # from a file.
    filename = 'encrypted_file.txt' # the file to write to/read from
    mode = 'encrypt' # set to 'encrypt' or 'decrypt'

    if mode == 'encrypt':
```

```
        message = '''"Quem a raposa quer enganar, muito tem que madrugar" -RJ'''
        pubKeyFilename = 'al_sweigart_pubkey.txt'
        print('Encrypting and writing to %s...' % (filename))
        encryptedText = encryptAndWriteToFile(filename, pubKeyFilename,
        message)

        print('Encrypted text:')
        print(encryptedText)

    elif mode == 'decrypt':
        privKeyFilename = 'al_sweigart_privkey.txt'
        print('Reading from %s and decrypting...' % (filename))
        decryptedText = readFromFileAndDecrypt(filename, privKeyFilename)

        print('Decrypted text:')
        print(decryptedText)


def getBlocksFromText(message, blockSize = DEFAULT_BLOCK_SIZE):
    # Converts a string message to a list of block integers.
    # Each integer represents 128 (= blockSize) string characters.

    messageBytes = message.encode('ascii') # convert string to bytes

    blockInts = []
    for blockStart in range(0, len(messageBytes), blockSize):
        # Calculate the block integer for this block of text
        blockInt = 0
        for i in range(blockStart, min(blockStart + blockSize,
            len(messageBytes))):
            blockInt += messageBytes[i] * (BYTE_SIZE ** (i % blockSize))
        blockInts.append(blockInt)
    return blockInts


def getTextFromBlocks(blockInts, messageLength,
Size = DEFAULT_BLOCK_SIZE):
    # Converts a list of block integers to the original message string.
```

```
    # The original message length is needed to properly convert the
    # last block integer.
    message = []
    for blockInt in blockInts:
        blockMessage = []
        for i in range(blockSize - 1, -1, -1):
            if len(message) + i < messageLength:
                # Decode the message string for the 128 (= blockSize)
                # characters from this block integer.
                asciiNumber = blockInt // (BYTE_SIZE ** i)
                blockInt = blockInt % (BYTE_SIZE ** i)
                blockMessage.insert(0, chr(asciiNumber))
        message.extend(blockMessage)
    return ''.join(message)


def encryptMessage(message, key, blockSize = DEFAULT_BLOCK_SIZE):
    # Converts the message string into a list of block integers, and
    # then encrypts each block integer. Pass the PUBLIC key to encrypt.
    encryptedBlocks = []
    n, e = key

    for block in getBlocksFromText(message, blockSize):
        # ciphertext = plaintext ^ e mod n
        encryptedBlocks.append(pow(block, e, n))
    return encryptedBlocks


def decryptMessage(encryptedBlocks, messageLength, key,
Size = DEFAULT_BLOCK_SIZE):
    # Decrypts a list of encrypted block ints into the original message
    # string. The original message length is required to properly decrypt
    # the last block. Be sure to pass the PRIVATE key to decrypt.
    decryptedBlocks = []
    n, d = key
    for block in encryptedBlocks:
        # plaintext = ciphertext ^ d mod n
        decryptedBlocks.append(pow(block, d, n))
```

```
    return getTextFromBlocks(decryptedBlocks, messageLength, blockSize)


def readKeyFile(keyFilename):
    # Given the filename of a file that contains a public or private key,
    # return the key as a (n,e) or (n,d) tuple value.
    fo = open(keyFilename)
    content = fo.read()
    fo.close()
    keySize, n, EorD = content.split(',')
    return (int(keySize), int(n), int(EorD))


def encryptAndWriteToFile(messageFilename, keyFilename, message,
Size = DEFAULT_BLOCK_SIZE):
    # Using a key from a key file, encrypt the message and save it to a
    # file. Returns the encrypted message string.
    keySize, n, e = readKeyFile(keyFilename)

    # Check that key size is greater than block size.
    if keySize < blockSize * 8: # * 8 to convert bytes to bits
        sys.exit('ERROR: Block size is %s bits and key size is %s bits.
        RSA cipher requires the block size to be <= the key size!
        Either increase the block size or use different keys.'
        % (blockSize * 8, len(keySize))

    # Encrypt the message
    encryptedBlocks = encryptMessage(message, (n, e), blockSize)

    # Convert the large int values to one string value.
    for i in range(len(encryptedBlocks)):
        encryptedBlocks[i] = str(encryptedBlocks[i])
    encryptedContent = ','.join(encryptedBlocks)

    # Write out the encrypted string to the output file.
    encryptedContent = '%s_%s_%s' % (len(message), blockSize,
ptedContent)
    fo = open(messageFilename, 'w')
```

```
    fo.write(encryptedContent)
    fo.close()
    # Also return the encrypted string.
    return encryptedContent


def readFromFileAndDecrypt(messageFilename, keyFilename):
    # Using a key from a key file, read an encrypted message from a
    # file and then decrypt it. Returns the decrypted message string.
    keySize, n, d = readKeyFile(keyFilename)


    # Read in message length and encrypted message from the file.
    fo = open(messageFilename)
    content = fo.read()
    messageLength, blockSize, encryptedMessage = content.split('_')
    messageLength = int(messageLength)
    blockSize = int(blockSize)

    # Check that key size is greater than block size.
    if keySize < blockSize * 8: # * 8 to convert bytes to bits
        sys.exit('ERROR: Block size is %s bits and key size is %s bits.
        RSA cipher requires the block size to be <= key size!
        Did you give the correct key file and encrypted file?'
        % (blockSize * ySize))

    # Convert the encrypted message into large int values.
    encryptedBlocks = []
    for block in encryptedMessage.split(','):
        encryptedBlocks.append(int(block))

    # Decrypt the large int values.
    return decryptMessage(encryptedBlocks, messageLength, (n, d),
Size)


# If rsaCipher.py is run (instead of imported as a module) call
# the main() function.
```

```
if __name__ == '__main__':
    main()
```

Assinatura. Para assinar (em vez de cifrar), a única diferença é que os expoentes E e d trocam os seus papeis. Isto é, $\mathrm{M} = m^d$ (em vez de m^{E}): Para a Samanta assinar a mensagem m e o Vitor verificá-la,

1. Samanta, para **gerar** a rubrica (ou *firma*), escolhe

   - dois números primos p e q, e
   - um expoente E relativamente primo a $(p-1)(q-1)$, e

   Samanta, para **transmitir** a rubrica, envia ao Vitor

   - o produto $\mathrm{N} = pq$ (o *módulo*) e o expoente E (a *chave pública*).

2. Samanta, para **assinar**,

   - calcula (pelo Algoritmo de Euclides [estendido]) d tal que $\mathrm{E}d \equiv 1 \mod (p-1)(q-1)$ (e o qual existe porque E é relativamente primo a $(p-1)(q-1)$),
   - calcula $\mathrm{M} = m^d \mod \mathrm{N}$, e
   - transmite M ao Vitor.

3. Vitor, para **verificar**,

   - calcula $\mathrm{M}^{\mathrm{E}} = m^{\mathrm{E}d} = m \mod \mathrm{N}$ (pela *Fórmula de Euler*).

Observação. A assinatura e a decifração são ambas a potenciação $\cdot^d$ à chave privada d. Logo, assinar um documento cifrado (pelo chave pública E que corresponde a d) equivale a decifrá-lo! Por isso, na prática,

- usam-se chaves diferentes para cifrar / decifrar e assinar / verificar, e
- assina-se um *hash criptográfico* $h(d)$ do documento d, um número de um certo número de bits que identifica (para evitar outro documento ter o mesmo hash), mas não permite deduzir d a partir de $h(d)$.

O `CrypTool 1` oferece no Menu `Individual Procedures -> RSA Cryptosystem` a entrada `Signature Generation` para experimentar com os valores

- da rubrica, e

- da mensagem.

Observamos que, em vez da *mensagem* original, é assinado

- um hash *criptográfico* (por exemplo, `MD5`) da mensagem original, e
- com informações adicionais, como
  - Nome do Assinador, e
  - Algoritmo usado para cifrar e calcular o hash.

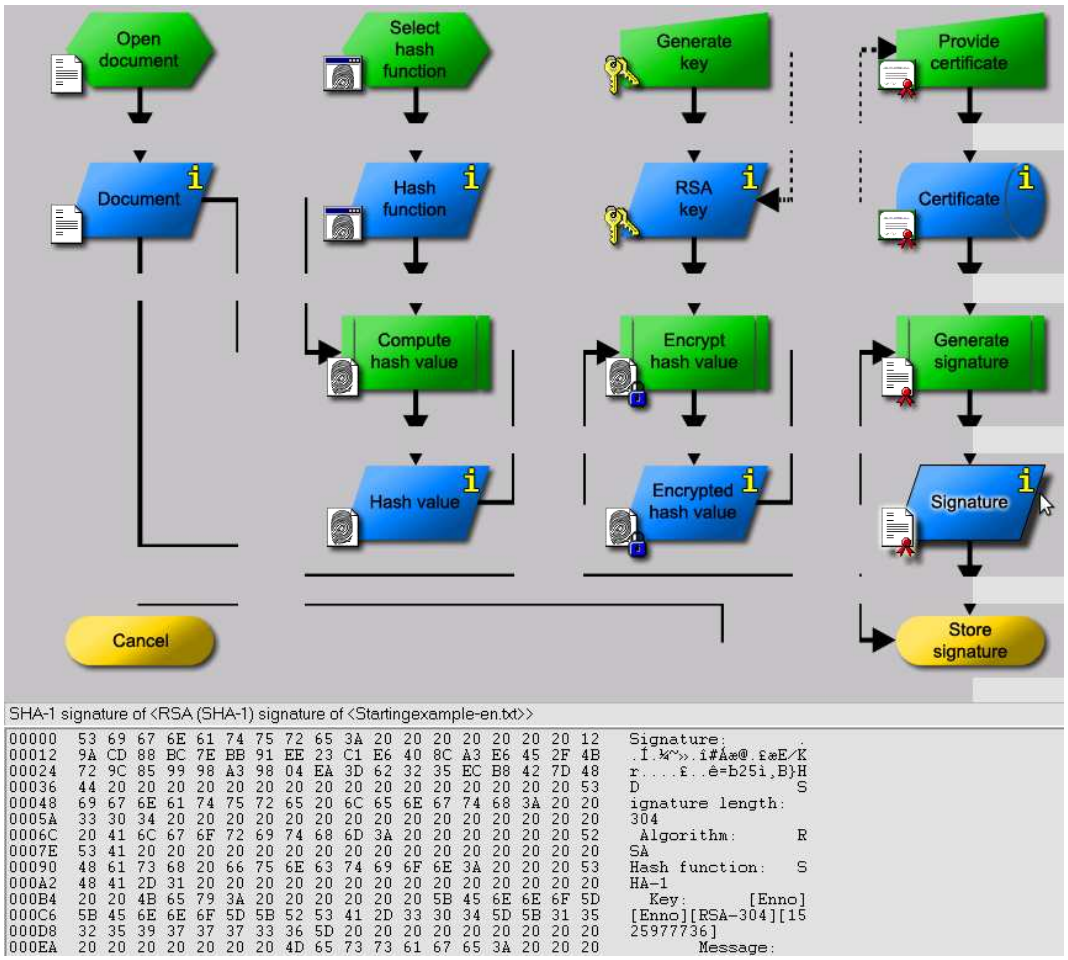


Figura 77: Os passos da assinatura pelo `RSA` no `CrypTool 1`

Segurança. Como são **públicos**

- o módulo N,
- o expoente E e
- a mensagem cifrada M ($= m^{\mathrm{E}}$),

a segurança computacional de RSA baseia-se unicamente na **dificuldade da computação da radiciação**

$$m \equiv \sqrt[\mathrm{E}]{\mathrm{M}}\ (= \mathrm{M}^{1/\mathrm{E}}) \bmod \mathrm{N}.$$

O **atalho** é o conhecimento dos dois fatores primos p e q de $\mathrm{N} = pq$ que possibilita calcular

- o módulo $\phi(\mathrm{N}) := (p-1)(q-1)$, e
- o inverso multiplicativo $d = 1/\mathrm{E}$ de E, isto é, d tal que

$$\mathrm{E}d \equiv 1 \bmod (p-1)(q-1);$$

tal que, pela *Fórmula de Euler*, $\mathrm{M}^d = m^{\mathrm{E}d} \equiv m \bmod \mathrm{N}$.

Um olheiro obteria a mensagem secreta m a partir de N, E e M, se pudesse computar

$$m \equiv \sqrt[\mathrm{E}]{\mathrm{M}}\ (= \mathrm{M}^{1/\mathrm{E}}) \bmod \mathrm{N}.$$

isto é, a *radiciação* $\sqrt[\mathrm{E}]{\cdot}$ que inverte a *potenciação* $x \mapsto x^d = y$,

$$\sqrt[\mathrm{E}]{y} = x \qquad \text{com } x \text{ tal que } x^{\mathrm{E}} = y.$$

Números Apropriados. Enquanto a potenciação é facilmente computável, a **radiciação é dificilmente computável** para escolhas de $\mathrm{N} = pq$ e E **apropriadas**, isto é, os números primos p e q suficientemente grandes (enquanto a escolha do expoente E é arbitrário, por exemplo, popularmente E = 2):

É o Teorema de Euclides que garante que haja números primos arbitrariamente **grandes** (> 1024 bits),

$$\#\{\text{ números primos }\} = \infty$$

Computação da Fatoração. Atualmente, o algoritmo mais rápido para calcular a fatoração pq a partir de N, é o *campo de número de peneira geral* de Lenstra et al. (1993). A grosso modo, o número de operações para fatorar um número inteiro de n bits é

$$\exp(\log n^{1/3}).$$

Texto Claro Joeirado (CPA = chosen plaintext attack). O atacante tem o texto cifrado e o texto claro correspondente, e pode escolher os textos claros; assim que o atacante possa deliberadamente variar, ou *adaptar*, o texto claro e analisar as alterações resultantes no texto cifrado. (A criptoanálise diferencial situa-se neste cenário.)

Este é o cenário mínimo para criptografia assimétrica! Como a chave de criptografia é pública, o atacante pode cifrar mensagens à vontade. Logo, se o atacante pode reduzir o número de textos claros possíveis, por exemplo, ele sabe que são ou "Sim" ou "Não", então ele pode cifrar todos estes textos claros possíveis pela chave pública e compará-los com o texto cifrado interceptado. Para este o algoritmo modelar RSA não sofrer deste ataque, é preciso de preencher o texto claro por dados aleatórios para robustecê-lo contra este ataque CPA.

Acolchamento. Na prática, a mensagem secreta m precisa de ser *enchida*, isto é, aleatoriamente aumentada. Senão, quando a mensagem clara m e o expoente E são pequenos (por exemplo, $\mathrm{E} = 3$), possivelmente a mensagem cifrada $\mathrm{M} = m^{\mathrm{E}}$ satisfaz $\mathrm{M} < \mathrm{N}$. Neste caso, a igualdade

$$m = \sqrt[\mathrm{E}]{\mathrm{M}} \quad \text{vale em } \mathbb{Z}!$$

Por isso, m é facilmente computável, por exemplo,

- pelo **Método da Bisseção** já apresentada, ou,
- numericamente mais eficaz, pelo método de Newton.

Ataques. Um ataque hipotético mais simples é o de Wiener quando a chave privada d é pequena: **Observação**. O Teorema de Wiener se aplica quando $d < 1/3 \cdot \mathrm{N}^{1/4}$, isto é, d é pequeno demais, e $q < p < 2q$, isto é, p e q são próximos demais.

Seja $n = pq$ com p e q números primos diferentes; põe $\phi(n) = (p-1)(q-1)$. Seja e a chave pública e $d = e^{-1} \bmod \phi(n)$, isto é, $ed \equiv 1 \bmod \phi(n)$.

Tem-se $ed \equiv 1 \bmod (p-1)(q-1)$, se, e tão-somente se, existe k tal que

$$ed = 1 + k(p-1)(q-1) = 1 + k(n - p - q + 1)$$

se, e tão-somente se,

$$kn - ed = k(p + q - 1) - 1.$$

Dividindo por dn,

$$\frac{k}{d} - \frac{e}{n} = \frac{k}{d}\left(\frac{1}{q} + \frac{1}{p} - \frac{1}{n}\right) - \frac{1}{dn}.$$

Teorema (de Wiener). Se $\left|\frac{a}{b} - x\right| < \frac{1}{2b^2}$, então $\frac{a}{b}$ é uma fração contínua que aproxima x.

Se as condições do teorema são satisfeitas, então, como e/n é público, é possível achar o segredo d como denominador da fração contínua (mais exatamente, de uma convergente de uma fração contínua) aproximativa k/d.

Observação. O Teorema de Wiener se aplica quando $d < 1/3n^{1/4}$, isto é, d é pequeno demais, e $q < p < 2q$, isto é, p e q são próximos demais.

Demonstração: Por definição $ed \equiv 1 \bmod \phi(n)$, se existe um inteiro k com $ed = k\phi(n) + 1$. Logo $|ed - k\phi(n)| = 1$ e $\left|\frac{e}{\phi(n)} - \frac{k}{d}\right| = \frac{1}{d\phi(n)}$. Usemos n como aproximação a $\phi(n)$, visto que $\phi(n) = (p-1)(q-1) = n - p - q + 1$ e logo $|n - \phi(n)| = p + q - 1 < 3\sqrt{n}$ (pois $q < p < 2q$).

Logo

$$\left|\frac{e}{n} - \frac{k}{d}\right| = \left|\frac{ed - kn}{nd}\right| = \left|\frac{ed - kn - k\phi(n) + k\phi(n)}{nd}\right| = \left|\frac{1 - k(n - \phi(n))}{nd}\right| < |\frac{3k\sqrt{n}}{nd}|.$$

Como $ke < k\phi(n) = de - 1 < de$, vale $k < d < \frac{1}{3}n^{\frac{1}{4}}$. Logo

$$\left|\frac{e}{n} - \frac{k}{d}\right| < \frac{3}{d\sqrt{n}} \times \frac{n^{\frac{1}{4}}}{3} < \frac{1}{dn^{\frac{1}{4}}} < \frac{1}{3d^2}. \qquad \text{q.e.d.}$$

O número das frações $\frac{k}{d}$ para $d < n$ que estão tão próximas de $\frac{e}{n}$ é limitado por $\log_2(n)$, e todas tais frações são frações contínuas de $\frac{e}{n}$. Isto é, o atacante apenas tem de calcular as $\log_2(n)$ primeiras frações contínuas de $\frac{e}{n}$, e uma delas será $\frac{k}{d}$ (que é irredutível porque $ed - k\phi(n) = 1$ implica que $\text{mdc}(k, d) = 1$).

Concluímos então que um algoritmo em tempo linear basta para encontrar a chave secreta d.

Afim de implementar o ataque de Wiener, segue o código `Python` para

1. calcular as frações contínuas que se aproximam da fração n/d:

```
def cf_expansion(n, d):
    e = []

    q = n // d
    r = n % d
    e.append(q)

    while r != 0:
        n, d = d, r
        q = n // d
        r = n % d
        e.append(q)

    return e
```

2. calcular a fração a partir de uma fração contínua $n/d = a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \ldots}}$ com coeficientes $[a_0, a_1, a_2, \ldots]$:

```
def convergents(e):
    n = [] # Nominators
    d = [] # Denominators

    for i in range(len(e)):
        if i == 0:
            ni = e[i]
            di = 1
        elif i == 1:
            ni = e[i]*e[i-1] + 1
            di = e[i]
        else: # i > 1
            ni = e[i]*n[i-1] + n[i-2]
            di = e[i]*d[i-1] + d[i-2]

        n.append(ni)
        d.append(di)
        yield (ni, di)
```

Observação. Mencionamos que se ao contrário E (e não d) é muito pequeno, então um ataque é muito mais difícil; vide o artigo retrospetivo por Boneh nas Notices of the AMS.

8.2 O Algoritmo ElGamal

Apresentamos o algoritmo ElGamal que cria

- uma chave pública para cifrar, e
- uma chave privada para decifrar.

Em particular, permite aos dois correspondentes de comunicar cifradamente através de um canal inseguro como na troca de chaves de Diffie-Hellman (se cada correspondente cria a sua própria chave pública e privada e compartilha a chave pública).

Este algoritmo, tão próximo da troca de chaves de Diffie-Hellman, cria uma chave secreta mútua. Isto é, como na troca de chaves de Diffie-Hellman, falta a vantagem (da criptografia assimétrica em comparação à criptografia simétrica) que a chave é comprometível em um único lugar.

Contudo, esta desvantagem é teórica, porque para cada mensagem uma nova chave mútua é criada. Se um atacante tem acesso à chave do remetente, provavelmente também à mensagem que enviou.

Em contraste ao RSA, onde os correspondentes precisam,

- para *cifrar*, criar uma chave (privada e) pública *permanente* pelo destinatário, e
- para *assinar*, criar uma chave privada (e pública) *permanente* pelo assinador,

os algoritmos que se baseiam no protocolo de Diffie-Hellman, que estabelece apenas uma chave secreta mútua e precisa da interação dos correspondentes, isto é, ambos tem de ser online, precisam, para contornar esta disponibilidade,

- para *cifrar*,
  - criar uma chave (privada e) pública *permanente* pelo destinatário, e
  - criar uma chave privada (e pública) *efêmera* pelo remetente, e

Figura 78: O Criador do Algoritmo Taher El Gamal (*1955)

- para *assinar*, *ambas* pelo assinador,
  - criar uma chave privada (e pública) *permanente* e
  - criar uma chave (privada e) pública *efêmera*.

Cifração. Para a Alice enviar a mensagem m secretamente ao Bob através de um canal inseguro, combinam primeiro

- um número primo p *apropriado* (o *módulo*), e
- um número natural g *apropriado* (a *base*),

onde *apropriado* se refere às mesmas propriedades como para o algoritmo de Diffie-Hellman, isto é,

- que p seja grande e $p-1$ tenha um fator primo grande q, e
- que $\#\{g^0, g^1, g^2, \ldots\} \geq q$.

1. Bob, para **gerar** uma chave, escolhe um número b,
   - calcula $\mathrm{B} = g^b \bmod p$, e
   - transmite B à Alice.
2. Alice, para **cifrar**, escolhe um número efêmero a,
   - calcula $\mathrm{A} = g^a \bmod p$,
   - calcula $c = \mathrm{B}^a \bmod p$ e $\mathrm{M} = mc \bmod p$, e
   - transmite A e M ao Bob.
3. Bob, para **decifrar**,
   - calcula $\mathrm{A}^b = (g^a)^b = g^{ba} = g^{ab} = (g^b)^a = \mathrm{B}^a = c \bmod p$
   - calcula $c^{-1} \bmod p$ (pelo Algoritmo de Euclides [estendido]), e
   - calcula $\mathrm{M}c^{-1} = mcc^{-1} = m \bmod p$.

Veja https://asecuritysite.com/encryption/elgamal para uma demonstração dos passos em `JavaScript`.

Observação. Em vez da multiplicação $\mathrm{M} = m \cdot c$ de m por c, qualquer outra operação invertível é possível; por exemplo, a adição $\mathrm{M} = m + c$.

Assinatura `ElGamal`. Veremos que e facílimo verificar mas dificílimo calcular a solução s de uma equação módulo um número primo grande p que foi potenciada de ambos os lados por uma base g: para resolvê-la precisaria

1. ou calcular o logaritmo modular com a base g de ambos os lados,
2. ou já conhecer os valores do logaritmo modular.

Na assinatura,

- quem subscreve já conhece os valores do logaritmos (2), enquanto
- quem verifica, como praticamente não consegue calcular o logaritmo modular (1), só pode verificar que quem assinou os conhece, isto é, constatar que é dono da chave segreda para calcular a solução s.

É isto uma assinatura, um vínculo infalsificável entre um documento e uma identidade.

Protocolo: Para a Samanta assinar um documento d e o Vitor verificar a assinatura, combinam primeiro

- um número primo p *apropriado*, e
- um número natural g *apropriado.*

onde *apropriado* se refere às mesmas propriedades como para o algoritmo de Diffie-Hellman, isto é,

- que p seja grande e $p-1$ tenha um fator primo grande q, e
- que $\#\{g^0, g^1, g^2, \ldots\} \geq q$.

1. Samanta, para **criar** uma rubrica (ou *firma*),
    1. escolhe um número f,
    2. calcula $\mathrm{F} = g^f \bmod p$, e
    3. transmite F ao Vitor.
2. Samanta, para **assinar** o documento d,
    1. escolhe um número efêmero e com $\mathrm{mdc}(e, p-1) = 1$ (para garantir $e^{-1} \bmod p-1$ existir),
    2. calcula $\mathrm{E} = g^e \bmod p-1$,
    3. calcula $\mathrm{D} = g^d \bmod p$,

4. resolve $D = F^E E^S \bmod p$ para S, isto é, tomando o logaritmo com base g em ambos os lados da equação, calcula $S = [d - Ef]/e \bmod p - 1$, e
5. transmite E e S ao Vitor.

3. Vitor, para **verificar** a assinatura,

1. calcula $F^E E^S \bmod p$,
2. calcula $D = g^d \bmod p$, e
3. verifica que $D = F^E E^S$.

Observação. Para calcular S a partir de d, E e F, usa-se a fórmula $S = \log_E(g^d/F^E) \bmod p$, isto é, precisa de calcular o logaritmo modular que leva tempo exponencial. É preciso saber f e e para calcular $S = [d - Ef]/e$ em tempo polinomial. Isto explica porque E em vez de e aparece como expoente de F: caso contrário, Samanta teria de publicar e com S; isto revelaria a sua rubrica secreta f!

Advertência. Se um dos dois números secretos, ou f ou e, for conhecido, então ambos, f e e, serão conhecidos (pela fórmula $S = [d - Ef]/e$)! Em particular, se a chave secreta auxiliar (*efêmera*) e for conhecida, então a rubrica secreta (*permanente*) f será conhecida pela fórmula $f = [d - se]/E$! Por exemplo, se e é usada para cifrar dois documentos diferentes d' e d'', então há tantas (isto é, duas) equações quanto incógnitas (e e f); logo, ambas podem ser resolvidas, isto é, e e f obtidas! Logo, é importantíssimo usar e uma única vez e manter secreta; em particular, a sua geração tem de ser suficientemente aleatória.

Assinatura `DSA`. A assinatura `DSA` baseia-se na assinatura do `ElGamal` (para um módulo primo com 1024 dígitos binários), mas é computacionalmente seis vezes mais econômica porque trabalha com um grupo menor de potências de g (de cardinalidade q com 160 dígitos binários para um fator primo q de $p - 1$ ao invés da cardinalidade $p - 1$ com 1024 dígitos binários).

Protocolo: Para a Samanta assinar um documento d e o Vitor verificar a assinatura, combinam primeiro

- um número primo p *apropriado*, isto é,
  - tal que p tenha L bits para L entre os múltiplos $\{512, 576, \ldots, 1024\}$ de 64, e
  - tal que $p - 1$ tenha um fator primo q de 160 bits.

- um número natural g *apropriado*; para isto,
  - escolhe g_0 em $\{1,\ldots,p-1\}$ tal que, com $p-1=qr$, valha $g_0^r > 1 \bmod p$, e
  - calcula $g = g_0^r \bmod p > 1$.

Os dados públicos são p, q e g

1. Para **criar** a rubrica (ou *firma*):
   1. Escolhe um número f em $\{1,\ldots,q-1\}$.
   2. Calcula $\mathrm{F} = g^f \bmod p$.
   3. Transmite F ao Vitor.
2. Samanta, para **assinar** o documento d_0 pela rubrica (f,F)
   1. Calcula o hash `SHA-1` d do documento d_0.
   2. Escolhe um número efêmero e *apropriado* em $\{1,\ldots,q-1\}$:
   3. Calcula $\mathrm{E} = (g^e \bmod p) \bmod q$; se $\mathrm{E} = 0$, então escolhe outro e.
   4. Resolve $\mathrm{DF}^{\mathrm{E}} = \mathrm{E}^{\mathrm{S}} \bmod q$ para S, isto é, calcula $\mathrm{S} = (d + \mathrm{E}\cdot f)/e \bmod q$; se $d + \mathrm{E}\cdot f \equiv 0 \bmod q$, então escolhe outro e.
   5. Transmite E e S ao Vitor.
3. Vitor, para **verificar** a assinatura,
   1. Calcula $\mathrm{S}' = \mathrm{S}^{-1} \bmod q$.
   2. Calcula $d' = d\cdot \mathrm{S}' \bmod q$.
   3. Calcula $\mathrm{E}' = \mathrm{E}\cdot \mathrm{S}' \bmod q$.
   4. Calcula $v = [g^{d'}\cdot \mathrm{F}^{\mathrm{E}'} \bmod p] \bmod q$, e
   5. verifica se $v = \mathrm{E}$.

Observamos, que em comparação com o `El Gamal`, o algoritmo `DSA`

- trabalha com uma base que é uma potência $g = g_0^r$ de uma raiz primitiva g_0 (para diminuir a sua ordem a q ao invés de $p-1$; esta diminuição facilitará os cálculos seguintes),
- trabalha com os expoentes de g módulo q ao invés de módulo p,
- ao invés de resolver $\mathrm{D} = \mathrm{E}^{\mathrm{S}}\mathrm{F}^{\mathrm{E}}$ (módulo p), resolve $\mathrm{D} = \mathrm{E}^{\mathrm{S}}/\mathrm{F}^{\mathrm{E}}$, isto é, $\mathrm{DF}^{\mathrm{E}} = \mathrm{E}^{\mathrm{S}}$ (módulo q) para S, e obtém $\mathrm{S} = [d + \mathrm{E}\cdot f]/e$ e
- conforme,
  - ao invés de verificar $\mathrm{D} = \mathrm{F}^{\mathrm{E}}\mathrm{E}^{\mathrm{S}}$, verifica $\mathrm{E}^{\mathrm{S}} = \mathrm{DF}^{\mathrm{E}}$;

– além disto, ao invés de verificar $E^S = DF^E$, verifica $E = \sqrt[S]{D}\sqrt[S]{F^E} = g^{d'}F^{E'}$ para $d' = dS'$ e $E' = ES' \bmod q$ onde $S' = S^{-1}$.

Como explica a subsecção Módulos Arbitrários em Seção 6.2, esta diminuição da ordem da base g compromete a segurança moderadamente:

- Se o módulo $m = pq$ é produto de dois fatores p e q sem fator comum, então o logaritmo modular

$$\log_g \bmod m$$

pode ser computado, pelo Teorema Chinês dos Restos, a partir dos logaritmos

$$\log_g \bmod p \quad \text{e} \quad \log_g \bmod q$$

pelo Algoritmo de Euclides (estendido).

- Se o módulo $m = p^e$ é uma potência de um primo p, então o logaritmo modular módulo m pode ser computado em tempo polinomial a partir do logaritmo módulo p

  – pelo Logaritmo de $1 + p\mathbb{Z}/p^{e-1}\mathbb{Z} \bmod p^e$, e
  – pelo Algoritmo de Euclides (estendido).

9 Criptografia por Curvas Elípticas

Entre todas as curvas, a graça das elípticas (dadas por uma equação $y^2 = x^3 + ax + b$) é que permitem somar pontos ($p + q + r = 0$ se uma reta passa por p, q e r). Ao restringirmos da equação às soluções (x, y) em $\mathbb{F}_p \times \mathbb{F}_p$ para um grande número primo p e fixarmos um tal ponto P,

- enquanto, dado um número n, é **fácil** computar a *exponencial*, isto é,

$$\mathrm{Q} = n\mathrm{P} = \mathrm{P} + \ldots + \mathrm{P},$$

- em contraste, dado $\mathrm{Q} = \mathrm{P} + \ldots + \mathrm{P}$, é **dificílimo** computar o *logaritmo*; isto é, quantas vezes P foi adicionado; isto é, computar o número n tal que $\mathrm{Q} = n\mathrm{P}$.

A criptografia por curvas elípticas `ECC` (Elliptic Curve Criptography) é uma variação do protocolo de `Diffie-Hellman`:

- Em vez de *multiplicarmos* repetidamente (n vezes) a base g em $\mathbb{F}_p^*$, isto é, calcular

$$g^n = g \cdots g,$$

- *adicionamos* repetidamente (n vezes) um ponto G, isto é, calculamos

$$n \cdot \mathrm{G} = \mathrm{G} + \cdots + \mathrm{G}.$$

A vantagem de usar como problema criptográfico,

- em vez do logaritmo sobre o *grupo multiplicativo* (de um corpo) finito (isto é, a função que para dados números g e y determina o escalar x em $\mathbb{N}$ tal que $y = g^x$),
- o *logaritmo* sobre o *grupo de uma curva elíptica* finita (isto é, a função que para dados pontos G e Y determina o escalar x em $\mathbb{N}$ tal que $\mathrm{Y} = x\mathrm{G}$)

é que em dependência do número de bites n de p (pelos algoritmos mais rápidos *presentemente* conhecidos)

- o tempo para computar o logaritmo sobre uma curva elíptica aumenta *linearmente* e leva cerca de $n/2$ operações, enquanto
- o tempo para computar o logaritmo multiplicativo aumenta *sublinearmente* e leva cerca de $n^{1/3}$ operações.

Por exemplo, a segurança obtida por uma chave de 2048 bites para o logaritmo multiplicativo equivale aproximadamente à de uma chave de 224 bites para o logaritmo sobre uma curva elíptica.

Nas próximas secções:

1. Vamos primeiro introduzir estes corpos finitos gerais (porque umas curvas elípticas finitas comuns definem-se sobre corpos finitos mais gerais do que os da forma $\mathbb{F}_p$ para um primo p até agora conhecidos.)
2. Definimos uma curva elíptica.
3. Estudamos a adição de pontos de uma curva elíptica.
4. Damos as implementações
    - da Troca de Chaves de Diffie-Hellman, e
    - da assinatura.
5. Estudamos o problema criptográfico por trás das curvas e os algoritmos que o resolvem.

9.1 Corpo Finitos

Anteriormente nos damos conta que já usamos a aritmética modular no dia-a-dia, por exemplo, para o módulo $m = 12$, na aritmética do relógio, e para $m = 7$, nos dias da semana. Mais geralmente, definimos, para m um número inteiro qualquer, o *anel* finito $\mathbb{Z}/m\mathbb{Z}$ (= um conjunto finito em que podemos somar e multiplicar); a grosso modo definimo-lo

- como conjunto por $\{0,1,\ldots,m-1\}$, e
- a soma (respetivamente produto) de x e y em $\mathbb{Z}/m\mathbb{Z} = \{0,1,\ldots,m-1\}$ é definida pelo resto da soma $x+y$ (respetivamente produto) em $\mathbb{Z}$ dividido por m.

Se $m = p$ é primo, então mostramos em Seção 5.2 que $\mathbb{Z}/p\mathbb{Z}$ é um *corpo* denotado por $\mathbb{F}_p$. Isto é, para todo a em $\mathbb{F}_p$ exceto 0, sempre existe a^{-1} em $\mathbb{F}_p$, o inverso *multiplicativo* de a definido por $aa^{-1} = 1$. Em outras palavras, em um corpo podemos dividir por qualquer número exceto 0. (Os exemplos mais comuns são os corpos infinitos $\mathbb{Q}$, $\mathbb{R}$ e $\mathbb{C}$.)

Na criptografia, curvas elípticas são definidas sobre corpos $\mathbb{F}_q$ cuja cardinalidade é uma potência $q = p^n$ de um número primo p (e não somente da forma $\mathbb{F}_p$ como era o caso até agora); por exemplo, $q = 2^n$ para um número n grande.

O caso $p = 2$ é particularmente apto para a computação (criptográfica). Os corpos da forma $\mathbb{F}_{2^n}$ são chamados *binários*. Demos primeiro um nome ao número p que figura em q.

Definição: O número primo p é a *característica* do corpo, o menor n em $\mathbb{N}$ tal que $n = 1 + \cdots + 1 = 0$.

O corpo $\mathbb{F}_q$ para $q = p^n$ é definido por polinômios de grau n sobre $\mathbb{F}_p$,

$$\mathbb{F}_q = \{a_{n-1}\mathrm{X}^{n-1} + \cdots + a_0 \text{ para } a_{n-1}, \ldots, a_0 \text{ em } \mathbb{F}_p\}.$$

- A adição + de dois polinômios é a adição de polinômios, isto é, a adição coeficiente a coeficiente em $\mathbb{F}_p$, e
- para multiplicar dois polinômios,
  1. Multiplica os polinômios, e
  2. Divide com resto por um polinômio $m(\mathrm{X})$ a ser definido.

O `Rijndael` Corpo $\mathbb{F}_{2^8}$. Por exemplo, para $q = p^n$ com $p = 2$ e $n = 8$, obtemos o corpo $\mathbb{F}_{2^8}$ usado no `AES`. Como conjunto

$$\mathbb{F}_q = \{ a_7\mathrm{X}^7 + \cdots + a_0 \text{ para } a_7, \ldots, a_0 \text{ em } \mathbb{F}_2\}.$$

isto é, as somas finitas $a_0 + a_1\mathrm{X} + a_2\mathrm{X}^2 + \cdots + a_7\mathrm{X}^7$ para a_0, a_1, …, a_7 em $\{0,1\}$.

- A adição + de dois polinômios é a adição de polinômios, isto é, a adição coeficiente a coeficiente em $\mathbb{F}_p$, e
- A multiplicação é primeiro a multiplicação de polinômios e depois a divisão com resto pelo polinômio

  $$m(\mathrm{X}) = \mathrm{X}^8 + \mathrm{X}^4 + \mathrm{X}^3 + \mathrm{X} + 1.$$

  Por exemplo, a multiplicação rende

  $$(\mathrm{X}^6+\mathrm{X}^4+\mathrm{X}^2+\mathrm{X}+1)(\mathrm{X}^7+\mathrm{X}+1) = \mathrm{X}^{13}+\mathrm{X}^{11}+\mathrm{X}^9+\mathrm{X}^8+\mathrm{X}^6+\mathrm{X}^5+\mathrm{X}^4+\mathrm{X}^3+1$$

e a divisão com resto por $m(\mathrm{X})$

$$\mathrm{X}^{13}+\mathrm{X}^{11}+\mathrm{X}^{9}+\mathrm{X}^{8}+\mathrm{X}^{6}+\mathrm{X}^{5}+\mathrm{X}^{4}+\mathrm{X}^{3}+1 = (\mathrm{X}^{5}+\mathrm{X}^{3}+1)(\mathrm{X}^{8}+\mathrm{X}^{4}+\mathrm{X}^{3}+\mathrm{X}+1)+\mathrm{X}^{7}+\mathrm{X}^{6}+1$$

Isto é,

$$(\mathrm{X}^6 + \mathrm{X}^4 + \mathrm{X}^2 + \mathrm{X} + 1)(\mathrm{X}^7 + \mathrm{X} + 1) = \mathrm{X}^7 + \mathrm{X}^6 + 1 \quad \text{em } \mathbb{F}_{2^8}.$$

O corpo $\mathbb{F}_{p^n}$ para um primo p. Seja $q = p^n$ para p primo e q em $\mathbb{N}$. Como conjunto,

$$\mathbb{F}_q := \{a_{n-1}\mathrm{X}^{n-1} + \cdots + a_0 : a_{n-1}, \ldots, a_0 \text{ em } \mathbb{F}_p\}.$$

isto é, as somas finitas $a_0 + a_1\mathrm{X} + a_2\mathrm{X}^2 + \cdots + a_{n-1}\mathrm{X}^{n-1}$ para a_0, a_1, …, a_{n-1} em $\{0, 1, \ldots, m-1\}$.

- A adição + de dois polinômios é a adição de polinômios, isto é, a adição coeficiente a coeficiente em $\mathbb{F}_p$, e
- a multiplicação de dois polinômios é primeiro a multiplicação de polinômios e depois a divisão com resto por um polinômio $m(\mathrm{X})$ onde
  - para $p > 2$, este polinômio $m(\mathrm{X})$ é da forma $m(\mathrm{X}) = \mathrm{X} + a\mathrm{X} + b$ para a e b em $\mathbb{F}_p$, e
  - para $p = 2$,
    * ou existe um polinômio da forma $m(\mathrm{X}) = \mathrm{X}^n + \mathrm{X}^a + 1$ com $k > 2$,
    * ou, caso contrário, um da forma $m(\mathrm{X}) = \mathrm{X}^n + \mathrm{X}^a + \mathrm{X}^b + \mathrm{X}^c + 1$ (como no caso do corpo $\mathbb{F}_{2^8}$ usado no AES).

9.2 Curvas Elípticas

Uma curva E sobre um corpo (de característica $\neq 2, 3$) é *elíptica* se dada por uma equação

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

para coeficientes a e b tais que a curva não seja *singular*, isto é, que a sua *discriminante* não se esvaía, $4a^3 + 27b^2 \neq 0$.

Nota.

- A equação $y^2 = x^3 + ax + b$ é a *forma de Weierstraß*, mas existem várias outras que se revelaram computacionalmente mais eficientes, como a de *Montgomery*

$$\mathrm{B}y^2 = x^3 + \mathrm{A}x^2 + x \quad \text{onde } \mathrm{B}(\mathrm{A}^2 - 4) \neq 0$$

que pode ser transformada em uma equação de Weierstrass pela substituição

$$(x, y) \mapsto (t, v) = \left(\frac{x}{\mathrm{B}} + \frac{\mathrm{A}}{3\mathrm{B}}, \frac{y}{\mathrm{B}}\right) \quad \text{com } a = \frac{3 - \mathrm{A}^2}{3\mathrm{B}^2} \text{ e } b = \frac{2\mathrm{A}^3 - 9\mathrm{A}}{27\mathrm{B}^3}$$

- Se a característica é 2, isto é, $\mathbb{F}_q$ com $q = 2^n$, a equação tem a forma $y^2 + cxy + dy = x^3 + ax + b$.

Após escolha de um domínio (por exemplo, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$ ou $\mathbb{F}_p$ para um número primo p) os pontos (x, y) que resolvem esta equação formam uma curva no plano sobre ele. Além dos pontos no plano, existe também o ponto no infinito (ou ponto *ideal*) que é denotado 0. Resumimos que, como conjunto, a curva elíptica é dada por

$$\mathrm{E} := \{ \text{ todos os pontos } (x, y) \text{ tais que } \mathrm{E}(x, y) = 0\} \cup \{0\}$$

Sobre um corpo finito $\mathbb{F}_q$, o número dos pontos #E é limitado por $q + 1 - t$ onde $t \leq 2\sqrt{q}$, isto é, assimptoticamente igual à cardinalidade $\#\mathbb{F}_q^* = q - 1$. (O algoritmo de Schoof computa #E em cerca de n^5 operações para $n = \log_2 q$ o número de dígitos binários de q.)

Curvas Contínuas e Finitas. Para o domínio $\mathbb{R}$, as curvas assumem as seguintes formas no plano real para diferentes parâmetros a e b:

Enquanto sobre os corpos finitos, obtemos um conjunto discreto de pontos (simétrico em volta do eixo horizontal médio).

Curvas usadas na Criptografia. Para o algoritmo criptográfico sobre esta curva ser *seguro*, isto é, a computação do logaritmo sobre ela demorar, existem restrições nas escolhas de $q = p^n$ e da curva elíptica (isto é, dos seus coeficientes definidores a e b), por exemplo,

Figura 79: Curvas elípticas reais

- que $q \geq 2^{224}$ (para ser tão seguro quanto o RSA com chaves de 2048 bits),
- que os coeficientes a e b
  - evitem que #{pontos sobre $\mathbb{F}_q$} = q) pela vulnerabilidade ao ataque de Smart, e
  - evitem que a curva seja supersingular pela vulnerabilidade ao ataque de Menezes, Okamoto e Vanstone, isto é, que #{pontos sobre $\mathbb{F}_q$} = $q+1$ para $p > 3$ (enquanto para $p = 2,3$ há exatamente três respetivamente quatro equações que definem curvas supersingulares).

A probabilidade que uma curva aleatoriamente gerada (isto é, cujos coeficientes a e b na equação $by^2 = x^3+ax^2+x$ sejam gerados aleatoriamente) seja vulnerável

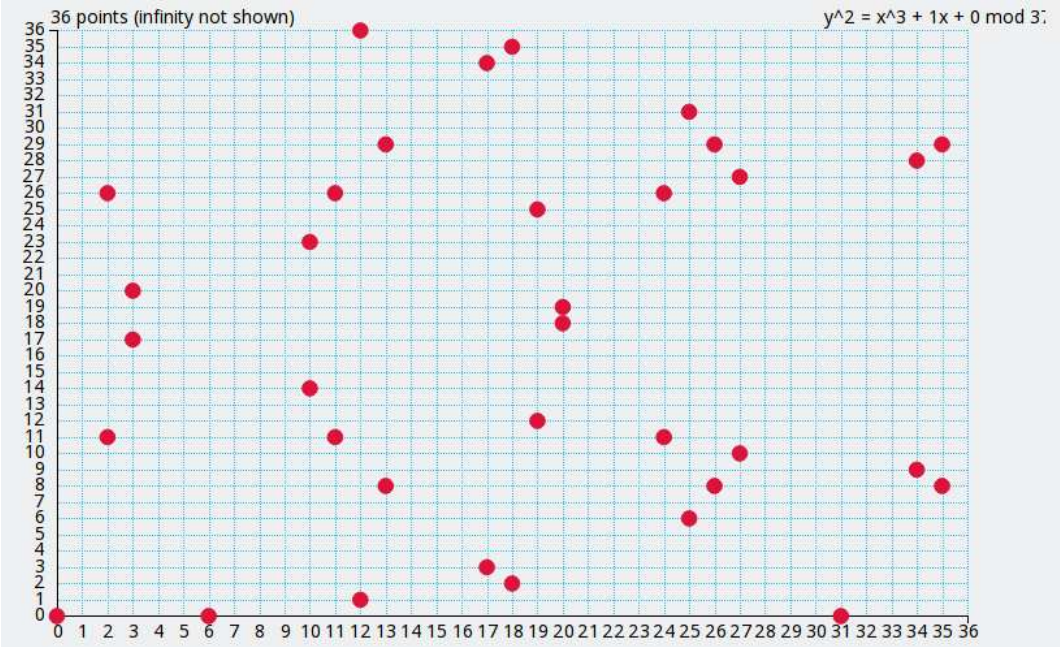


Figura 80: Curva elíptica finita

a um destes ataques é negligenciável.

Escolhas seguras são, entre outras:

- A `Curve25519` com $y^2 = x^3 + 486662x^2 + x$ sobre $\mathbb{F}_q$ com $q = p^2$ onde $p = 2^{255} - 19$ (de onde o seu nome); o seu número de pontos é #E = $2^{252} + 27742317777372353535851937790883648493$. (Esta curva tornou-se popular como alternativa imparcial às curvas recomendadas, e logo desconfiadas, pelo `NIST` o "National Institute for Standards and Technology".)
- Curvas de Edwards (de 2007) da forma $x^2 + xy = 1 + dx^2y^2$ para um inteiro $d \neq 0, 1$.
- Curvas de Koblitz e Curvas binárias sobre corpos binários de forma $x^2 + xy = x^3 + ax^2 + 1$ e $x^2 + xy = x^3 + x^2 + b$ para a e b em $\{0,1\}$ que permitem uma adição (e multiplicação) particularmente eficiente. Exemplos padronizados são, `nistk163`, `nistk283`, `nistk571` e `nistb163`, `nistb283`, `nistb571` definidas sobre o corpo binário com 163, 283 e 571 bites.

- A curva elíptica finita `Brainpool P256r` usada para cifrar os dados no micro-chip da carteira de identidade alemã.
- Para garantir que os coeficientes não fossem escolhidos tais que intencionalmente comprometam a segurança criptográfica, são frequentemente obtidos aleatoriamente, isto é,

    1. obtidos por um número gerado aleatoriamente (um *seed*), e
    2. transformados por um hash criptográfico como `SHA-1`.

9.3 Adição e Multiplicação Escalar

Os pontos de uma curva elíptica podem ser *somados*. Como é esta soma definida? Geometricamente a soma de três pontos p, q e r numa curva elíptica no plano euclidiano (isto é, em $\mathbb{R} \times \mathbb{R}$) é definida pela igualdade $p + q + r = 0$ se uma reta passa por p, q e r. Contudo, sobre corpos finitos esta igualdade geometria não se aplica mais e precisamos de uma definição algébrica (e a qual é além disto a única maneira que o computador entende).

Grupos Abstratos. Com efeito, os pontos da curva elíptica com a adição + formam um *grupo comutativo*. Um grupo comutativo é um conjunto G com

- uma *adição*, isto é, uma *operação binária* + que, dados dois argumentos P e Q, produz um resultado denotado por P + Q e a qual é
    - *associativa*: Para três elementos P, Q e R, vale P+(Q+R) = (P+Q)+R.
    - *comutativa*: Para dois argumentos P e Q, vale P + Q = Q + P.

e para a qual

- exista um elemento *neutro* 0, isto é, P + 0 = 0 + P = P para todo ponto P, e
- exista o *inverso*, isto é, para todo P em G, existe um −P em G tal que P + (−P) = 0

Exemplos.

- Os exemplos mais usuais são os conjuntos $\mathbb{Z}$, $\mathbb{Q}$ ou $\mathbb{R}$ com a adição + comum.
- O conjunto $\mathbb{N}$ com a adição *não* é um grupo porque o inverso da adição $-a$ falta!

- O leitor atento observa que todas as condições para um grupo são igualmente satisfeitas pela *multiplicação* $\cdot$ em, entre outros, $\mathbb{Q}^*$ ou $\mathbb{R}^*$ (onde $\cdot^*$ exclui 0).

O Grupo dos Pontos de uma Curva Elíptica. Dada uma curva elíptica, ela define um *grupo*

- cujos *elementos* são os pontos, isto é,

$$\mathrm{E} := \{ \text{ todos os pontos } (x,y) \text{ tais que } \mathrm{E}(x,y) = 0\} \cup \{0\}; \text{ e}$$

- cujo *elemento neutro* 0 é o ponto *ideal* infinito O, nas coordenadas homogêneas dado por $[0:1:0]$.
- o *inverso aditivo* $-\mathrm{P}$ de qualquer ponto P é o ponto obtido por reflexão de P à volta do eixo-x (onde usamos que a curva é simétrica em relação ao eixo-x).
- Geometricamente, no plano euclidiano (isto é, em $\mathbb{R}\times\mathbb{R}$) a *soma* (negativa) R de dois pontos P e Q é dada por $\mathrm{P}+\mathrm{Q}+\mathrm{R}=0$ se, e tão-somente se, uma reta passa por P, Q e R, exceto
  - se um dos pontos é O, então $\mathrm{P}+\mathrm{O}=\mathrm{P}=\mathrm{O}+\mathrm{P}$,
  - se P e Q forem opostos um ao outro, então $\mathrm{P}+\mathrm{Q}=\mathrm{O}$, e
  - se $\mathrm{P}=\mathrm{Q}$, então R é definido pelo ponto em que a linha *tangente* da curva em P intersecta a curva outra vez; exceto se P for um ponto de inflexão (um ponto onde a concavidade da curva muda), então R é definido por P.

Adição Geométrica (sobre os números reais). Se olhamos os pontos *reais* da curva E, isto é, todos os pontos (x,y) em $\mathbb{R}\times\mathbb{R}$ tais que $\mathrm{E}(x,y)=0$, então a adição tem uma significação *geométrica:* Tem-se $\mathrm{P}+\mathrm{Q}+\mathrm{R}=0$ se P, Q e R estão na mesma reta. Isto é,

- Se P e Q são desiguais, então a reta que liga P e Q intersecta a curva em outro ponto $-\mathrm{R}$;
- Se P e Q são iguais, então usamos a tangente no ponto $\mathrm{P}=\mathrm{Q}$.

O espelhamento de $-\mathrm{R}$ ao longo do eixo-x é o ponto $\mathrm{R}=\mathrm{P}+\mathrm{Q}$.

O CrypTool 1 demonstra esta adição geométrica em uma animação acessível na entrada Point Addition on Elliptic Curves no menu Indiv. Procedures -> Number Theory - Interactive.

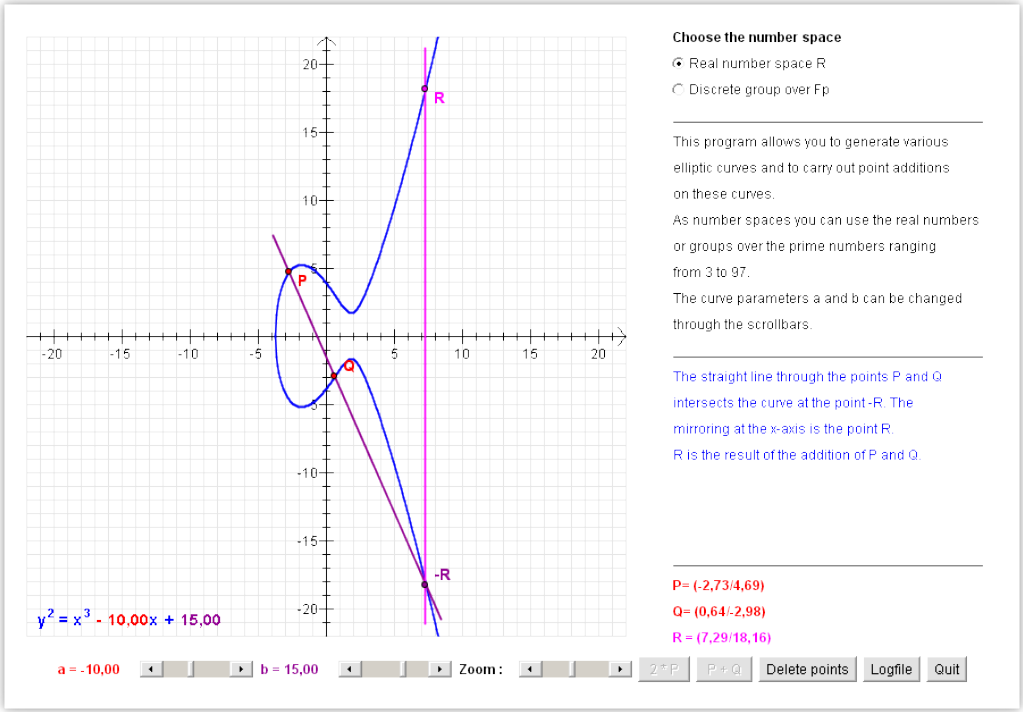


Figura 81: Adição de dois pontos sobre os números reais no CrypTool 1

Adição Algébrica (sobre um corpo finito). Esta descrição geométrica da adição nos leva à seguinte descrição algébrica: Expressa por coordenadas cartesianas a adição de dois pontos de uma curva elíptica é dada por uma fórmula *algébrica*, isto é, envolve apenas as operações básicas da adição, multiplicação e potenciação. (Logo, podemos substituir as incógnitas por valores em qualquer domínio, seja $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$ ou $\mathbb{F}_q$.)

Proposição: Denote

$$\mathrm{P} + \mathrm{Q} = \mathrm{R} \quad \text{por} \quad (x_\mathrm{P}, y_\mathrm{P}) + (x_\mathrm{Q}, y_\mathrm{Q}) = (x_\mathrm{R}, y_\mathrm{R}).$$

Se a curve E é dada por $\mathrm{Y}^2 = \mathrm{X}^3 + a\mathrm{X} + b$ e os pontos P, Q e R são não-nulos, então

$$x_\mathrm{R} = s^2 - x_\mathrm{P} - x_\mathrm{Q} \quad \text{e} \quad y_\mathrm{R} = s(x_\mathrm{P} - x_\mathrm{R}) - y_\mathrm{P} \quad (*)$$

onde s é o grau da inclinação da reta que passa por P e Q, dada por

$$s = \frac{y_Q - y_P}{x_Q - x_P} \quad \text{caso } x_Q \neq x_P, \quad \text{e} \quad s = \frac{3x_P^2 + a}{2y_P} \quad \text{caso } x_Q = x_P.$$

Demonstração: Dada a curva $Y^2 = X^3 + aX + b$ sobre o corpo K (cuja característica seja $\neq 2, 3$), e os pontos $P = (x_P, y_P)$ e $Q = (x_Q, y_Q)$ na curva.

1. Se $x_P \neq x_Q$, então seja $y = sx + d$ a equação cartesiana da linha que intersecta P e Q com a inclinação

   $$s = \frac{y_P - y_Q}{x_P - x_Q}.$$

   Para os valores x_P, x_Q e x_R a equação da linha é igual à da curva

   $$(sx + d)^2 = x^3 + ax + b,$$

   ou, equivalentemente,

   $$0 = x^3 - s^2x^2 - 2xd + ax + b - d^2.$$

   As raízes dessa equação são exatamente x_P, x_Q e x_R; logo

   $$(x-x_P)(x-x_Q)(x-x_R) = x^3+x^2(-x_P-x_Q-x_R)+x(x_Px_Q+x_Px_R+x_Qx_R)-x_Px_Qx_R$$

   Logo, o coeficiente de x^2 dá o valor x_R; o valor de y_R segue por substituir x_R na equação cartesiana da linha. Concluímos que as coordenadas $(x_R, y_R) = R = -(P + Q)$ são

   $$x_R = s^2 - x_P - x_Q \quad \text{e} \quad y_R = y_P + s(x_R - x_P).$$

2. Se $x_P = x_Q$, então

   1. ou $y_P = -y_Q$, incluindo o caso em que $y_P = y_Q = 0$, então a soma é definida como o; assim, o inverso de cada ponto na curva é encontrado refletindo-o no eixo-x,
   2. ou $y_P = y_Q$, então $Q = P$ e $R = (x_R, y_R) = -(P + P) = -2P = -2Q$ é dado por

      $$x_R = s^2 - 2x_P \quad \text{e} \quad y_R = y_P + s(x_R - x_P) \quad \text{com } s = \frac{3{x_P}^2 + a}{2y_P}.$$

Observação. Para certas curvas específicas, estas fórmulas podem ser simplificadas: Por exemplo, numa Curva de Edwards da forma

$$x^2 + xy = 1 + dx^2y^2$$

para $d \neq 0{,}1$ (com elemento neutro 0 o ponto $(0,\ 1)$), a soma dos pontos $p = (x_P, y_P)$ e $q = (x_Q, y_Q)$ é dada pela fórmula

$$(x_P, y_P) + (x_Q, y_Q) = \left(\frac{x_P y_Q + x_Q y_P}{1 + d x_P x_Q y_P y_Q}, \frac{y_P y_Q - x_P x_Q}{1 - d x_P x_Q y_P y_Q} \right)$$

(e o inverso de um ponto (x, y) é $(-x, y)$). Se d não é um quadrado em $\mathbb{F}_Q$, então não existem pontos *excecionais*: Os denominadores $1 + dx_P x_Q y_P y_Q$ e $1 - dx_P x_Q y_P y_Q$ são sempre diferentes de zero.

Se, em vez de $\mathbb{R}$, olhamos os pontos com entradas em um corpo finito $\mathbb{F}_q$ da curva E, isto é, todos os pontos (x, y) em $\mathbb{F}_q \times \mathbb{F}_q$ tais que $E(x, y) = 0$, a adição é univocamente definida pela fórmula $(*)$.

O `CrypTool 1` demonstra esta adição na entrada `Point Addition on Elliptic Curves` no menu `Indiv. Procedures -> Number Theory - Interactive`.

Multiplicação Escalar.

A adição leva à *multiplicação escalar* pela iterada adição. Isto é,

- à *exponencial*, dado x em $\mathbb{N}$, para P um ponto da curva elíptica, seja $x \cdot P := P + \cdots + P$ (iterado d vezes) por um número natural d;
- e ao *logaritmo*: dado Y e P pontos da curva elíptica, qual é a x em $\mathbb{N}$ tal que $Y = x \cdot P$?

Ponto de Base.

Como o grupo de pontos sobre um corpo finito $\mathbb{F}_q$ é finito (de cardinalidade aproximativamente q), necessariamente para qualquer ponto P o conjunto $\langle P \rangle := \{P, 2P, \ldots\}$ é finito. Isto é, existem n e $n+m$ em $\{0, 1, \ldots., q-1\}$ tais que $nP = (n + m)P$, isto é, existe um inteiro $m < q$ tal que $mP = 0$.

Nota. O site http://www.graui.de/code/elliptic2/ mostra para uma curva elíptica sobre um corpo finito da forma $\mathbb{F}_p$ a tabela de adição entre os pontos, e para cada ponto P o grupo finito $\langle P \rangle$ gerado por ele.

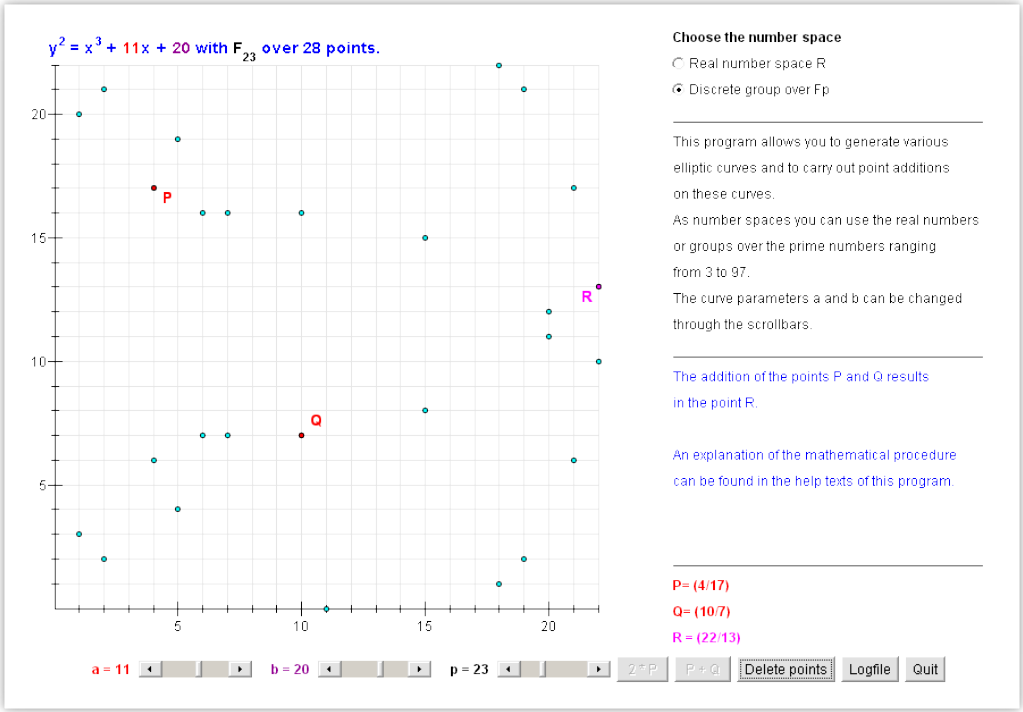


Figura 82: Adição de dois pontos sobre um corpo finito no `CrypTool 1`

A cardinalidade $n = \#\langle P \rangle$ é o menor m tal que $mP = 0$ e é chamada de *ordem* do ponto P. Pelo Teorema de Lagrange (em Seção 7.2), n divide #E.

Para cifrar, além

- do primo p que define o corpo $\mathbb{F}_p$, e
- dos coeficientes a e b que definem os pontos E no plano sobre $\mathbb{F}_p$ (sujeitos à condição $4a^3 + 27b^2 \not\equiv 0 \bmod p$ para não ser singular)

é escolhido

- um *ponto de base* G em E.

O número criptograficamente mais importante é a ordem n do ponto de base G, que, para resistir

- contra o Pohlig-Hellman attack (que reduz o problema aos seus fatores primos) deve ser primo

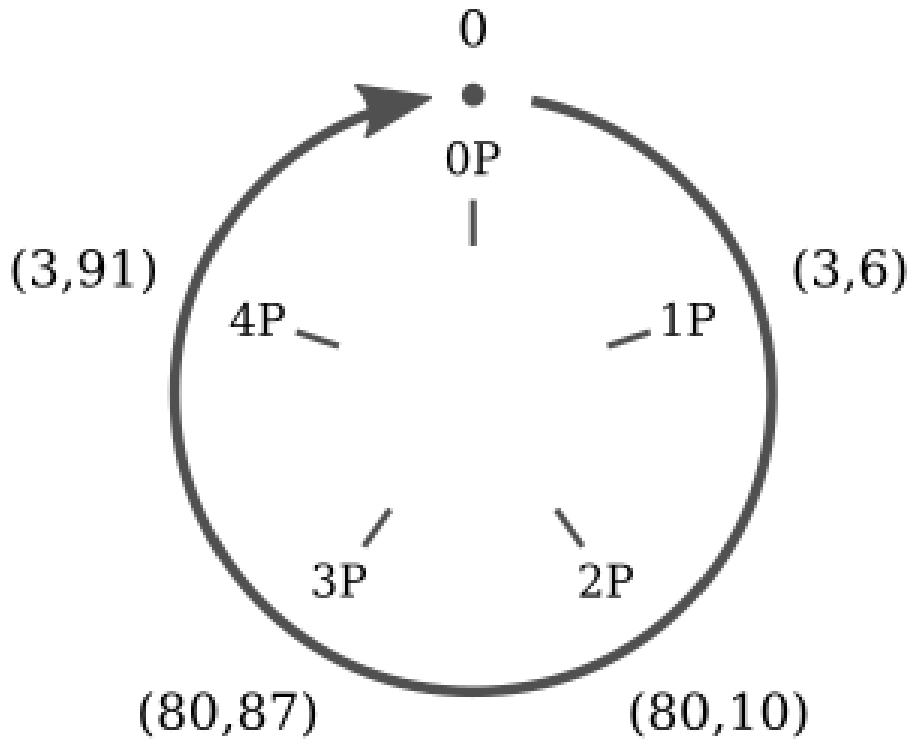


Figura 83: Ciclidade de um ponto numa curva eliptica finita

- contra o Pollard's rho attack deve ser $\geq 2^{224}$ (para torná-lo computacionalmente inviável).

Para encontrar um ponto de base G de uma ordem n suficientemente grande e com a ordem do corpo q suficiente grande (para resistir contra o Pollard's ρ attack), executa os passos seguintes:

1. Escolhe aleatoriamente coeficientes a e b em $\mathbb{F}_q$.
2. Computa N = #E($\mathbb{F}_q$) pelo algoritmo de Schoof.
3. Verifica que N $\neq q, q+1$ (para evitar os ataques de Smart e Menezes, Okamoto e Vanstone). Caso contrário, volta ao primeiro passo.
4. Verifica se exista um fator *primo* n de N tal que
    - $n > 2^{224}$, e
    - $n > 4\sqrt{q}$ (para evitar o ataque de Pohlig-Hellman). Caso contrário, volta ao primeiro passo.

5. Escolhe aleatoriamente pontos g em E até $G = hG \neq 0$ para $h := N/n$.

Proposição. O ponto assim encontrado tem ordem n.

Demonstração: Para qualquer ponto P, vale $\#EP = nhP = 0$ pelo Teorema de Lagrange. Isto é, $nG = 0$ para $G = hP$, e pelo Teorema de Lagrange $\#\langle G\rangle$ divide n. Como n é primo, ou $\#\langle G\rangle = 1$, ou $\#\langle G\rangle = p$. Como $G \neq 1$, vale $\#\langle G\rangle > 1$, isto é, $= n$.

Exemplos (de pontos de base).

- A curva elíptica `Curve25519` com $y^2 = x^3 + 486662x^2 + x$ sobre $\mathbb{F}_q$ com $q = p^2$ onde $p = 2^{255} - 19$, usa como ponto de base $G = (x_G, y_G)$ univocamente determinado

  – por $x_G = 9$
  – (e $y_G < q/2$, isto é, e y_G = 14781619 44758954 47910205 93568409 98688726 46061346 16475288 96488183 77555862 37401).

  (Uns pontos com $x_G < 9$ tem uma ordem consideravelmente menor que a de G.)

- A curva elíptica `secp256k1` definida pela equação $y^2 = x^3 + 7x$ sobre $\mathbb{F}_p$ com p = 0xffffffffffff ffffffffffff ffffffffffff fffffefffffc2f em notação hexadecimal (especificada pelo `SECG`, o "Standards for Efficient Cryptography Group", implementada por OpenSSL, e, entre outros, usada pelo `Bitcoin` para assinar transações) usa como ponto de base $G = (x_G, y_G)$ com

  – x_G = 0x79be667e f9dcbbac 55a06295 ce870b07 029bfcdb 2dce28d9 59f2815b 16f81798
  – y_G = 0x483ada77 26a3c465 5da4fbfc 0e1108a8 fd17b448 a6855419 9c47d08f fb10d4b8, e
  – cuja ordem, isto é, o números dos pontos da curva elíptica, é n = 0xffffff ffffff ffffff fffffe baaedce6 af48a03b bfd25e8c d0364141.

9.4 Os algoritmos usando ECC

A criptografia por curvas elípticas ECC (Elliptic Curve Criptography) usa a troca de Chaves de Diffie-Hellman para

1. estabelecer uma chave secreta, para
2. transformá-la em um hash criptográfico, para
3. usá-lo para cifrar a comunicação por um algoritmo criptográfico simétrico.

A cifração pela ECC é padronizada pelo ECIES (Elliptic Curve Integrated Encryption Scheme), um procedimento *híbrido* (mistura criptografia assimétrica com criptografia simétrica).

Uma vez a chave (= um ponto da curva elíptica finita) secreta mútua c estabelecida, Alice e Bob derivam dela uma chave para uma cifra simétrica como AES ou 3DES. (Esta derivação chama se uma KDF (Key Derivation Function) função de derivação de chave que transforma uma informação secreta no formato apropriado. Uma tal função padronizada é a ANSI-X9.63-KDF com a opção SHA-1.) Por exemplo, o protocolo TLS

- usa a coordenada x do ponto c,
- concatena a ela números relativos à conexão (um tal acréscimo publicamente conhecido é chamado de *sal*), e
- calcula um hash criptográfico deste número concatenado.

Troca de Chaves. Transferimos o protocolo de Diffie-Hellman da multiplicação em um corpo finito à adição em uma curva elíptica finita: Denote G um ponto da curva, e $x\mathrm{G} = \mathrm{G} + \cdots + \mathrm{G}$ a iterada adição (x vezes) sobre a curva elíptica finita (em vez de g e $g^x = g \cdots g$ para o grupo multiplicativo de um corpo finito).

Passos. No protocolo ECDH (Elliptic Curve Diffie-Hellman), para Alice e Bob construírem uma chave secreta através de um canal inseguro, combinam primeiro

- uma potência q de um número primo p *apropriado*,
- uma curva elíptica E *apropriada* sobre $\mathbb{F}_q$, e
- um ponto G *apropriado* em E.

e depois

1. Alice, para gerar **uma metade** da chave, escolhe um número a,
   - calcula $\mathrm{A} \equiv a\mathrm{G}$, e
   - transmite A ao Bob.
2. Bob, para gerar **outra metade** da chave, escolhe um número b,
   - calcula $\mathrm{B} \equiv b\mathrm{G}$, e
   - transmite B à Alice.
3. A chave secreta **mútua** entre Alice e Bob é

   $$c := b\mathrm{A} = ba\mathrm{G} = ab\mathrm{G} = a\mathrm{B}$$

   Observamos que para ambos calcular o mesmo ponto c, a adição tem de ser

- associativa, e
- a comutativa;

isto é, é importante que E é um grupo.

O protocolo ECDHE, com o E adicional significando 'Ephemeral', é quanto à troca de chaves igual ao protocolo ECDH. 'Ephemeral' significa que as chaves são efémeras (e necessariamente assinadas pelas chaves permanentes).

Implementação em Python. Em https://github.com/andreacorbellini/ecc/blob/master/scripts/ecdhe.py encontra-se uma implementação em Python do ECDH.

Assinatura. **E**lliptic **C**urve **D**igital **S**ignature **A**lgorithm é uma modificação do algoritmo DSA que usa a iterada adição de dois pontos de uma curva elíptica no lugar da exponenciação de dois números usada pelo DSA. A principal vantagem deste algoritmo é que ele pode se contentar com chaves menores para oferecer a mesma segurança que DSA o RSA devido aos ataques (conhecidos) menos avançados.

Passos no ECDSA. Para a Samanta assinar um documento d e o Vitor verificar a assinatura, combinam primeiro

- uma (potência de um) número primo p *apropriado*,
- uma curva elíptica E sobre $\mathbb{F}_p$.
- um ponto G (que pertença a E) de ordem n.

1. Samanta, para **criar** uma rubrica (ou *firma*),
    1. escolhe um número f em $\{1, \ldots, n-1\}$,
    2. calcula $\mathrm{F} = f\mathrm{G}$.
    3. transmite F ao Vitor.
2. Samanta, para **assinar** o (hash do) documento d,
    1. escolhe um número efêmero e em $\{1, \ldots, n-1\}$,
    2. calcula a coordenada E_x mod n de $\mathrm{E} = e\mathrm{G}$; Si $\mathrm{E}_x = 0$, então escolhe outro e;
    3. resolve $d\mathrm{G} + \mathrm{E}_\mathrm{X}\mathrm{F} = \mathrm{SE}$ mod p para S, isto é, calcula $\mathrm{S} \equiv [d + \mathrm{E}_x f]/e$ mod n; se $d + \mathrm{E}_x f \equiv 0 \bmod n$, então escolhe outro e;
    4. transmite E_x e S ao Vitor.
3. Vitor, para **verificar** a assinatura,
    1. calcula $\mathrm{S}' = \mathrm{S}^{-1}$ mod n,
    2. calcula $d' = d \cdot \mathrm{S}'$ mod n,
    3. calcula $\mathrm{E}' = \mathrm{E}_x \cdot \mathrm{S}'$ mod n,
    4. calcula $v = d'\mathrm{G} + \mathrm{E}'\mathrm{F}$, e
    5. verifica que $v_x = \mathrm{E}_x$.

Observação. Observamos, que em comparação com o DSA, o algoritmo ECDSA

- trabalha com um ponto G na curva elíptica em vez de um número g em $\mathbb{Z}/p\mathbb{Z}$,
- trabalha com os múltiplos $\cdot\mathrm{G}$ de G em vez de potências $g^{\cdot}$ de g módulo q,
- conforme,
    - em vez de resolver $g^d\mathrm{F}^\mathrm{E} = \mathrm{E}^\mathrm{S}$ mod p para S, isto é, calcular $\mathrm{S} \equiv [d + \mathrm{E} \cdot f]/e$ mod p, resolve $d\mathrm{G} + \mathrm{E}_\mathrm{X}\mathrm{F} = \mathrm{SE}$ na curva elíptica para S, isto é, calcula $\mathrm{S} \equiv [d + \mathrm{E}_x f]/e$ mod n,

– em vez de verificar $g^d F^E = E^S \bmod p$, isto é, verificar $E = \sqrt[S]{D}\sqrt[S]{F^E} = g^{d'} F^{E'}$ para $d' = dS'$ e $E' = ES' \bmod q$ onde $S' = S^{-1}$, verifica $E_x = [d'G + E'F]_x$.

Advertência. Se um dos dois números secretos, ou f, ou e, for conhecido, então ambos, f e e, serão conhecidos (pela fórmula $S = [d + E_x f]/e$)! Em particular, se a chave secreta auxiliar (*efêmera*) e for conhecida, então a rubrica secreta (*permanente*) f será conhecida pela fórmula $d = [d + se]/E_x$! Logo, é importantíssimo que e guardar secreta; em particular, a sua geração tem de ser suficientemente aleatória.

Implementação em Python. Em https://github.com/andreacorbellini/ecc/blob/master/scripts/ecdsa.py encontra-se uma implementação em Python do ECDSA.

9.5 Segurança

A ECC, a criptografia por curvas elípticas finitas ganha mais e mais popularidade porque o seu problema criptográfico (o logaritmo sobre uma curva elíptica finita) é presentemente considerado computacionalmente mais difícil do que o do RSA (a fatoração em primos) ou do ElGamal (o logaritmo sobre o grupo multiplicativo de um corpo finito). Por isso, chaves pequenas para a ECC conseguem o mesmo nível de segurança como chaves grandes para o RSA ou o ElGamal. Como exemplo, a segurança obtida por uma chave de 224 bites da ECC corresponde à obtida por uma chave de 2048 bites do RSA ou ElGamal. Esta economia entre os tamanhos das chaves de um fator considerável corresponde a uma economia computacional de um fator semelhante. Quanto a usabilidade,

- uma tal chave pública da ECC pode ser compartilhada por soletração (são 56 letras na notação hexadecimal,
- enquanto uma chave pública do RSA ou ElGamal tem que ser compartilhada por um arquivo (a que é referida por uma impressão digital por conveniência).

Estudamos o problema criptográfico por trás da ECC em mais detalhes e comparamo-lo com o do RSA e El Gamal.

Exponencial e Logaritmo Genérico. Ambos os grupos considerados na criptografia, o grupo multiplicativo de um corpo finito e o grupo de uma curva elíptica finita, são grupos *cíclicos finitos*, isto é, *gerados* por um elemento g de ordem *finita* n. Matematicamente dito, são iguais a um grupo do tipo

$$\mathrm{G} = \langle g \rangle = \{g^0, g, g^2, ..., g^{n-1}\} \simeq \{0, 1, 2, ..., n-1\} = \mathbb{Z}/n\mathbb{Z}$$

Exponencial. Dado um gerador g em G, a identificação de $\mathbb{Z}/n\mathbb{Z}$ com G por $x \mapsto g^x$, a *exponencial*

$$\exp\colon \mathbb{Z}/n\mathbb{Z} \to \mathrm{G}$$

é rapidamente computado em *qualquer* grupo comutativo G: Dado d em $\{1, ..., n-1\}$, para calcular g^d, expande d na base binária

$$d = d_0 + 2d_1 + 2^2 d_2 + \cdots + 2^m d_m$$

para $d_0, \ldots, d_m$ em $\{0,1\}$ e calcula por multiplicação por 2 sucessivamente g^2, $g^{2^2}, \ldots, g^{2^m}$. Então

$$g^d = g^{d_0 + 2d_1 + 2^2 d_2 + \cdots + 2^m d_m} = g^{d_0} g^{2d_1} g^{2^2 d_2} \cdots g^{2^m d_m}.$$

Isto é, calculamos g^d em $\leq 2\log_2 n$ operações do grupo.

Exemplo. Para G o grupo de pontos de uma curva elíptica, P em G, e d em $\mathbb{N}$ calculamos sucessivamente 2P, $2^2 = 2 \cdot \mathrm{P}$, $2^3\mathrm{P} = 2 \cdot 2^2\mathrm{P}$. Sejam i_0, i_1, ... os índices dos dígitos binários d_{i_0}, d_{i_1}, ... que são diferentes de 0. Então

$$d\mathrm{P} = 2^{i_0}\mathrm{P} + 2^{i_1}\mathrm{P} + \cdots$$

Logaritmo. Porém, dado um gerador g em G, a computação da identificação inversa, do *logaritmo*

$$\log\colon \mathrm{G} \to \mathbb{Z}/n\mathbb{Z},$$

isto é, dado y em G, calcular x em $\{0, ..., n-1\}$ tal que $y = g^x$ é geralmente árdua: Pelo Teorema de Shoup em Shoup (1997), todo algoritmo *genérico*, isto é, que usa apenas as operações do grupo, leva pelo menos $\sqrt{n}$ operações (do grupo) para calcular o logaritmo.

Um algoritmo genérico que atinge esta rapidez (exceto um fator 2) é o `Baby Step, Giant Step` (ou algoritmo de Shanks): Dado h em $\langle g \rangle$, para calcular d em $\{0, 1, ..., n-1\}$ tal que $h = g^d$:

1. Põe $m := \lceil \sqrt{n-1} \rceil$ (onde $\lceil \cdot \rceil$ indica o menor inteiro acima ou igual ao número real $\cdot$).
2. Calcula $\alpha_0 = g^0$, $\alpha_1 = g^m$, $\alpha_2 = g^{2m}$, ..., $\alpha_{m-1} = g^{(m-1)m}$. (O passo gigantesco)
3. Calcula $\beta_0 = h$, $\beta_1 = hg^{-1}$, $\beta_2 = hg^{-2}$, ..., $hg^{-(m-1)}$ até encontrar algum β_i que seja igual a algum α_j. (O passo bebé)
4. Se $\beta_i = \alpha_j$, então o resultado é $d = jm + i$.

Este algoritmo funciona, pois:

- Todo d em $\{0, ..., m^2-1\}$ é pela divisão com resto por m da forma $d = qm+r$ para $q, r < m$; logo todo elemento h em $\langle g \rangle$ é da forma $g^{qm+r} = h$;
- Se $\beta_i = \alpha_j$, então $g^{jm} = hg^{-i}$, logo $g^{jm+i} = h$.

Nota. Para curvas elípticas, o Pollard's ρ-algorithm é levemente mais rápido.

Algoritmos Genéricos e Específicos. Esta estimativa das operações necessárias para computar o logaritmo de um grupo, isto é, que usa apenas as operações do grupo, aplica-se a algoritmos *genéricos*. Contudo, algoritmos *específicos*, isto é, que usam as propriedades específicos do grupo (como o das unidades de um corpo finito ou o de uma curva elíptica), podem usar menos.

Por exemplo, para o grupo multiplicativo de um corpo finito, existem, com efeito, algoritmos mais rápidos (chamados *sub-exponenciais*) para calcular a fatoração e o logaritmo. São baseados no *Index Calculus* que faz uso das propriedades deste grupo específico. Por exemplo, para os problemas criptográficos do `RSA` e `Diffie-Hellman` sobre o grupo multiplicativo de um corpo finito, isto é,

- ou a fatoração de um inteiro em seus fatores primos,
- ou o logaritmo do grupo multiplicativo de um corpo finito,

o algoritmo presentemente mais rápido é o *Campo de número de peneira geral* (de Lenstra et al. (1993) para a fatoração, e de Gordon (1993) para o logaritmo).

A grosso modo, a sua *complexidade*, o número de operações do grupo, (dita *sub-exponencial*) para um grande n é

$$2^{n^{1/3}(\mathrm{C}+o(1))(\log n)^{2/3}}$$

onde $\mathrm{C} = (64/9)^{1/3} \approx 1.92$ e $o(1)$ significa uma função f sobre $\mathbf{N}$ tal que $f(n) \to 0$ para $n \to \infty$.

Em contraste, todos os algoritmos conhecidos para o problema criptográfico da ECC, isto é, calcular o logaritmo sobre uma curva elíptica finita, são algoritmos genéricos. Para estes algoritmos genéricos, pelo Teorema de Shoup, a complexidade $2^{n/2}$ no número de bites n da entrada é a menor possível. O algoritmo presentemente mais rápido é o Pollard's ρ-algorithm que tem a grosso modo uma complexidade (dita *exponencial*) de

$$2^{n/2+C}$$

onde $C = \log_2(\pi/4)^{1/2} \simeq -0.175$.

Comparação entre os Tamanhos de Chaves. Esta Tabela compara os tamanhos de chaves em bites a um nível de segurança comparável entre

- um algoritmo simétrico como o AES,
- um algoritmo assimétrico por curvas elípticas, e
- um algoritmo assimétrico como o Diffie-Hellman ou o RSA.

Chave Simétrica	Chave Assimétrica Elíptica	Chave Assimétrica Clássica
80	160	1024
112	224	2048
128	256	3072
192	384	7680
256	512	15360

Os números da tabela são estimados pelo algoritmo, presentemente, mais rápido para resolver os seus problemas criptográficos: Dada uma chave de entrada de n bites,

- para um algoritmo simétrico como o AES, o algoritmo mais rápido é provar todas as possíveis chaves, cuja complexidade (= o número de operações) é
$$2^n;$$
- para o logaritmo sobre uma curve elíptica finita, o algoritmo mais rápido é atualmente o Pollard's ρ-algorithm cuja complexidade a grosso modo é
$$\sqrt{n}.$$

- para o algoritmo assimétrico, ou RSA, ou Diffie-Hellman, sobre o grupo multiplicativo de um corpo finito, o algoritmo mais rápido é o *Campo de número de peneira geral* (baseado no Index Calculus) cuja complexidade, a grosso modo, para um grande n é

$$2^{n^{1/3}}.$$

Na prática, estas chaves menores da ECC aceleram por um fator > 5 as operações criptográficas em comparação às do RSA e ElGamal (além de facilitarem a sua troca entre homens e economizam largura de banda). Contudo, recordemo-nos que existem também desvantagens da ECC em comparação ao RSA:

- A assinatura depende de um Pseudo Random Number Generator, um gerador de números aleatórios que, quando é mal programado, revela a chave privada!
- A ECC é mais recente; por isso:
  - menos comprovada do que o RSA, e
  - umas implementações ainda são patenteadas (por Certicom).

10 Função Hash

Uma função (*hash* ou de *dispersão* ou de *espalhamento*) é um algoritmo que envia

- entradas (diferentes) de tamanho *variável*
- a saídas (praticamente sempre diferentes) de um tamanho *fixo*;

isto é, mapeia uma sequência de baites de um comprimento *arbitrariamente* grande para uma sequência de baites de comprimento *fixo*, usualmente entre 128 e 512 bites; conceptualmente, transforma uma grande quantidade de *dados* em uma pequena quantidade de *informações.*

Para ser uma função hash razoável, precisa de

- *difundir* bem, isto é, a inversão de um bite da entrada implica a inversão de cerca da metade dos bites da saída (uma tal funções de hash é dita satisfazer o critério *estrito* do *efeito de avalanche*), isto é, cada bite da saída deve depender de cada bite da entrada,
- ser *sobrejetora*, isto é, todas as sequências de comprimento fixo possíveis são valor da função hash, e
- semelhar a uma *variável uniformemente aleatória*, isto é, a probabilidade de cada um dos valores da função é a mesma.

A função de hash é *criptográfica* (ou uma *função de embaralhamento*) se o algoritmo resiste

- contra a criação de *uma imagem inversa* (isto é, a função é *unidirecional*): dada uma saída, a maneira mais rápida para achar uma entrada com esta saída é por força bruta, isto é, por provar todas as possíveis entradas,
- contra a criação de *uma segunda imagem inversa*: dada uma entrada, a maneira mais rápida para achar *outra* entrada com a mesma saída é por força bruta, isto é, por provar todas as possíveis entradas,
- contra a criação de *colisões*: a maneira mais rápido para achar duas entradas (arbitrárias) com a mesma saída é por força bruta, isto é, por provar todas as possíveis entradas.

Segundo o princípio de Kerckhof, o algoritmo deve ser público.

Na prática, a resistência

- mais importante é a contra a criação de uma segunda imagem inversa,e
- a menos importante é a contra colisões; existem vários algoritmos, por exemplo, `MD4`, `MD5` e `SHA-1` que não resistem contra colisões, mas continuam a ser populares.

Por exemplo, o algoritmo `CRC` é uma função de hash (não criptográfica); funções de hash criptográficas comuns são, por exemplo, `MD4`, `MD5`, `SHA-1`, `SHA-256` e `SHA-3`.

O valor de uma função *hash* é chamado de *hash* ou *checksum* (= *soma de verificação*) ou *dispersão* ou *escrutínio.* Frequentemente, tem um comprimento de 16 ou 32 baites e é representada em base hexadecimal cujos algarismos são de 0 a 9 e de A a F.

Por exemplo, o valor da função hash `SHA-256` da entrada `Oi!` é

```
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
```

10.1 Descrição

Lembremo-nos de que uma função hash razoável é

- *sobrejetora*, isto é, todas as sequências de comprimento fixo possíveis são valor da função hash, e
- semelha a uma *variável uniformemente aleatória*, isto é, a probabilidade de cada um dos valores da função é a mesma.

Logo, se, por exemplo, a saída tem 256 bites, então idealmente cada valor deveria ter a mesma probabilidade 2^{-256}. Isto é, a saída identifica a entrada praticamente *univocamente* (com uma chance de colisão de idealmente 2^{-256}); Por isso, convém pensar de um hash de dados, por exemplo, de um arquivo, como a sua (carteira de) *identidade* (ou mais exatamente, o número da identidade, ou `CPF`, já que o hash é um número); um hash identifica *muitos* dados por *pouca* informação.

Como o comprimento a sequência do hash é limitada (raramente ≥ 512 bites) enquanto o comprimento a sequência da entrada é ilimitada, *existem colisões*, isto é, hashes iguais de arquivos diferentes. Porém, o algoritmo minimiza a probabilidade de colisões pela distribuição mais uniforme possível dos seus

Figura 84: Identidade

valores: Intuitivamente, torná-los o mais aleatórios possíveis; mais exatamente, toda sequência de comprimento fixo possível é um valor e a probabilidade de cada um dos valores é a mesma.

Os algoritmos de hash (criptográficos) mais usados são os de 16 baites (= 128 bites) `MD4` e `MD5` ou o `SHA-1`, de 20 baites (= 160 bites). Apesar de todos eles, `MD4`, `MD5` e `SHA-1` não resistem contra colisões, continuam populares para usos sem alto risco.

- `MD4` (`Message-Digest algorithm`): Desenvolvido em 1991 por Ron Rivest, um dos três criadores do algoritmo `RSA` e (da empresa `RSA Data Security`); rápido, mas vulnerável a criação de pré-imagens. Descrito na `RFC 1320`.
- `MD5`: Desenvolvido pela `RSA Data Security`. Descrito na `RFC 132`. Enquanto é vulnerável a colisões, mas não à criação de uma segunda imagem inversa. Muito utilizado para

  - a verificação de integridade, por softwares com protocolo par-a-par (P2P, ou Peer-to-Peer, em inglês), e
  - para o armazenamento de senhas.

- SHA-1 (*Secure Hash Algorithm*): Desenvolvido pelo NIST, o National Institute for Standards and Technology. É vulnerável a colisões, mas não à criação de uma segunda imagem inversa.

- WHIRLPOOL: função criptográfica de hash desenvolvida por Paulo S. L. M. Barreto e por Vincent Rijmen, o co-autor do AES; relativamente recente, e até hoje, em 2018, considerada segura. Foi adotada como parte do padrão internacional ISO 10118-3.

Atualmente recomendáveis são, entre outros, as versões mais recentes SHA-256 e SHA-3 do Secure Hash Algorithm.

10.2 Exemplos Simplistas

Damos exemplos de funções hash ruins, isto é,

- não difundem bem, isto é, a inversão de um bite da entrada não implica a inversão de cerca da metade dos bites da saída (acontece, por exemplo, se a sequência entrante é simplesmente truncada), e
- não são criptográficas, isto é, a saída permite deduzir informações sobre a entrada; em particular, dada uma saída, facilmente se constroem entradas com esta saída.

Para a construção de funções hash (criptográficas) atuais como MD4 olha, por exemplo, a Tese de Diploma por Daniel Knopf. O embaralhamento é conseguido por iterações

- de substituição e
- de transposição,

semelhante às cifras de bloco como o AES.

- *Divisão com resto* (ou congruência linear) por um número (natural) m,

$$h(k) = r \quad \text{onde } k = qm + r \text{ e } r \text{ em } \{0, \ldots, m-1\}.$$

Por exemplo, para $k = 1000$ e $m = 13$ obtemos

$$h(1000) = 7$$

pois $1000 = 61 * 13 + 7$.

Quando $m = 2^n$, a divisão com resto é o truncamento da expansão binária de k à sua soma incial até a n-ésima entrada; por exemplo, para $n = 2$

$$b_0 + b_1 2 + b_2 2^2 + b_3 2^3 + \cdots \mapsto b_0 + b_1 2.$$

- *Soma dos dígitos* (binários), isto é,

$$b_0 + b_1 2 + b_2 2^2 + b_3 2^3 + \cdots \mapsto b_0 + b_1 + b_2 + b_3 + \cdots$$

- As duas funções podem ser compostas, isto é,

  1. a soma dos algarismos, e
  2. o resto da soma dividido por m.

  Por exemplo, quando m é uma potência de 2, o truncamento da soma dos dígitos binários a certo número de dígitos binários.

10.3 Exemplos Modernos

O meta-método de Merkle, ou a construção Merkle-Damgård, constrói a partir de uma função de compressão sem perda (isto é, cujas entradas têm o mesmo comprimento como a saída) uma função de compressão com perda, isto é, uma função de hash. Ele permite reduzir à imunidade da *função hash inteira* contra

- a criação de *uma imagem inversa*, e
- a criação de *colisões*

à correspondente imunidade da *função de compressão.*

Esquema de Merkle-Damgard. Esta construção precisa de

- um *valor inicial* (IV = initialization value),
- uma *função de compressão* C, e
- um *acolchoamento* para aumentar a entrada m da função de hash a $\bar{m}$ tal que $\bar{m} = (m_1, m_2, \ldots)$ consiste de um múltiplo de blocos m_1, m_2, ... que são aceitos pela função de compressão.

Com $h_0 = \text{IV}$ são calculados para $i = 1, 2, \ldots$

$$h_i = \text{C}(m_i, h_{i-1}).$$

Isto é, na computação do hash do bloco atual entra, além do valor do bloco atual, o valor da função de compressão do último bloco.

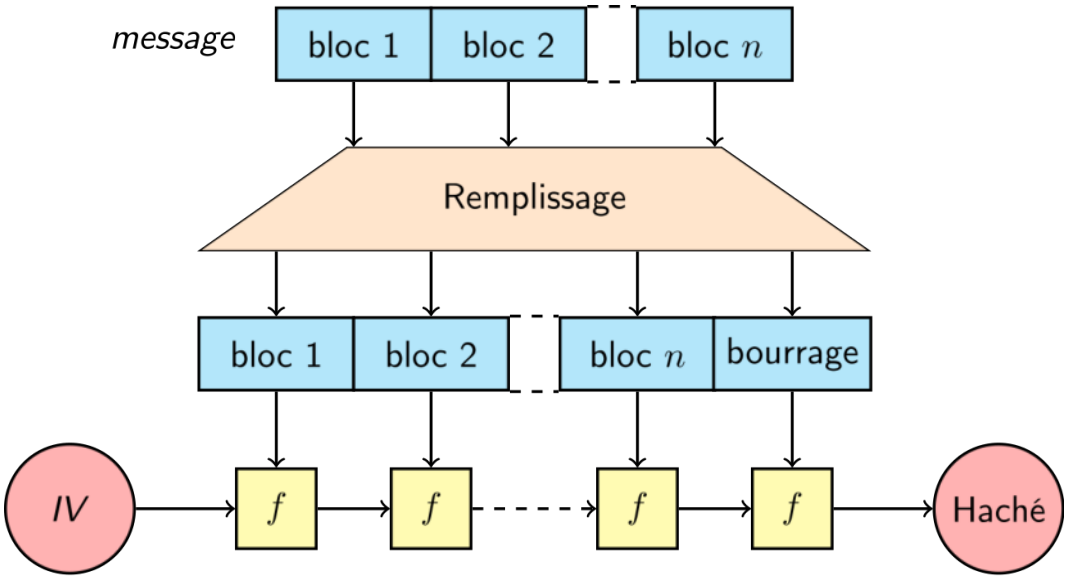


Figura 85: O meta-método de Merkle

A função de compressão C consiste de uma Cifra de Feistel (ou rede de substituição e permutação) c onde

- ou, no esquema de *Mayer-Davis*, m_i é a chave e H_{i-1} o texto claro; depois o texto cifrado é adicionado (por XOR) a H_{i-1}. Isto é

$$h_i = h_{i-1} \oplus c(h_{i-1}, m_i)$$

- ou no esquema de *Miyaguchi-Preneel*, m_i é o texto claro e (uma alteração $g(\mathrm{H}_{i-1})$ de) H_{i-1} é a chave; depois, como no esquema de Mayer-Davis, o texto cifrado é adicionado (por XOR) a H_{i-1}. Isto é

$$h_i = h_{i-1} \oplus c(m_i, g(h_{i-1}))$$

A adição de h_{i-1} garante que a função de compressão C não seja mais invertível (ao contrário da cifra c para uma chave fixa); isto é, para dada saída, não é mais possível saber a entrada (única). Caso contrário, para criar uma colisão, dada uma saída h_i, poderia decifrar por chaves diferentes, m'_i e m''_i para obter entradas diferentes h'_{i-1} e h''_{i-1}.

Acolchoamento. Para reduzir a imunidade da função hash à da de compressão, o acolchoamento $m \mapsto \overline{m}$ precisa satisfazer as condições suficientes:

- m é um segmento inicial de $\overline{m}$, isto é, a mensagem é prolongada, mas o início não alterado!
- duas mensagens do mesmo comprimento são prolongadas pelo mesmo segmento final.
- duas mensagens de comprimentos diferentes são prolongadas diferentemente tal que difiram no último bloco.

O acolchoamento mais simples que cumpre estas condições é a que anexa o comprimento $|m|$ a m e enche o segmento entre o fim de m e $|m|$ pelo número de 0s prescrito pelo tamanho do bloco, isto é, a concatenação

$$\overline{m} = m \,||\, 0^k \,||\, |m|.$$

Para evitar colisões, não basta o acolchoamento encher o resto da mensagem com zeros: Desta maneira, duas mensagens que só diferem no número de zeros finais no último final teriam o mesmo acolchoamento!

Em vez disto, a maneira mais simples seria anexar um dígito 1, e em seguida o resto com 0s. Porém, veremos que isto permitiria colisões com o meta-método de Merkle se o valor inicial IV foi escolhido da seguinte maneira:

Denotem $\mathrm{B}_1, \ldots, \mathrm{B}_k$ os blocos da mensagem e IV o valor inicial. O hash é calculado pela iterada aplicação da função de compressão $\mathrm{C}\colon (\mathrm{X}, \mathrm{B}) \to \mathrm{C}(\mathrm{X}, \mathrm{B})$

$$\mathrm{H}(\mathrm{M}) = \mathrm{C}(\mathrm{C}(..\mathrm{C}(\mathrm{C}(\mathrm{IV}, \mathrm{B}_1), \mathrm{B}_2)..\mathrm{B}_{k-1}), \mathrm{B}_k)$$

A graça do meta-método de Merkle é a redução de colisões da função hash à função de compressão: uma colisão do hash implicaria uma colisão da função de compressão, isto é, diferentes pares de blocos X', B' e X'', B'' com $C(X', B') = C(X'', B'')$.

Para ver isto, observamos que

- se as mensagens m' e m'' colidentes têm comprimentos diferentes, então os últimos blocos B'_k e B''_k são diferentes (porque contêm os comprimentos diferentes!) mas colidem, porque o valor da função de compressão da última entrada é o da função de hash.
- caso contrário, isto é, as mensagens colidentes têm o mesmo comprimento, então existe um bloco o mais à direita onde as mensagens acolchoadas diferem e conseguiremos encontrar uma colisão de blocos depois dele.

Sem o comprimento no acolchoamento, a colisão de duas mensagens com comprimentos diferentes só pode ultimamente ser reduzida a uma pré-imagem do valor inicial IV sob a função de compressão, isto é, um valor B tal que

$$\mathrm{IV} = C(\mathrm{IV}, B)$$

Se a sua escolha foi arbitrária, os autores de `MD5` e `SHA-256` poderiam ter inserida a seguinte porta traseira: Ambos os algoritmos usam o esquema de Mayer-Davis, isto é, uma função de compressão

$$C(X, K) = X \oplus E(X, K)$$

para uma cifra de Feistel E com chave K; em particular, com K fixo, a função de decifração $E_K = E(\cdot, K)$ é invertível! Ora, se quisessem, os autores poderiam ter escolhido uma chave K e ter definido

$$\mathrm{IV} := E_K^{-1}(0)$$

exatamente tal que $\mathrm{IV} = C(\mathrm{IV}, K)$. Logo $C(\mathrm{IV}, B) = C(C(\mathrm{IV}, K), B)$, isto é, uma colisão entre os hashes de B e $K \oplus B$!

Já que, por exemplo, `MD5` e `SHA-256` escolhem como IV um valor cuja pré-imagem supostamente não é conhecida (por exemplo, os dígitos hexadecimais em ordem crescente no `MD5` ou os dígitos decimais dos primeiros oito primos no `SHA-256`) este problema é sobretudo teórico antes de prático.

SHA-256. Pseudo-Código do SHA-1

```
Note 1: All variables are unsigned 32-bit quantities, and
    wrap modulo 232 when calculating, except for
    - ml, the message length, which is a 64-bit quantity, and
    - hh, the message digest, which is a 160-bit quantity.
Note 2: All constants in this pseudo code are in big endian.
    Within each word, the most significant byte is stored leftmost.

Initialize variables:

h0 = 0x67452301
h1 = 0xEFCDAB89
h2 = 0x98BADCFE
h3 = 0x10325476
h4 = 0xC3D2E1F0

ml = message length in bits
    (always a multiple of the number of bits in a character).

Pre-processing:
append the bit '1' to the message, e.g.
    by adding 0x80 if message length is a multiple of 8 bits.
append 0 <= k < 512 bits '0', such that the resulting message length
    in bits is congruent to -64 \equiv 448 (mod 512)
append ml, the original message length, as a 64-bit big-endian integer.
Thus, the total length is a multiple of 512 bits.

Process the message in successive 512-bit chunks:
break message into 512-bit chunks
for each chunk
   break chunk into sixteen 32-bit big-endian words w[i], 0 <= i <= 15

    Extend the sixteen 32-bit words into eighty 32-bit words:
    for i from 16 to 79
      w[i] = (w[i-3] xor w[i-8] xor w[i-14] xor w[i-16]) leftrotate 1

    Initialize hash value for this chunk:
```

```
    a = h0
    b = h1
    c = h2
    d = h3
    e = h4

    Main loop:[3][55]
    for i from 0 to 79
        if 0 \leq i \leq 19 then
            f = (b and c) or ((not b) and d)
            k = 0x5A827999
        else if 20 \leq i \leq 39
            f = b xor c xor d
            k = 0x6ED9EBA1
        else if 40 \leq i \leq 59
            f = (b and c) or (b and d) or (c and d)
            k = 0x8F1BBCDC
        else if 60 \leq i \leq 79
            f = b xor c xor d
            k = 0xCA62C1D6

        temp = (a leftrotate 5) + f + e + k + w[i]
        e = d
        d = c
        c = b leftrotate 30
        b = a
        a = temp

    Add this chunk's hash to result so far:
    h0 = h0 + a
    h1 = h1 + b
    h2 = h2 + c
    h3 = h3 + d
    h4 = h4 + e

Produce the final hash value (big-endian) as a 160-bit number:
hh =
    (h0 leftshift 128) or (h1 leftshift 96) or
```

```
(h2 leftshift 64)  or (h3 leftshift 32) or h4
```

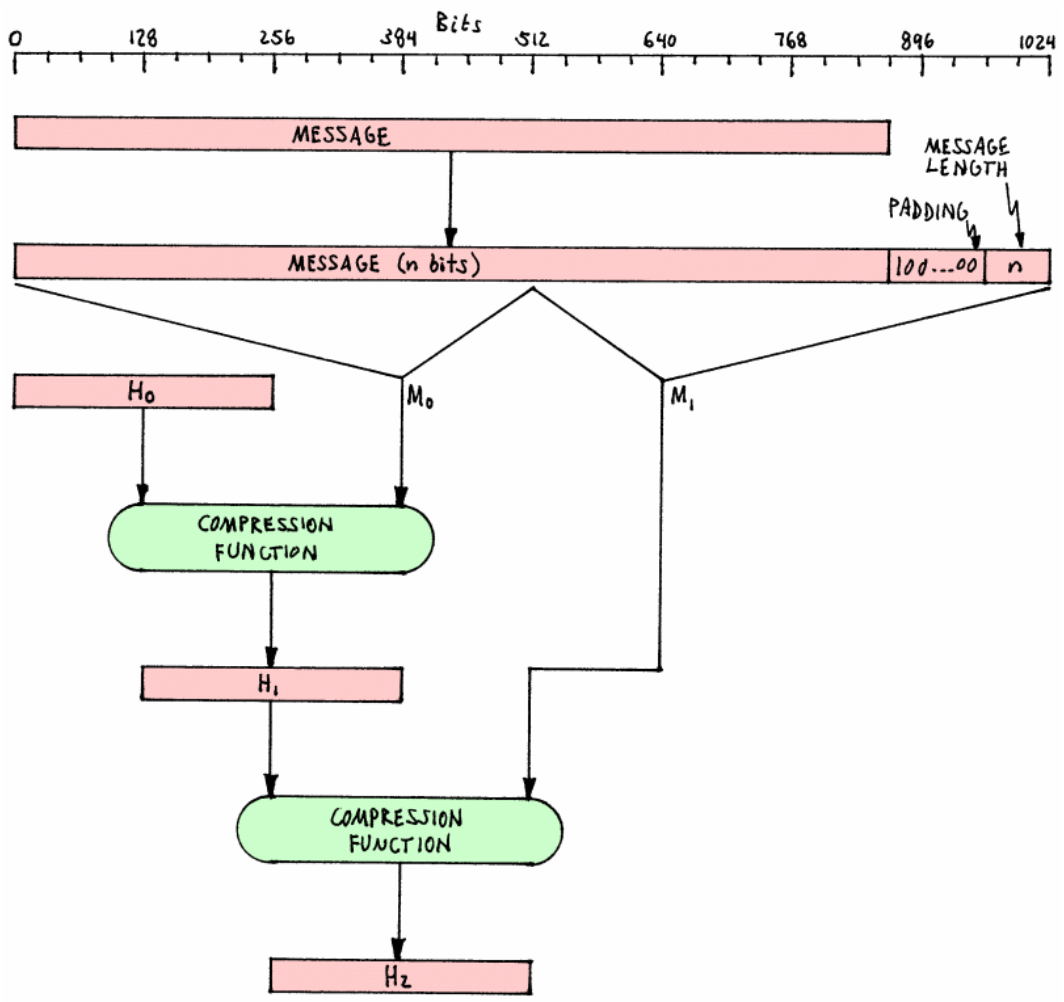


Figura 86: Compressão

onde as funções nos diagramas são definidas pelas operações básicas:

$\sigma_0(X) = (Xright - rotate7) xor (Xright - rotate18) xor (Xright - shift3)$

$\sigma_1(X) = (Xright - rotate17) xor (Xright - rotate19) xor (Xright - shift10)$

$\Sigma_0(X) = (Xright - rotate2) xor (Xright - rotate13) xor (Xright - rotate22)$

$\Sigma_1(X) = (Xright - rotate6) xor (Xright - rotate11) xor (Xright - rotate25)$

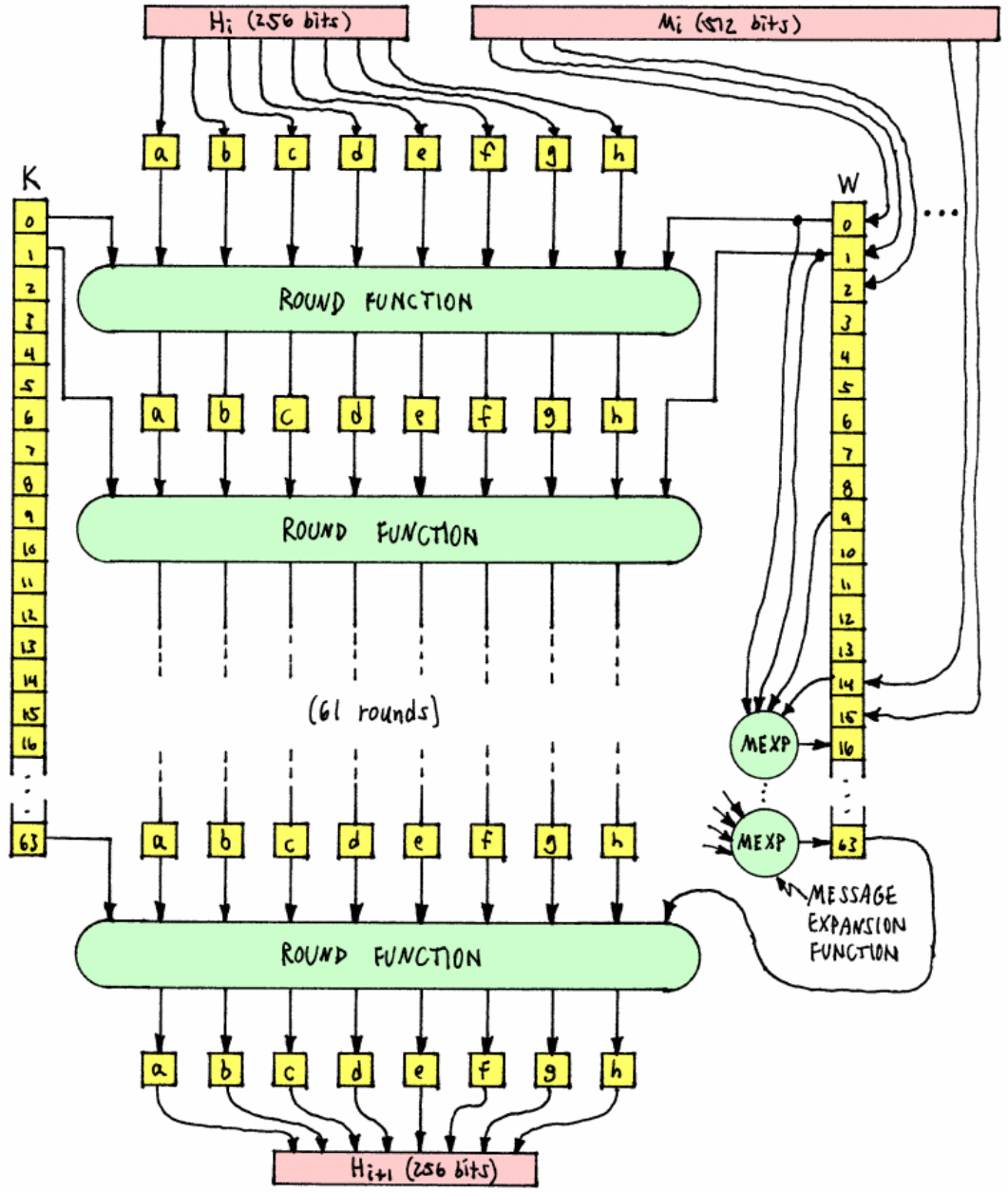


Figura 87: Visão Geral

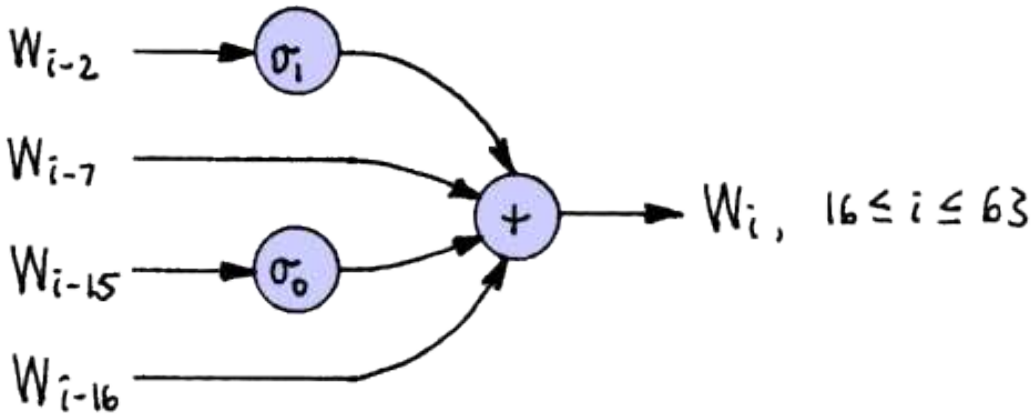


Figura 88: Expansão

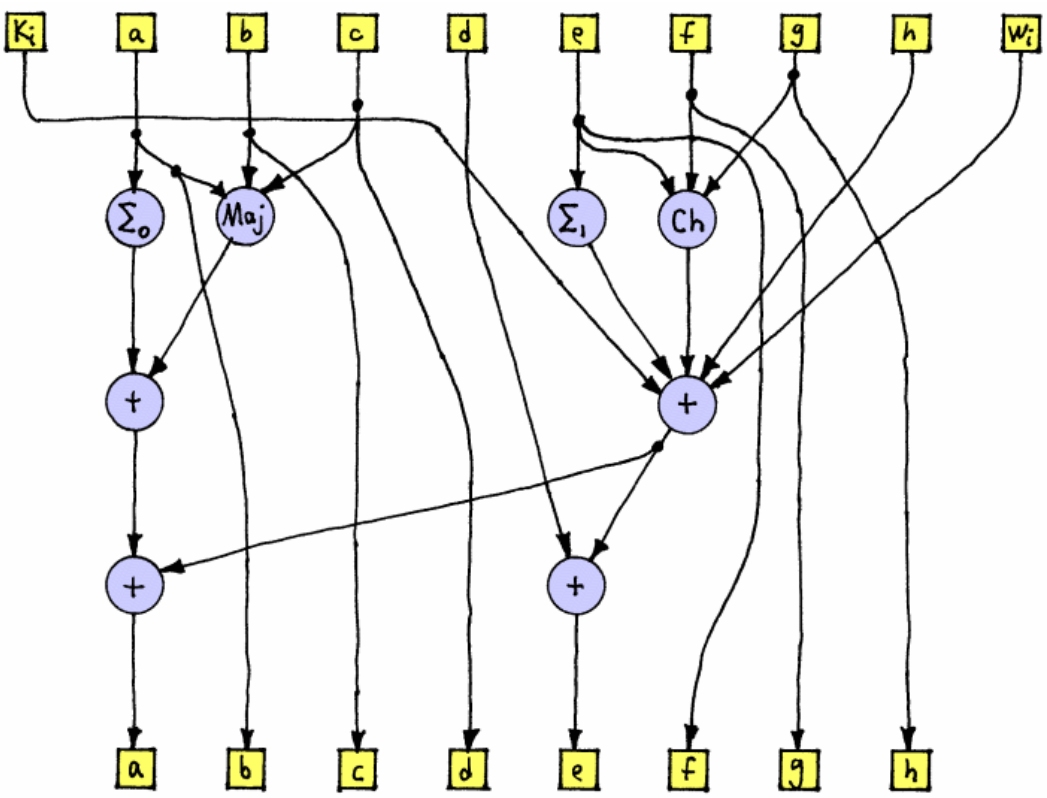


Figura 89: Rodada

Ch(X,Y,Z) = (XandY)xor((notX)andZ)

Max(X,Y,Z) = (XandY)xor(XandZ)xor(YandZ)

Em http://www.metamorphosite.com/one-way-hash-encryption-sha1-data-software tem uma explicação detalhada e um simulador do SHA-1.

10.4 Usos

Funções de Hash (*não* necessariamente criptográficas, como CRC) são usadas:

- para consulta de dados rápidas (isto é, em tempo fixo, independente do número de entradas)
  - em uma tabela hash, e
  - em uma árvore de Merkle; e
- para garantir a *integridade* de um arquivo diante modificações *acidentais*, isto é, para detetar diferenças entre o arquivo e uma versão de referência (tipicamente a antes do transporte do arquivo).

Funções de Hash *criptográficas* são usadas:

- Para distribuir valores uniformemente (key stretching), intuitivamente, torná-los menos previsíveis; isto é, como KDF (= Key Derivation Function);
  - em particular, para *gerar e armazenar senhas*, isto é, como PBKDF (= Password Based Key Derivation Function).
- Para garantir a *integridade* de um arquivo diante modificações *voluntárias* (e acidentais), isto é, para detetar diferenças entre o arquivo e uma versão de referência (tipicamente a antes do transporte do arquivo);
  - em particular, para garantir a *autenticidade* de um arquivo: detetar diferenças entre o arquivo e uma versão que estava sob controle de uma pessoa específica.

Observação: Atenção à diferença entre *autenticidade* e *autentication:* A primeira garante a igualdade dos dados recebidos e enviados de uma pessoa (por exemplo, na assinatura digital), a última a identidade desta pessoa (por exemplo, no acesso de um site seguro).

Os algoritmos de hash criptográficos alistados acima, `MD4/5`, `SHA` ... distribuem os valores uniformemente, mas são *rápidos*; logo, não são apropriados à criação de senhas, porque são vulneráveis a ataques de força bruta. Para impedi-los, é necessário que o algoritmo do `PBKDF`, por exemplo, `PBKDF1`, `PBKDF2`, `bcrypt`, `scrypt`, `Argon2` (um candidato novo e o mais prometedor), seja

- deliberadamente *lento*, isto é, requer muito tempo para ser computado, em particular, em hardware especializado como,
  - para o consumidor comum, um `GPU`, um microprocessador gráfico, e
  - para alguém com mais recursos, um `ASIC`, um microprocessador, ou `C`ircuito `I`ntegrado, `A`daptado para uma aplicação e`S`pecífica, aqui a computação de um `hash`;

  a este fim, recomenda-se atualmente `bcrypt`; e
- deliberadamente *guloso*, isto é, requer muita memória para ser computado (o objetivo do novo algoritmo `scrypt`);
- usada uma única vez para cada entrada (garantido por um `salt`, uma pitada de sal, um argumento adicional único, comumente aleatório [e gerado por outra função de hash], da função de hash): sem `salt`, o algoritmo é propenso a ataques por `Rainbow Tables`, a comparação com os hashes das entradas mais prováveis (guardados em tabelas, donde o nome do ataque), por exemplo, das senhas mais usuais.

10.5 Tabela de Dispersão

Uma *tabela hash* (ou *tabela de dispersão*) usa uma função hash para endereçar as entradas (= linhas) de um *array*, isto é, uma tabela de dados.

Cada entrada tem um nome, isto é, uma identificação única (= *chave*). Por exemplo, a chave é o nome de uma pessoa. Os dados da chave, por exemplo, o seu número de telefone, são guardados em uma linha da tabela. Estas linhas são numeradas a partir de 0.

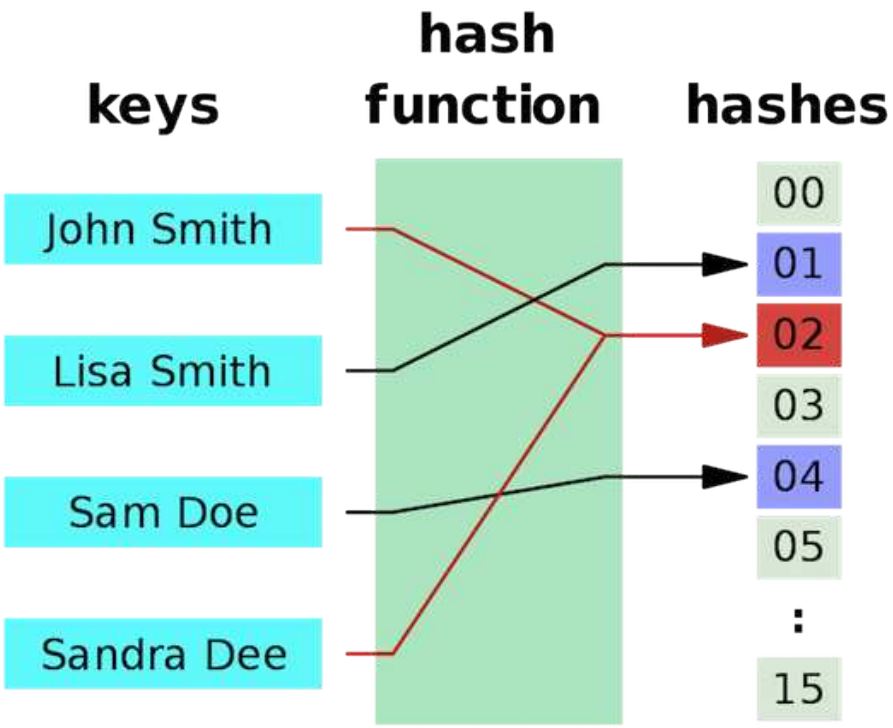


Figura 90: Tabela de Dispersão

O número da linha, o seu *endereço*, da chave é determinado pelo seu hash. Como vantagem, em tempo fixo, os dados podem

- a partir da chave ser encontrados, e
- adicionados.

Enquanto,

- para uma lista de n entradas, a busca compara em média $n/2$ entradas, e
- para uma árvore binária de n entradas, $\log_2 n$ entradas.

Colisões, isto é, duas entradas com o mesmo hash são mais frequentes do que se imagina; vide o *Paradoxo do Aniversário*. Para evitar colisões, é necessário

- escolher o número de endereços suficientemente grande, e

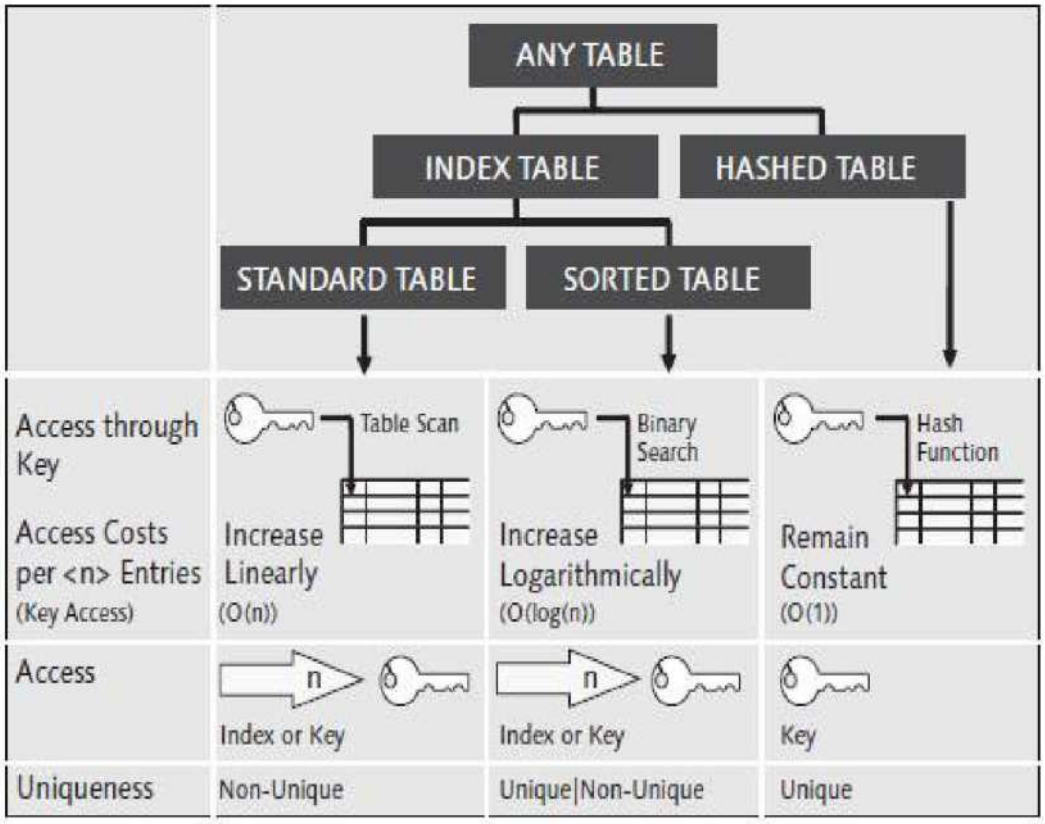


Figura 91: Comparação das estruturas de dados

- a função de hash suficientemente *injetora*, isto é, raramente envia argumentos diferentes a valores iguais.

Quando ocorrem colisões, umas estratégia são

- em vez do número de hash endereçar uma única entrada, usar `Chaining`: endereçar um *bucket*, um *balde* de várias entradas, uma lista, ou
- usar `Open Hashing`: colocar a entrada em outra posição livre, por exemplo,
  - a próxima, ou
  - usar outra função hash para calculá-la (`Double Hashing`).

Até 80% da tabela sendo repletos, ocorrem em média ≤ 3 colisões. Isto é, encontrar uma entrada leva 3 operações. Para comparar, com 1000 entradas, levaria em média

- em uma tabela 500 operações, e
- em uma árvore binária cerca de 10 operações.

Porém, depois deste fator as colisões ocorrem com uma tal frequência que o uso de outra estrutura de dados, por exemplo, uma árvore binária, é recomendável.

10.6 Árvore de Merkle

Uma **árvore de dispersão** ou **de Merkle** (inventada em 1979 por `Ralph Merkle`) agrupa dados em blocos por uma árvore (binária), cujos *vértices* (= nós) são hashes e cujas *folhas* (= nós terminais) contêm os blocos de dados, para poder verificar rapidamente (em tempo *linear*) cada bloco de dados pela computação do hash da raiz.

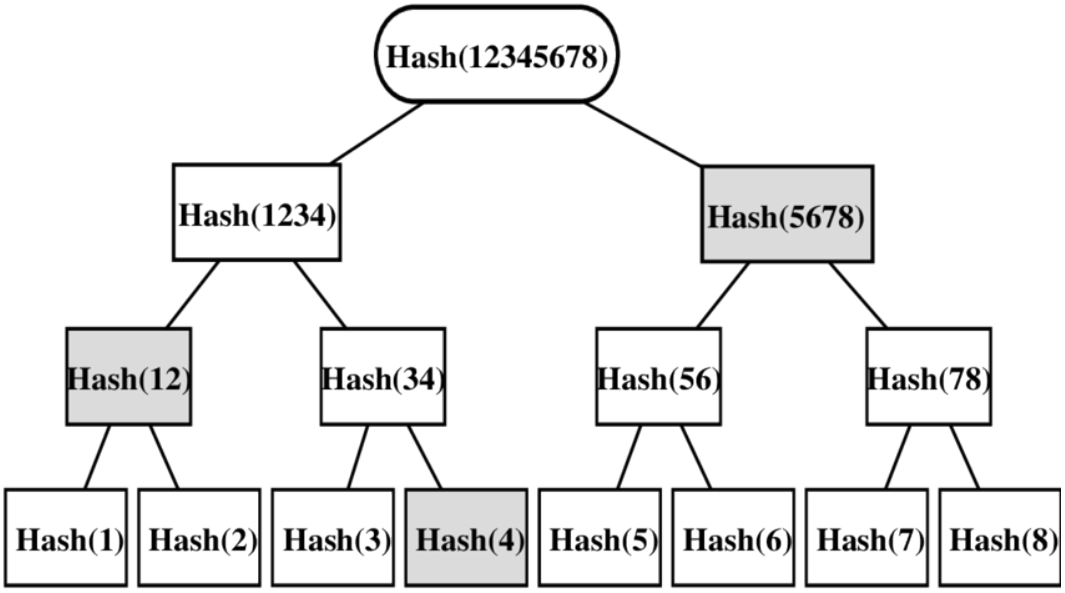


Figura 92: Verificação do terceiro bloco duma árvore de Merkle pela computação do hash radical. O hash de cada nó (mãe) é o hash (da concatenação) das duas filhas. Os hashes que necessários e que foram informados para calcular o do topo são cinzentos.

Em uma *árvore* de dispersão de profundeza n (= 2^n folhas)o bloco de dados de cada folha é verificável

- pelo **conhecimento**
  - de n hashes "antagonistas" de uma fonte **dubitável**, e
  - do hash do topo de uma fonte **confiável**, e
- pelo **cálculo**
  - do hash do bloco de dados da folha,
  - dos n hashes dos seus antecessores, e
  - a comparação do hash da raiz calculado com o obtido.

Uso. O principal uso de árvores de Merkle é garantir que blocos de dados recebidos de outros pares em uma rede ponto-a-ponto (P2P) sejam recebidos intactos e inalterados.

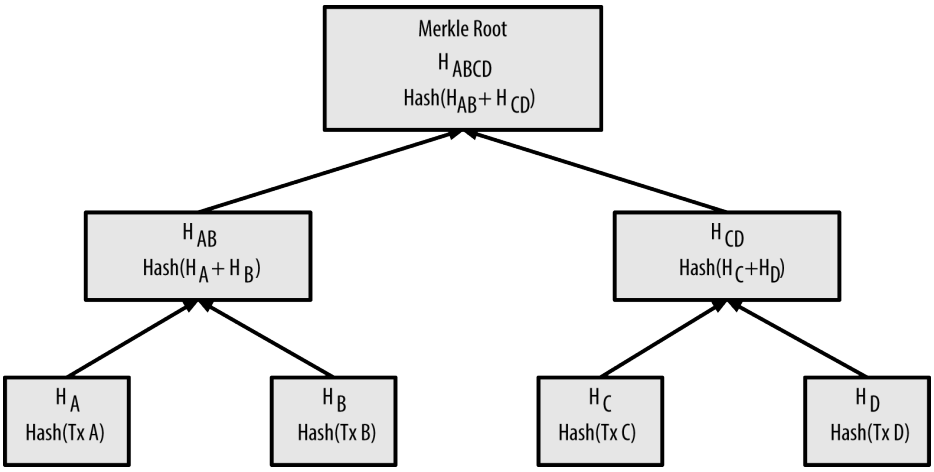


Figura 93: Árvore de Transações

Funcionamento. Uma árvore de Merkle é uma árvore (usualmente binária) de hashes cujas folhas são blocos de dados de um arquivo (ou conjunto de arquivos). Cada nó acima das folhas na árvore é o hash dos seus dois filhos. Por exemplo, na imagem,

- Hash(34) é o hash da concatenação dos hashes Hash(3) e Hash(4), isto é, Hash(34) = Hash(Hash(3)||Hash(4)),
- Hash(1234) é o hash da concatenação dos hashes Hash(12) e Hash(34), isto é, Hash(1234) = Hash(Hash(12)||Hash(34)), e

- o hash radical `Hash(12345678)` é o hash da concatenação dos hashes `Hash(1234)` e `Hash(5678)`, isto é,

$$\text{Hash}(12345678) = \text{Hash}(\text{Hash}(1234)||\text{Hash}(5678)).$$

.

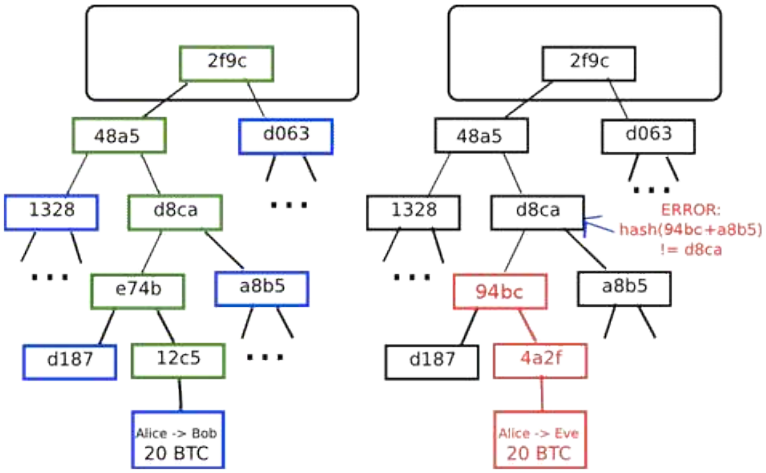


Figura 94: Árvore de Merkle com defeito

Normalmente uma `função de embaralhamento *criptográfico*` é usada, por exemplo, `SHA-1`. Porém, se a árvore de Merkle só precisa proteger contra danos involuntários, `somas de verificação` não necessariamente criptográficas, como `CRCs` são usadas.

No topo da árvore de Merkle reside a *dispersão de raiz* ou *dispersão mestre*. Por exemplo, em uma rede `P2P`, a dispersão de raiz é recebida de uma fonte confiável, por exemplo, de um site reconhecido. A árvore de Merkle em si é recebida de qualquer ponto na rede `P2P` (não particularmente confiável). Esta árvore de Merkle (não particularmente confiável) recebida é comparada pelo cálculo dos hashes das folhas com a dispersão de raiz confiável para verificar a integridade da árvore de Merkle.

A principal vantagem de uma árvore (de profundeza n com 2^n folhas, isto é, blocos de dados), em vez de uma lista, de `dispersão` é que a integridade de cada folha pode ser verificada pelo cálculo de n (em vez de 2^n) hashes (e a sua comparação com o hash da raiz).

Por exemplo, em Figura 92, a verificação da integridade da Folha 3 requer apenas

- o conhecimento dos hashes `Hash(4)` `Hash(12)`, e `Hash(5678)`, e
- a computação dos hashes `Hash(3)` `Hash(34)`, `Hash(1234)`, e `Hash(12345678)`
- a comparação entre o `Hash(12345678)` calculado e o hash obtido da fonte confiável.

11 Criptomoedas

Ao contrário de uma moeda comum, em que os negociantes confiam em um terceiro, o banco, que mantém um livro-razão de todas as transações, em uma *criptomoeda*, eles confiam em uma *blockchain*, literalmente *cadeia de blocos*, um livro-razão público de todas as transações; mantido (e replicado) por uma rede de milhares de computadores e praticamente infalsificável.

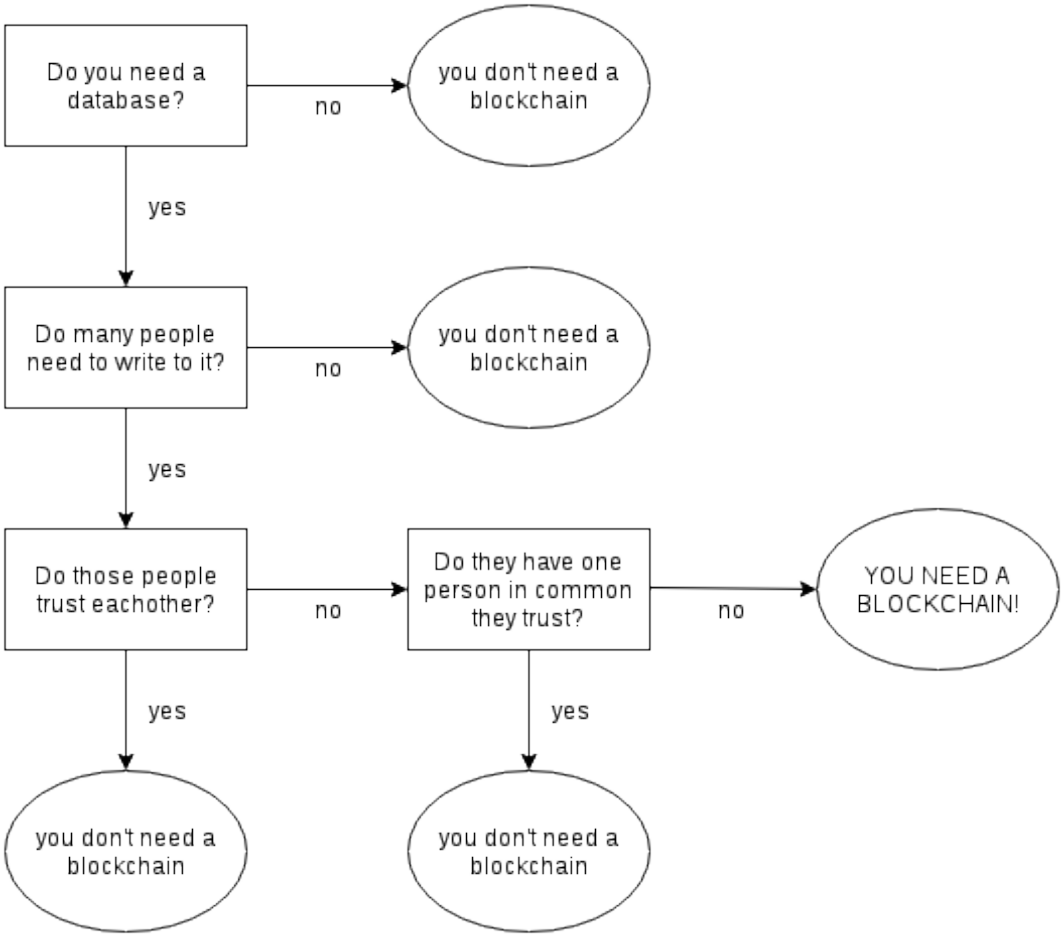


Figura 95: Necessidade de uma cadeia de blocos. Em breve: Uma cadeia de blocos é útil se é preciso um banco de dados para múltiplos usuários que confiam nem neles nem num terceiro.

Para poder confiar na cadeia de blocos (na ausência de um terceiro), isto é, para garantir a imutabilidade dos blocos antigos (= os que já figuram na cadeia

em contraste aos que serão acrescentados), a cadeia tem de ser pública, isto é, toda a rede pode ler e acrescentar blocos. Com efeito, é perfeitamente viável alterar (blocos antigos d') a cadeia; porém, esta alteração será detetada pela rede e em seguida a cadeia alterada rejeitada.

Os meios criptográficos para implementar uma criptomoeda são

- a *criptografia assimétrica* para assinar as transações, e
- as *funções de hash criptográficas* para
  - endereçar as transações, e
  - dificultar a extensão da blockchain (em particular, a sua modificação maléfica).

Recordemo-nos de que uma *função hash* é uma função que envia (*praticamente* sempre) sequências de bites diferentes (de um comprimento *arbitrariamente grande*) a sequências de bites diferentes (de um comprimento *fixo*). Os seus valores, os *hashes*, servem como informações curtas que endereçam, ou indicam, (praticamente sem equívoco) dados grandes, por exemplo,

- para os blocos da cadeia, e
- para as transferências (em uma árvore de Merkle) que constituem um bloco.

A função de hash é *criptográfica*, se o processo é praticamente unidirecional, isto é, se o Hash não permite fazer conclusões sobre o conteúdo original. O Bitcoin usa esta dificuldade da computação da entrada de uma saída de uma função de hash "como prova de trabalho" que é necessário para estender a blockchain, o livro-razão, impedindo a sua alteração (maléfica).

Existem outras criptomoedas, cada uma com as suas próprias características. Destacam-se, entre outras, por exemplo,

- o `BurstCoin` que usa uma "prova de espaço" em vez de uma prova de trabalho, isto é, a reserva de um espaço no disco rígido, para ganhar moedas.

Tem também a inconveniência que todas as transações do `Bitcoin` são públicas (com como única ofuscação o conhecimento da pessoa por trás da chave pública [por trás do endereço `Bitcoin`]). Para aumentar a privacidade, existem, entre outras moedas

- o Monero que usa *assinaturas de grupo* (vide Seção 4.1) para confirmar as transações entre os negociantes, assim ofuscando o fluxo das moedas, e
- o Zcash que usa zk-SNARKs em vez de assinaturas de grupo, uma *prova de não-interativa conhecimento zero.*

Veremos

1. como a blockchain funciona, a cadeia de blocos,
2. de que consiste um bloco (de transações),
3. como os bitcoins são gerados, a mineração, e porque este trabalho árduo, esta *prova de trabalho* é indispensável para garantir a integridade da cadeia, e
4. como as transações se fazem.

11.1 Sobrevoo

Sobrevoemos a arquitetura da Blockchain (do Bitcoin):

Blocos. A blockchain, literalmente, é uma cadeia de blocos. Há um bloco inicial, o bloco Genesis e blocos que apontam aos seus antecessores.

Esta flecha que aponta ao antecessor é um *hash*, uma identificação do conteúdo inteiro do bloco antecedente; é um *endereço* que figura no cabeçalho do bloco.

O tronco do bloco consiste de transações, em média 2000 delas, entre os usuários do bitcoin.

Laços. Como o *hash* do bloco depende do conteúdo inteiro do bloco, a singela alteração de um bit muda o seu hash, isto é, invalida a flecha do sucessor para ele! Logo, para os blocos continuarem a formar uma cadeia, é preciso alterar esta flecha. Logo, o hash do sucessor muda! Logo, para os blocos continuarem a formar uma cadeia, é preciso alterar a flecha do sucessor do sucessor, e assim por diante, uma reação em cadeia!

Quer dizer, se alterarmos algum detalhe, por exemplo, uma transação no primeiro bloco, todas as flechas (= hashes) que seguem têm de ser recalculadas!

Mineração. Entra a questão da mineração que dificulta muito esta alteração, porque somente blocos cujos hashes são pequenos, quer dizer começam com muitos zeros, são aceitos pela rede (isto é, pelos nós que verificam). Então, não basta alterar uma transação e calcular os novos hashes de todos os sucessores. É preciso fazer esta alteração tal que todos os hashes sejam pequenos!

É muito difícil encontrar uma tal alteração. Atualmente, a busca de um tal bloco requer um bilhão de anos num computador usual; mas só dez minutos pela rede dos mineradores. Enquanto o malfazejo começou a buscar os blocos aceitáveis, a rede já criou vários outros, invalidando este trabalho!

Transações. Todos os bitcoins que existem foram gerados por mineração, isto é, foram dados (pela coinbase transação) como recompensa àquele que criou um bloco (com um hash pequeno).

Todas as outras transações têm uma entrada e saída. Na entrada junta-se uma quantia suficiente para pagar o que será gastado. Tem de gastar tudo! Por isso, usualmente uma transação inclui o remetente como destinatário, o troco. O que não for gastado ganhará o minerador que incluirá esta transação no seu bloco como recompensa (atualmente em volta de 40 $).

O destinatário é designado pela sua chave pública, e só na hora em que gastará as moedas precisa de demonstrar a posse da sua chave privada correspondente, pela sua assinatura da transação.

Curvas Elípticas. O `Bitcoin` usa a criptografia com curvas elípticas finitas para assinar as transações. O Diffie-Hellman usa anéis finitos como $\{0,1,\ldots,m\}$ (por exemplo, $m = 12$ para o relógio, e um número primo como $p = 2^{255} - 19$ com 100 algarismos no `Bitcoin`). O conceito usado pelo `Bitcoin` semelha à de Diffie-Hellman, só que em vez números $0,1,2,\ldots$ usa *pares* de números (x,y), isto é, pontos de uma curva, de um gráfico, dito *elíptica*.

A graça desta curva é que podemos adicionar pontos: vale $p + q + r = 0$ se os três pontos estão na mesma reta. Em vez de multiplicar várias vezes o mesmo número, adicionamos o mesmo ponto. É fácil adicionar pontos com esta receita sobre estes anéis finitos, mas é dificílimo saber quantas vezes um ponto foi adicionado a si mesmo para obter o ponto resultante. A cifração corresponde à adição, a decifração ao conhecimento deste número de vezes.

Finalmente, o esquema de assinatura é uma variação da assinatura de `ElGamal`: a assinatura demonstra que o dono da chave privada conseguiu resolver uma equação difícil, tão dificílimo que é praticamente impossível resolvê-la sem esta chave privada que fornece um atalho.

11.2 Blockchain

A *blockchain* é uma cadeia de blocos que são ligados, isto é, cada bloco tem um hash de outro bloco, de que pensamos como um indicador (ou endereço) para o bloco *anterior*.

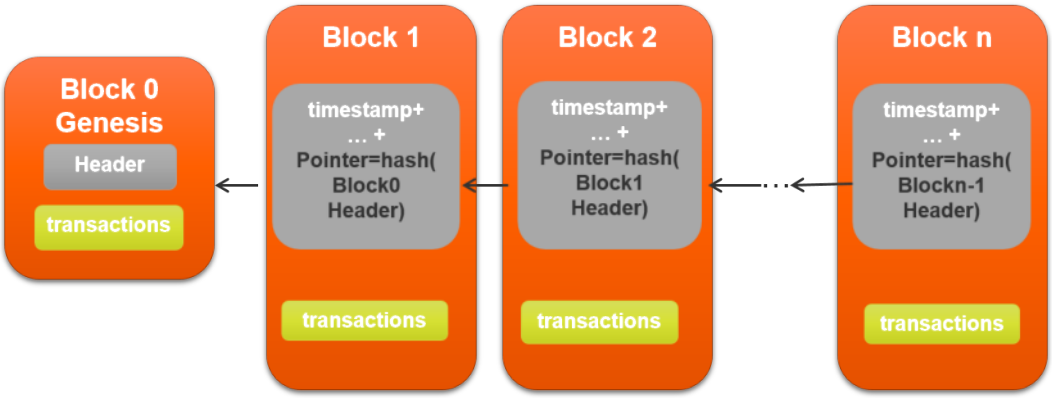


Figura 96: Blockchain

Ocorrem ramificações, blocos *orfanados*; porém, só a cadeia mais comprida (mais exatamente, esta cuja construção era computacionalmente a mais trabalhosa, isto é, a soma das dificuldades dos trabalhos dos blocos; vide prova de trabalho em Seção 11.4) é considerada válida. Como é dificílimo estender a cadeia por outro bloco, raramente as ramificações têm mais de um bloco.

A cadeia começou no dia 3 em janeiro de 2009 às 18:15 UTC, provavelmente por Satoshi Nakamoto, com o primeiro bloco, o *bloco de Génesis*.

Cada bloco consiste

- de um *cabeçalho*, os meta-dados, que contém
  - o indicador para o bloco anterior, e
  - informações sobre a sua criação.

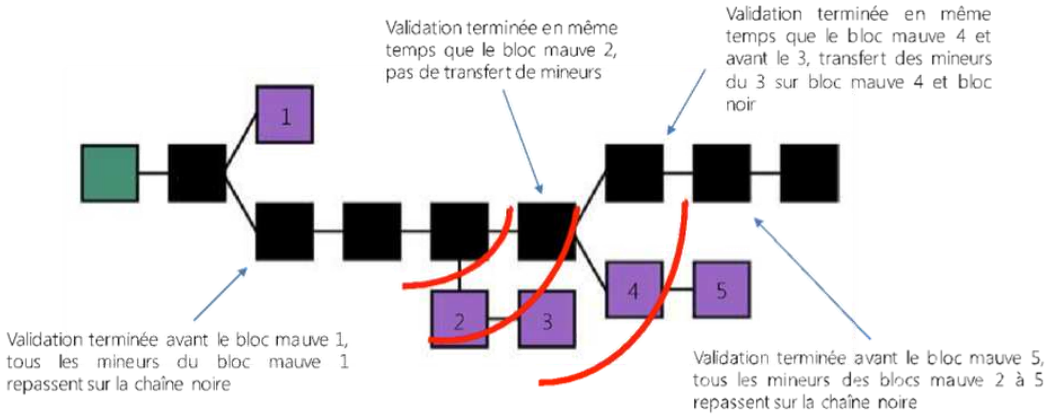


Figura 97: Gerenciamento de Bifurcações

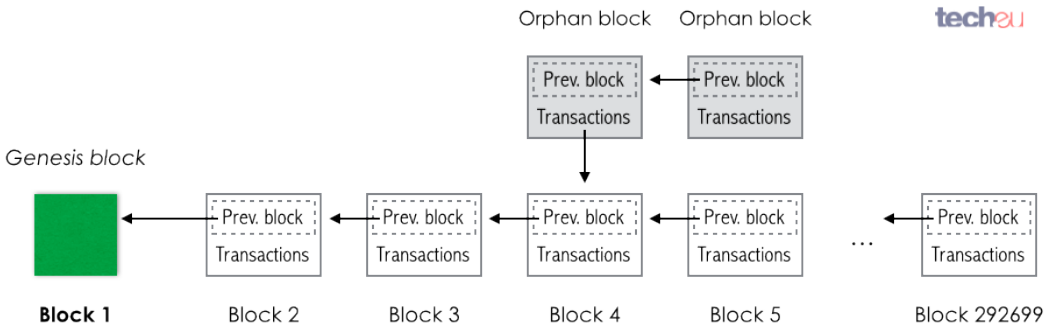


Figura 98: Bloco de Génesis

- do seu *conteúdo*, os dados, muitas (cerca de um megabaite de) *transações* (agrupadas em uma *árvore de Merkle*) entre os usuários do Bitcoin.

O hash do bloco depende do seu conteúdo todo. Isto é, qualquer alteração, implica a alteração do seu hash, em particular,

- a alteração de uma de suas transações,
- a alteração do seu indicador ao bloco anterior.

Por exemplo:

1. se uma das suas transações muda, então o seu hash.
2. Logo, o indicador do bloco posterior muda, logo o hash do bloco posterior.

3. Igual para o hash do bloco posterior ao bloco posterior, e
4. ... assim por diante.

Em outras palavras, ocorre uma reação em cadeia: A alteração de uma singela transação em um bloco invalida todos os hashes dos seus blocos posteriores.

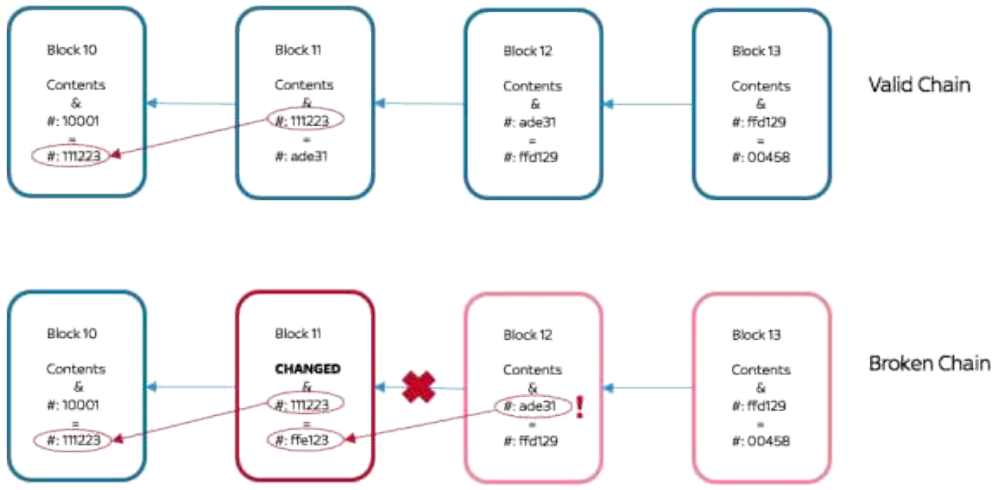


Figura 99: Bloco inválido

Logo, para os blocos continuarem a formar uma cadeia, é preciso recalcular todos os indicadores (, isto é, o hash do bloco posterior) de todos os blocos que seguem o bloco alterado. Normalmente, estes novos blocos invalidam a blockchain porque estes novos hashes não conformam mais ao padrão requerido (= um número suficiente de dígitos iniciais iguais a 0) para ser aceitos na blockchain.

11.3 Bloco

Cada *bloco* agrupa *transações* entre os usuários em uma *árvore de Merkle.*

Estrutura.

Campo	Descrição	Tamanho
Magic	= 0xD9B4BEF9	4 baites
Cabeçalho	Contém 6 items	80 baites
Tamanho do Bloco		4 baites
Número de transações		1 – 9 baites
Transações		

Cada bloco agrupa transações (que têm, em média, 5000 baites). No início, é indicado o seu tamanho (até um 1 Megabaite) e o seu número das transações (em média 2000).

A primeira transação, a `coinbase` transação, a recompensa pelo trabalho feito pela sua criação, é alterável pelo criador do bloco e, logo, comumente ele e os seus colaboradores são os destinatários.

As outras transações são transmitidas à rede pelos remetentes, isto é, pagadores, e todos os criadores de blocos, os *mineradores*, decidem quais transações incluírem no bloco que estão criando. Para incentivar a inclusão de uma transação, o remetente pode pagar uma taxa ao criador do bloco; por isso, frequentemente o minerador inclui tantas transações quanto possíveis, cerca de 1 Megabaite.

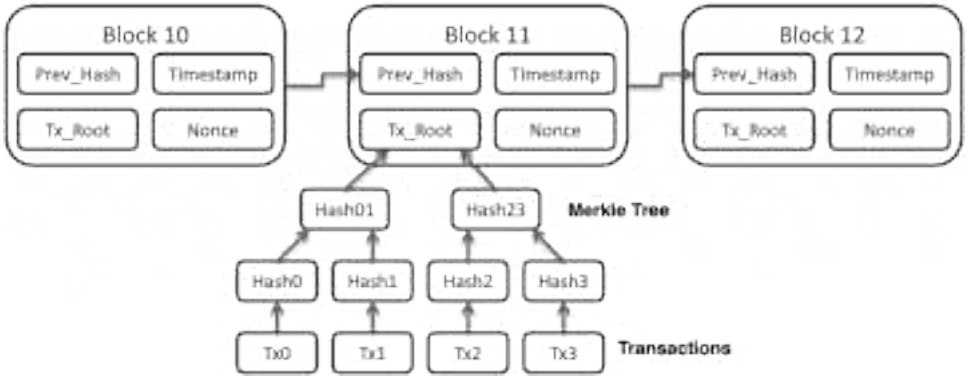


Figura 100: Para poder verificar rapidamente as transações, elas são agrupadas em uma árvore de Merkle, uma árvore (binária), cujos vértices são hashes e cujas folhas são transações.

Cabeçalho. O cabeçalho de cada bloco contém:

1. a versão do software usado,
2. o hash do bloco anterior (para os blocos formarem uma cadeia),
3. o hash da raiz da árvore de merkle das transações,
4. o timestamp, a data de criação em segundos contados desde 1970-01-01 às 00:00 UTC, e
5. a dificuldade, grosso modo o número de zeros com que o hash do bloco precisa de começar para ele poder estender a cadeia, e
6. um `nonce`, um campo sem conteúdo (semântico); serve para alterar o hash do bloco sem alterar o seu conteúdo (semântico).

A função de hash usada pelo `Bitcoin` é `SHA-256`. Em mais detalhes:

Campo	Atualizado quando...	Tamanho
Versão	o software é atualizado	4 baites
Hash do Bloco anterior	um novo bloco é criado	32 baites
Hash radical da árvore	uma transação foi aceita	32 baites
Data da criação	uns segundos passaram	4 baites
Dificuldade	cada 2016-ésimo bloco	4 baites
Nonce	outro hash é provado	4 baites

O "corpo" do bloco, o seu conteúdo (em contraste aos metadados do cabeçalho) são as transações. Estas são agrupadas em uma árvore "de merkle", uma árvore (normalmente binária) de hashes onde o hash de um nó é calculado pelos dos sucessores; os dados, aqui as transações, constituem as suas folhas.

O `nonce` serve na busca de um hash para modificar o bloco sem alterar o seu conteúdo. Como mencionamos acima, faz tempo que, quase com certeza, os 4 baites do `nonce` não são suficientes para encontrar um hash suficientemente pequeno. Isto é, após 2^{32} incrementos, nenhum hash é suficientemente pequeno. Neste caso, o `ExtraNonce`, a mensagem da `coinbase` transação (que tem 100 baites) é iterada. Porém, neste caso o hash da raiz da árvore de Merkle precisa de ser recomputada.

`Coinbase`. A primeira transação, a `coinbase` transação, a recompensa pelo trabalho feito pela sua criação, é criada pelo criador do bloco e, logo, comumente ele e os seus colaboradores são os destinatários.

Cada transação tem

- um `input`, uma entrada, e
- um `output`, uma saída,

Todas as transações, exceto a inicial, o input junta outputs de transações (por referir aos seus hashes) cuja soma é maior (ou igual) que a soma dos outputs. Na transação inicial do bloco, o input é arbitrário e a quantia é a recompensa dada ao minerador, inicialmente 50 bitcoins, 12,5 em 2018.

O primeiro bloco da cadeia, minerado por Satoshi Nakamoto, o pseudónimo do criador anónimo do Bitcoin, contém o título da página um da Financial Times:

```
The Times 03/Jan/2009
Chancellor on brink of second bailout for banks
```

Porém, como mencionado acima, frequentemente o conteúdo não é legível para o homem; em vez disto, é um valor a fins técnicos, por exemplo, para alterar o bloco tal que o seu hash seja suficientemente pequeno.

11.4 Extensão da Blockchain

Para acrescentar um bloco à blockchain, é preciso dar uma *prova de trabalho*, o cálculo de (um *cabeçalho* de) um bloco tal que o seu hash seja pequeno, isto é, que a sua expansão binária começa com um grande número de dígitos zero (atualmente, em agosto de 2018, com 20 zeros na expansão hexadecimal ou 80 zeros na expansão binária). Talvez o software mais usado a este fim seja atualmente CGMiner; existem versões para todos os sistemas operacionais, umas adaptadas para processadores gráfico (`GPUs`) e outras para processadores especificamente programados para mineração (`ASICs`).

O hash de um bloco minerado no dia 31 de agosto de 2018:

```
00000000000000000000e17d5d11694f4602f68a8ab629093a7f61baf6fea77
```

Irreversibilidade. Enquanto o alto custo de trabalho é inútil para erguer uma cadeia de blocos, é essencial para a integridade da cadeia porque impossibilita praticamente a alteração (maléfica) de blocos anteriores; isto é, garante a *irreversibilidade* da blockchain.

Uma vez a transação é em um bloco da blockchain que foi estendido por, por exemplo, pelo menos cinco, outros blocos, é praticamente impossível substituir os blocos da blockchain por outros (porque enquanto estes blocos são arduamente buscados, a blockchain já foi estendida por outros blocos).

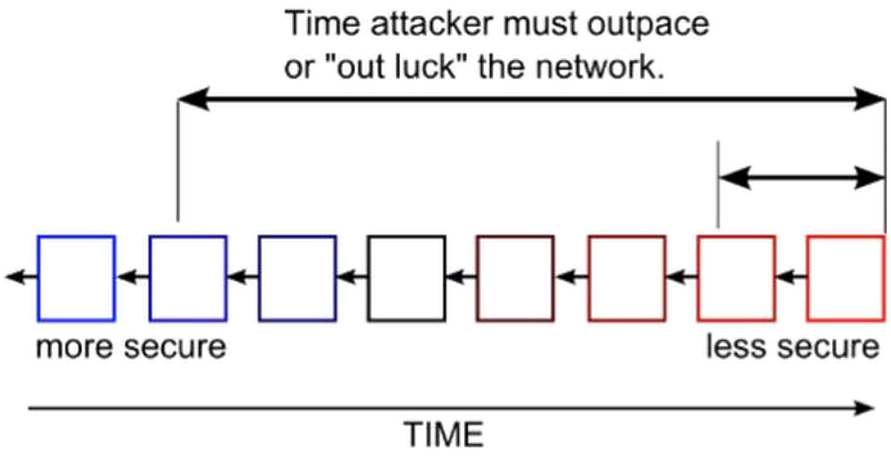


Figura 101: Sobrepujança

Como cada bloco contém em particular o hash do bloco anterior, e o hash depende dele, a modificação de um bloco necessita a modificação do bloco posterior; como cada bloco precisa de ter um hash que comece com muitos 0's para ser aceito na blockchain, é muito trabalhoso encontrá-los. Um imenso trabalho contra o tempo:

Para alterar um bloco na cadeia, o singelo minerador precisa de

- recalcular todos os blocos posteriores (tal que os seus hashes comecem com um número de 0's suficiente),
- enquanto todos os outros mineradores concatenam outros blocos!

Esta segurança permite dispensar de um terceiro entre os atores, isto é, de uma autoridade fiducial. Em vez dela, os nós da rede acordam sobre a validade da transação.

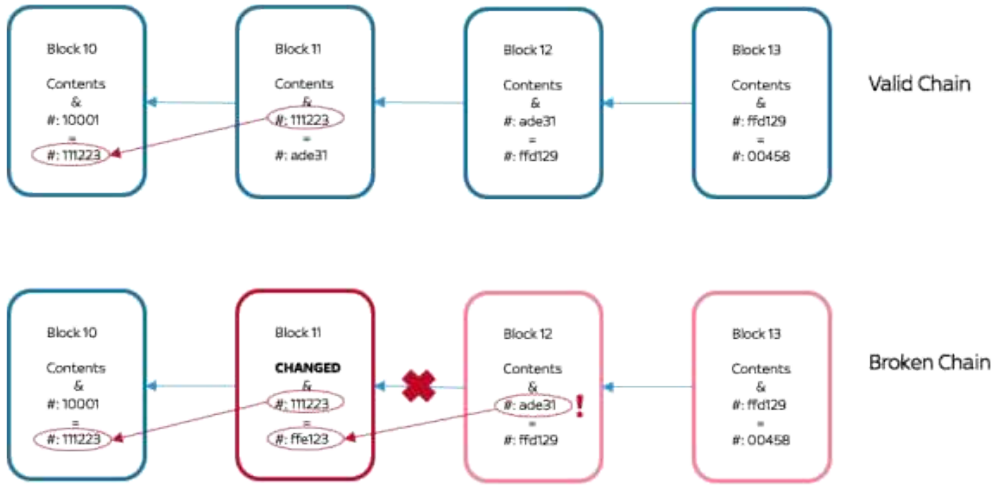


Figura 102: Bloco inválido

Porém, se um minerador tem mais força computacional que todos os outros mineradores, então pode aplicar a ataque de 51%:

1. Remete uma transação,
2. espera a sua confirmação, isto é, espera a sua inclusão em um bloco da blockchain, % e a sua prolongação por 6 outros blocos,
3. no momento em que é inclusa em um bloco, bifurca a blockchain pela prolongação do bloco precedente por outro bloco que não inclua esta transação,
4. continua a prolongar este ramo pela criação de outros blocos,
5. quando a transação é confirmada, isto é, depois 6 blocos, transmite à rede os blocos que criou. Pela sua suposta maior força computacional, este ramo é mais comprido do que a blockchain; logo, os nós aceitam o novo ramo como nova blockchain válida.

Prova de Trabalho. Uma *prova de trabalho* é a apresentação de uma informação cuja obtenção é computacionalmente custosa. Frequentemente, o trabalho consiste em realizar um evento pouco provável por muitas repetições, por força bruta.

`Bitcoin` usa a prova de trabalho introduzida por `Hashcash` para prevenir spam

por uma prova de trabalho para o envio de cada e-mail a cada destinatário; logo, torna o envio em massas custoso.

Em Bitcoin, a prova de trabalho é para a extensão da cadeia por cada novo bloco.

Hash. O trabalho consiste em buscar um bloco cujo hash com 32 baites pelo algoritmo SHA-256 seja menor que um certo número alvo:

- inicialmente, em 2009 um número que começa com 4 baites que são 0,
- atualmente, em 2018, um número que começa com 10 baites que são 0.
- O número de zeros é continuamente (cada 2016-ésimo bloco) ajustado tal que a busca leve em média (mundialmente!) 10 minutos.

Outros algoritmos de Hash (em vez de SHA-256) comuns para provas de trabalhos são, por exemplo,

- Scrypt, e
- SHA-3.

Exemplo. Concatenamos iterativamente à sequência de caracteres "Hello, world!" um número, um nonce, um número a uso único (= once em inglês), tal que (a expansão hexadecimal [com os 16 algarismos 0 – –9 e A – F] d') o seu hash SHA-256 comece com 'oooo'. Existem $16^4 = 4096$ combinações de quatro dígitos hexadecimais; por isso, se os valores da função hash são uniformemente distribuídos, esperamos cerca de 4096 tentativas para encontrá-lo. Com efeito, após 4251 tentativas, que levam um milissegundo num computador moderno, obtemos

```
"Hello, world!0" => 1312AF178C253F84028D480A6ADC1E25...
"Hello, world!1" => E9AFC424B79E4F6AB42D99C81156D3A1...
"Hello, world!2" => AE37343A357A8297591625E7134CBEA2...
...
"Hello, world!4248" => 6E110D98B388E77E9C6F042AC6B49...
"Hello, world!4249" => C004190B822F1669CAC8DC37E761C...
"Hello, world!4250" => 0000C3AF42FC31103F1FDC0151FA7...
```

Porém, em Bitcoin, o objeto hasheado, o cabeçalho do bloco, é mais complexo, em particular, porque contém a raiz da árvore (de Merkle) das transações (= um hash).

Dificuldade. O campo `dificuldade` no cabeçalho do bloco compara

- o esforço computacional que *está* (isto é, atualmente) em média necessário para encontrar um nem médiaovo bloco da blockchain

com

- o esforço computacional que *era* em média necessário para encontrar o primeiro bloco (o bloco 'Génesis') da blockchain:

No início, para o primeiro bloco, o bloco Génesis, ser aceite na cadeia, a expansão binária do hash do seu cabeçalho precisou de começar com 32 zeros (= o tamanho, em bites, do `nonce` no cabeçalho do bloco); isto é, eram necessárias 2^{32} (cerca de 4 bilhões) computações de hashes para encontrar um bloco com um tal hash (que levam cerca de 15 minutos em um notebook atual).

Desde então, o número dos zeros é reajustada depois de cada 2016-ésimo bloco para garantir a computação do cabeçalho pela rede levar, em média 10 minutos. (Se a computação de um novo bloco leva, em média, 10 minutos, então a de 2016 novos blocos leva, em média, 2 semanas.)

Cada reajuste (após 2016 blocos) calcula a nova dificuldade como proporção entre

- o tempo alvo $A = 2016 \cdot 10$ minutos (cerca de duas semanas), e
- o tempo U em minutos que a criação dos últimos 2016 blocos levou;

isto é,

$$\text{nova dificuldade} = A/U \cdot \text{ultima dificuldade}$$

Para abrandar saltos, a nova dificuldade é no máximo 4, isto é, mesmo se $A/U > 4$ (quer dizer, a rede levou menos de três dias e meio), então a nova dificuldade é 4.

Mineração. Computar de um cabeçalho de um bloco cujo hash seja menor que o alvo atual, isto é, cuja expansão binária comece com um número suficiente de zeros, é chamada de *minerar*; o computador que minera (no Brasil) de *minerador* (para não sobrecarregar o significado da palavra *mineiro*).

A dificuldade é em cada momento (com uma dilação de cerca de duas semanas) proporcional à juntada força computacional dos mineradores: Quanto mais mineradores, mais difícil, quanto menos mineradores, mais fácil. Atualmente, em 2018, a expansão binária do hash tem de começar com ≥ 80 zeros para ser aceito na rede.

Tempo de Computação. Como o hash usado no bitcoin é criptográfico (atualmente SHA-256), isto é, a sua saída não permite deduzir a entrada, a única maneira prática de encontrar um cabeçalho tal que o seu hash seja suficientemente pequeno é a de força bruta, isto é, provar simplesmente todas as possibilidades. Como uma função de hash praticamente é uniformemente aleatória, isto é, as probabilidades entre as saídas são as mesmas, leva para um número n (dos dígitos binários iniciais iguais a 0) em média cerca de 2^n tentativas até encontrar um tal cabeçalho.

1. No início da blockchain, $n = 32$ e eram necessárias 2^{32} (cerca de 4 bilhões) computações de hashes para encontrar um bloco com um tal hash (o que levou cerca de 15 minutos em um notebook).
2. Atualmente, $n \geq 80$ e são necessárias em média mais de $2^{80} \approx 1{,}2 \cdot 10^{24}$ (um septilhão (= um trilhão vezes um trilhão)) de hashes para o bloco ser aceite na cadeia. Um processador rápido de um microcomputador, por exemplo, o `Intel Core i7 2600`, consegue cerca de $2{,}4 \cdot 10^7 = 24$ milhões hashes por segundo. Por isso, demorará em média cerca de $5 \cdot 10^{16}$ segundos ($> 10^9$ anos, isto é, um bilhão de anos) até ele encontrar um bloco que será aceite na cadeia.

Isto passa uma ideia da juntada força computacional atual dos mineradores, já que calculam este hash em média em dez minutos.em média Como a função hash usada pelo `Bitcoin` é SHA-256, como discutido em Seção 10.4, ela é computada rapidamente por um CPU, um microprocessador a uso geral de um computador pessoal, e em particular,

- para o consumidor comum, um GPU, por um processador gráfico; cerca de 100 vezes mais rápido do que um CPU, e

- para alguém com mais recursos, por um ASIC, um microprocessador, ou Circuito Integrado, Adaptado para uma aplicação eSpecífica, aqui a computação do SHA-256; cerca de 100 000 vezes mais rápido do que um CPU.

Com efeito, o gasto de energia para a mineração equivale a cada momento o da Austria inteira. Por isso, surgiram conceitos alternativos à prova de trabalho (= proof of work), por exemplo, a *prova de cacife* (= proof of stake) onde o criador do bloco é determinado pela sua posse em vez da sua força computacional. Contudo, até agora estes conceitos alternativos não funcionaram tão bem na prática.

Isto dito, uma vez encontrado um tal bloco e o seu hash, dados os dois, a verificação que o hash seja suficientemente pequeno é rápida (e simplesmente consiste em calcular o hash do bloco e comparar com o hash dado).

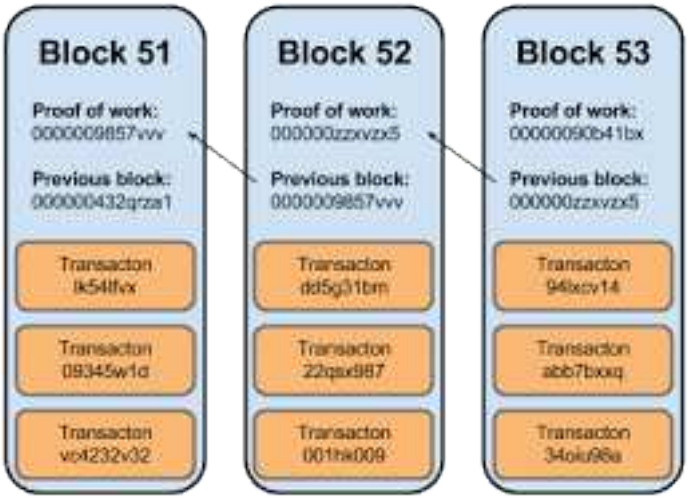


Figura 103: Blockchain

Tempo Disponível. O intervalo para criar um novo bloco é, em média, 10 minutos.

- O processo de Poisson mostra que a probabilidade para um bloco ter encontrado neste intervalo de 10 minutos é cerca de 63% (mais exatamente, $1 - 1/e$).

- Isto é, quase dois terços dos blocos serão encontrados em, no máximo, 10 minutos.
- Em 30 minutes, a probabilidade aumenta a 95%, e em uma hora a 99.7%.

Alterar o Cabeçalho. Para alterar o cabeçalho do bloco, recordemo-nos de que os dados variáveis do cabeçalho são (cf. Seção 11.3)

- o hash da raiz da árvore de merkle das transações, e
- um `nonce`, um campo sem conteúdo (semântico) que serve para alterar o hash do bloco sem alterar o seu conteúdo (semântico).

Em particular, o conteúdo principal do bloco, as transações, entram no seu hash só indiretamente pelo hash da raiz da árvore de merkle.

Atualmente, com $n \geq 80$, como o `nonce` tem só 4 baites, isto é, 32 bites, raramente (mais exatamente, com uma chance de cerca de 2^{32-80}) entre todos os seus possíveis valores encontra-se um tal que o hash do bloco seja suficientemente pequeno. Por isso, modificam-se, além do nonce,

- o `timestamp`, a data de criação em segundos contados desde 1970-01-01 às 00:00 UTC,
- as transações, por exemplo, a sua ordem, e
- a `coinbase`, a mensagem (de 100 baites) que acompanha a primeira transação no bloco e transfere uma recompensa ao minerador pelo seu trabalho. Enquanto a `coinbase` providencia amplo espaço para alterar o hash, é uma opção mais custosa que o `timestamp` ou `hash`, porque, fazendo parte das transações, a sua alteração implica a recomputação dos hashes da árvore de merkle que armazena as transações. (Lembremo-nos de que é só o hash da raiz da árvore que entra no cabeçalho do bloco.)

Recompensa. Ao descobrir um novo bloco, o minerador transmite-o à rede e os nós verificam, entre outros,

- que o hash do cabeçalho comece de fato com o número de 0s exigidos pela dificuldade atual, e
- que todas as transações sejam válidas.

Após este bloco ser estendido por pelo menos 100 blocos (e assim a rede ter certeza que ele permaneça na blockchain, a cadeia mais comprida), o descobridor

- Ganha uma certa quantia de bitcoins como recompensa para o trabalho da descoberta: inicialmente, em 2009 foram 50, atualmente, em 2018, são 12,5; o valor é dividida pela metade após 210000 blocos (como um bloco é minerado de 10 em 10 minutos, isto leva cerca de 4 anos).
- Ganha todas as taxas de transação inclusas no bloco descoberto; um incentivo para incluir a transação no bloco.

Como a recompensa diminui gradualmente (até esvair-se após $21000000 = (50 + 25 + 12{,}5 + \cdots) \cdot 210000$ blocos), as taxas terão um papel mais e mais importante para incentivar os mineradores.

Para distribuir mais uniformemente o ganho da mineração, mineradores juntam as suas forças computacionais em grupos, `clusters`, para compartilharem os bitcoins recompensados.

11.5 Transação

Todos os bitcoins são gerados por mineração e transferidos aos mineradores, aos criadores do bloco, pela primeira transação de cada bloco, a `coinbase` que é destinada aos criadores.

Uma vez gerado, um(a moeda de) `bitcoin` muda de lugar por uma cadeia de assinaturas digitais: O dono transfere o seu `bitcoin` ao próximo

- pela sua assinatura (com a sua chave privada) de (hashes das) transferências anteriores em que ele recebeu este bitcoin, e
- pela indicação da quantia e de (um hash da) chave pública do próximo destinatário.

A soma constada na carteira da interface gráfica `Bitcoin Core` do bitcoin (desenvolvida inicialmente por Satoshi Nakamoto sob o nome `Bitcoin-Qt` e usado por 90% dos negociantes) é a soma de todas as transações que o dono da carteira recebeu: Para a Alice pagar certa quantia, por exemplo, 4 `bitcoin`, ao Bob,

1. o programa busca transações cuja soma é ≥ 4, por exemplo,

- uma de 2, e
- outra de 3 `bitcoin`,

2. transfere

    - a soma de 4 `bitcoin` ao Bob, e
    - o troco de 1 `bitcoin` à Alice.

Isto é, Alice ganhou outra transação (embora de um valor menor que as duas iniciais).

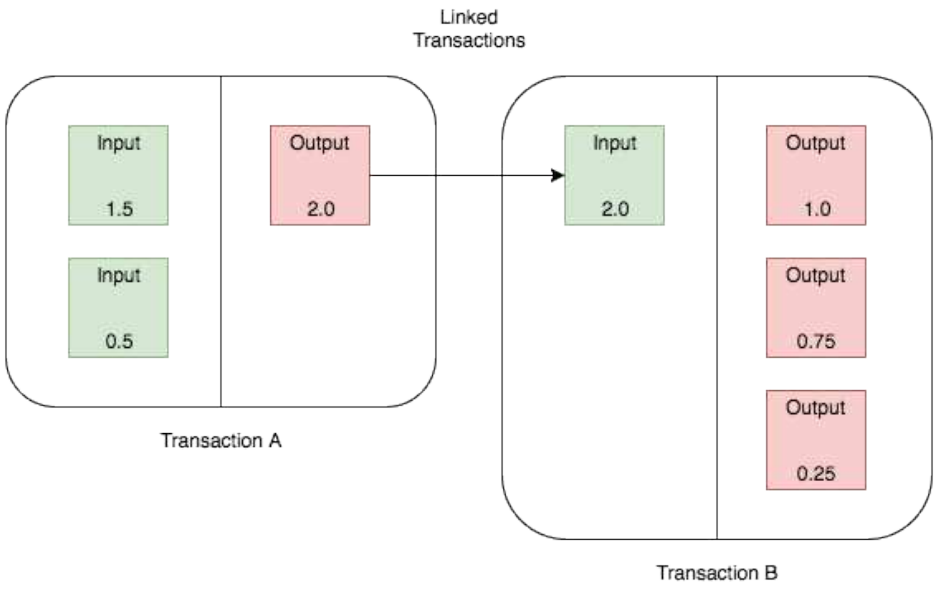


Figura 104: Esgotamento das entradas de uma transação

Endereço de Bitcoin. A identidade de cada negociante corresponde à sua chave assimétrica (ECDSA), um par de

- uma chave *pública* com 32 baites, e
- uma chave *privada*.

Para obter o seu *endereço*, várias funções de hash são aplicadas à chave pública; a sequência final de 25 letras é codificada pela Base58 cujos 58 algarismos são

- todos os números,
- todas as letras minúsculas, e
- todas as letras maiúsculas
- exceto o número e a letra 0O e as letras Il pelo risco da sua confusão.

Para mais detalhes, vide https://gobittest.appspot.com/Address:

Para calcular o hash da chave pública:

1. Aplica SHA-256 à chave pública ECDSA,
2. Aplica RIPEMD-160 (ao último resultado), e
3. Prefixa com 0x0000 para P2PKH, 0x0005 for P2SH.

Para acrescentar uma soma de verificação:

4. Aplica uma vez SHA-256,
5. Aplica outra vez SHA-256,
6. Usa os primeiro quatro baites como soma de verificação, e
7. Sufixa a soma de verificação ao resultado em 3.

Para abreviar o resultado (de uma maneira humanamente legível):

8. Codifica em Base58.

Para obter o formato do scriptPubkey usado em transações, precisa

1. Descodificar da Base58 à base hexadecimal,
2. remover a soma de verificação, e
3. remover o prefixo

e finalmente acrescenta certo código de instrução.

O uso de um hash em vez da chave pública como endereço,

- além de abreviá-lo,
- flexibiliza a transição a outro algoritmo de criptografia assimétrica caso o atual (ECDSA) seja comprometida um dia, e
- esconde a chave pública até a quantia recebida é gastada (para que a assinatura que indica a chave pública é necessária).

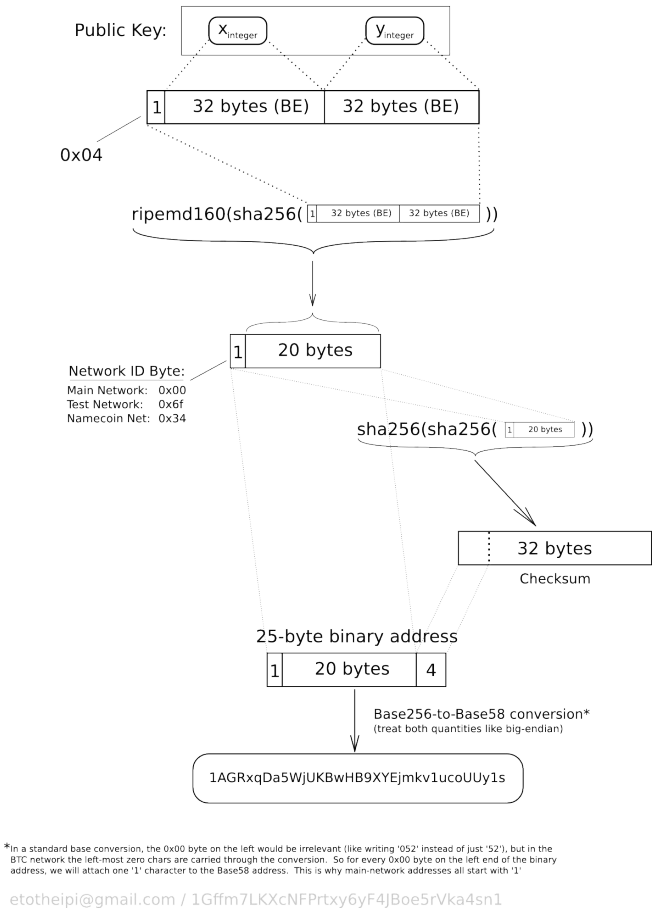


Figura 105: Conversão da Chave Pública

Tipos de Transações. Uma *transação* é uma transferência de bitcoins, primeiro transmitida na rede, em seguida colecionada em um bloco por um minerador e publicado. Todas as transações são públicas, enquanto (os hashes d') as chaves públicas são anónimas, isto é, não revelam a priori o nome dos donos.

Há dois tipos de transações,

- a da *geração* dos bitcoins ao criar um novo bloco, e
- a de negociação *entre* dois (grupos de) usuários, o remetente e o destinatário.

- A *transação de negociação* tem
  - um `input` que refere a saídas de transações em que o remetente recebeu bitcoins, e
  - um `output` que transfere aos destinatários uma quantia **igual** a da soma das saídas das transações da entrada.

    Frequentemente, a transação inclui um **troco**, isto é, o remetente figura como um dos destinatários: Se a soma é menor, a diferença é paga ao criador do bloco como incentivo a incluir a transação.

  Uma vez incluída em um bloco na cadeia, e este bloco ser estendido por um número suficiente (≥ 6) de outros, a transação pode ser considerada irreversível, é *confirmada*.

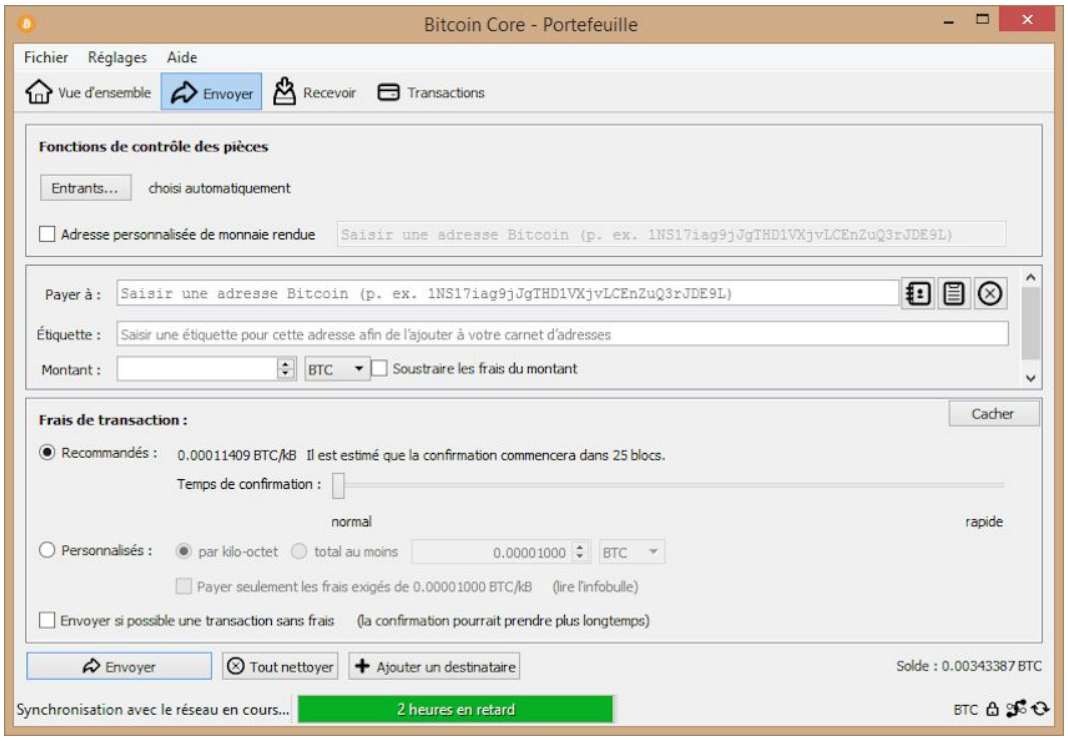


Figura 106: Transação no aplicativo `Bitcoin Core`. O pagamento de uma taxa de transação ajuda a diminuir o tempo de confirmação.

- A *transação de geração* é determinada pelo minerador:

– como única informação no input um campo `coinbase` de 100 baites (em vez de `scriptSig`) cujo conteúdo é arbitrário. (É frequentemente (ab)usado como `ExtraNonce`, isto é, `Nonce` adicional para a mineração, visto que o tamanho do `Nonce`, 4 baites, atualmente é insuficiente para encontrar um bloco cujo hash sejá suficientemente pequeno. Entretanto, diferente do nonce, o `ExtraNonce` entra só indiretamente na transação pelo hash da raiz da árvore de Merkle. Logo, a modificação do `ExtraNonce` requer a recomputação dos hashes do ramo da coinbase na árvore de Merkle das transações.)
– o `output` distribui a recompensa (de 12,5 bitcoins em 2018) aos favorecidos pelo minerador.

Etapas de uma Transação de Negociação. A Alice, para mandar 4 Bitcoins (ou 400000000 Satoshis) ao Bob,

1. No input,
    1. escolhe transações que recebeu cuja soma é $\geq$ a quantia que mandará ao Bob, por exemplo, de $2 + 3 \geq 4$, e
    2. refere aos seus hashes e, para cada transação, o índice no output que refere a ela como recipiente.
2. No output, indica
    1. a quantia 400000000 (em Satoshis) que mandará ao Bob,
    2. o endereço do Bob, um hash da sua chave pública.
    3. além da quantia do troco e do próprio endereço
3. Finalmente, no input,
    1. cria um hash
        - dos hashes e índices das transações do input,
        - dos endereços nos outputs dos quais recebeu do input,
        - do endereço do Bob e da quantia a ser transferida;
        - do próprio endereço e da quantia do troco. (Toda sobra será dada ao minerador.)
    2. assina este hash pela sua chave privada.

Isto é, Alice demonstra pela sua assinatura que as transações que recebeu pertencem a ela, pois

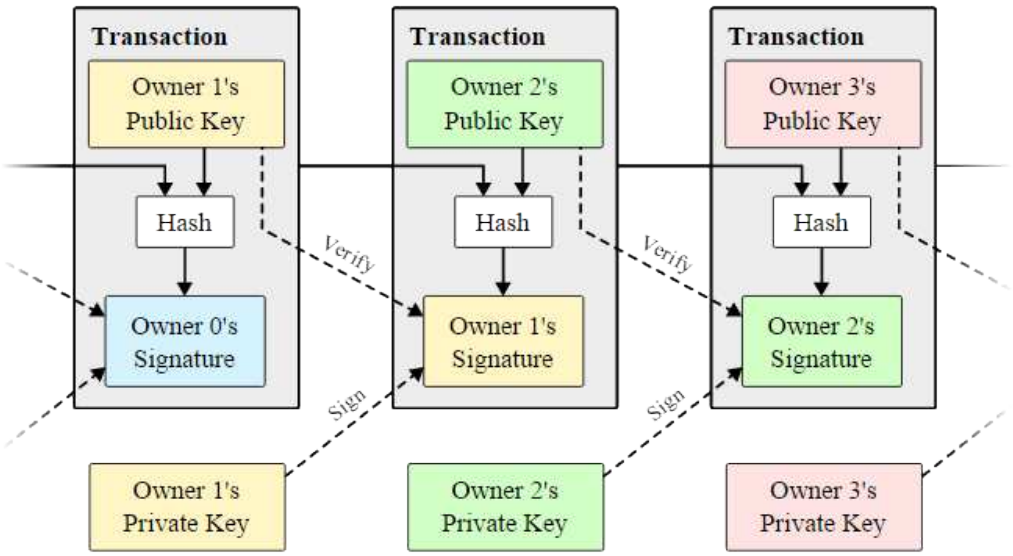


Figura 107: No hash assinado pelo remetente (= o antigo dono) entram o endereço (= chave pública) do destinatário (= o novo dono) e a transação recebida e que será gastada; em particular, nela figura a chave pública do remetente.

- o hash da sua chave pública é o seu endereço, e
- a chave privada que corresponde à chave pública assinou o hash da transação (com todos os dados alistados acima).

Dados de uma Transação. Os dados principais de uma transação que

- coleciona as quantias de uma singela transação na entrada, e

```
Previous tx: f5d8ee39a4...b9a6
Index: 0
scriptSig: 30450220...1d10
90db022100e2...1501
```

- destina um endereço na saída,

```
Value: 5000000000
scriptPubKey: OP_DUP OP_HASH160.4043...9d
OP_EQUALVERIFY OP_CHECKSIG
```

A entrada, o input, nesta transação recebe 50 BTC da saída, do output, #0 em transação $f5d8$... e envia os, isto é, 50 BTC ao endereço 4043... Assina todas estas informações por `30450` ....

Input. O **input** é uma referência a uma saída de uma transação anterior, frequentemente, uma transação tem várias. A sua soma (menos uma opcional taxa de transação paga ao minerador para inclui-la no seu bloco) é esgotada nas saídas da transação. No exemplo

- **Previous tx** é o hash da transação anterior,
- **Index** é a saída da transação referida, e
- **ScriptSig** é a primeira metade, uma assinatura, de um script que concatena, principalmente,
  - uma assinatura e
  - uma chave pública.

  A chave pública tem de coincidir com a dada pelo hash no índice do output da transação anterior, e é usada para verificar a assinatura (em mais detalhes, uma assinatura pelo algoritmo `ECDSA` com a curva secp256k1 de um hash da transação). A chave pública e a assinatura garantem a transação ser criada pelo destinatário da saída da transação anterior. Os dados assinados consistem principalmente
  - O *número de saídas* das antigas transações a serem gastadas,
  - para cada saída,
    * o **hash** da antiga transação cuja saída será gastada,
    * o **índice** da saída na antiga transação,
  - a nossa **chave pública**,
  - o *número das saídas* na nossa nova transação,
  - para cada saída,
    * a **quantia**, e
    * o **output script** que usualmente consiste na instrução

      `OP_DUP OP_HASH160 06f1b... OP_EQUALVERIFY OP_CHECKSIG`

      cujo código hexadecimal `06f1b...` indica o endereço do destinatário.

Output. O **output** instruí o envio da soma dos bitcoins da entrada:

- **Value** é o número de Satoshi (onde 1 BTC = 100000000 Satoshi) do valor do output, e
- O output script, usualmente **scriptPubKey** que indica o endereço (= um hash de uma chave pública). Mais exatamente, um *script* é uma sequência de instruções que começa com OP_ (OP_DUP, ...) formulada na linguagem Script; por exemplo,

  ```
  OP_DUP (0x76)
  OP_HASH160 (0xa9)
  0x14 (= indica que 20 baites de dados seguem)
  cba5...e8 (os dados)
  OP_EQUALVERIFY (0x88)
  OP_CHECKSIG (0xac)
  ```

  o que é concatenada a

  ```
  OP_DUP OP_HASH160 cba5f7...9ee8 OP_EQUALVERIFY OP_CHECKSIG
  ```

  Para o transporte e a assinatura, todas as instruções são codificadas por baites

  ```
  76a914cba5f746dc1e41a932a7d38be7aba95944619ee888ac
  ```

Todo output pode ser referido **uma única vez só** em uma transação posterior; por isso, para evitar perda, a soma **inteira** do input precisa ser enviada! Por exemplo, se a soma da entrada é 5 BTC, mas o usuário quer apenas pagar 1 BTC, então ele cria dois ouputs,

- um de 1 BTC para o destinatário, e
- outro de 4 BTC para ele, o remetente (= o **troco**).

Toda a quantia que não foi gastada será considerada uma *taxa de transação*, isto é, transferida ao criador do bloco (como incentivo de incluir esta transação antes de outras).

11.6 Processamento de transações

Na rede do bitcoin, há

- os *nós completos*, que transmitem e validam as transações, e
- os *mineradores*, que criam os blocos que estendem a cadeia.

Um minerador é incentivado

- a enviar o bloco minerado por ele aos nós o mais rápido possível — caso contrário, o bloco minerado por outro será usado,
- a receber blocos minerados por outrem, para continuar a minerar a partir deste bloco como fim da cadeia, e
- validar as transações porque todo blocos com uma transação inválida é rejeitado pelos nós completos.

Porém, em geral, não é incentivado a enviar o bloco minerado por outrem; em vez disto, esconde a sua existência até ter minerado um próprio bloco que o estende. Esta transmissão instantânea cabe aos nós.

Um usuário é incentivado a manter um nó para poder a todo ponto garantir a integridade da blockchain. Porém, a manutenção de um nó inteiro é sobretudo uma necessidade altruísta que é essencial para o funcionamento do Bitcoin: É só razoavelmente seguro usar um nó parcial (que não tem a cadeia inteira) enquanto a maior parte da rede são nós inteiros que garantem a validade.

Processamento. Resumimos o processamento de uma transação, desde a sua emissão até a sua conclusão:

1. O usuário envia uma transação pelo seu aplicativo de carteira.
2. A transação é difusa pelos nós e faz parte do 'acervo de transações não-confirmadas'.
3. Mineradores,
    1. escolhem transações deste acervo (de preferência, as que lhe pagam a maior taxa de transação, a diferença entre a entrada e saída),
    2. verificam-nas (por exemplo, se o saldo do pagador é suficiente), e
    3. acrescentam-nas ao bloco que tentam gerar;

comumente, até esgotar o tamanho máximo do bloco (de 1 Megabaite).

4. Cada minerador tenta de modificar o bloco de tal maneira que o seu hash seja suficientemente pequeno (por exemplo, atualmente, que comece com 10 baites nulos). A probabilidade de encontrar um tal bloco, isto é, a dificuldade do problema, é a mesma para todos os blocos e mineradores. Esta dificuldade é ajustada cada 2016-ésimo bloco (isto é, após cerca duas semanas) dependendo da força computacional total dos mineradores para um tal bloco sempre ser encontrado em média em dez minutos.

   Quando um novo bloco é concatenado à cadeia, todos os mineradores têm de recomeçar com outro bloco, já que, por exemplo,

   - umas transações foram feitas,
   - o indicador (ao hash do bloco anteiror) mudou.

5. Todo minerador que encontra um bloco válido transmite-o aos nós da rede.

6. O nó verifica se o bloco seja válido, isto é,

   - que todas as transações sejam válidas, e
   - que o hash seja suficientemente pequeno.

7. Se o nó acorda sobre a validade do bloco, então o concatenam à cadeia. Uma confirmação de uma transação é cada extensão da cadeia por um bloco após aquele que contém a transação. A extensão da cadeia pelo bloco que contém a transação conta como primeira confirmação, a após este bloco como segunda confirmação, e assim por diante. A chance da cadeia mudar após seis confirmações (isto é, após, em média, uma hora) é minúscula; por isso, comumente, uma transação é vista como conclua após a sexta confirmação.

Verificação. Para o nó verificar rapidamente a validade da entrada de uma transação, isto é, se as transações redimidas não foram gastadas ainda, ele atualiza a cada momento o banco-de-dados *Unspent Transaction Ouput* (UTXO) de todas as transações que ainda não foram gastadas. Atualmente, o UTXO tem cerca de 1 GB e é guardada na memória para garantir consultas rápidas.

Recordemo-nos de que cada transação é única, e pode ser gastada apenas uma vez, e inteiramente (para obter o troco, cria se outra transação). Quando uma transação é transmitida,

- as transações anteriores que figuram no input são removidas do `UTXO`, e
- as novas transações do output são adicionadas ao `UTXO`.

Para impedir o gasto duplo, o nó verifica se uma transação é no `UTXO`: caso sim, permite a transação, caso contrário, a impede.

Confirmação. Uma *confirmação* de uma transação é cada extensão da cadeia por um bloco após aquele que contém a transação. A extensão da cadeia pelo bloco que contém a transação conta como *primeira* confirmação, a após este bloco como *segunda* confirmação, e assim por diante.

A chance da cadeia mudar após *seis* confirmações (isto é, após, em média, uma hora) é minúscula; por isso, comumente, uma transação é considerada conclusa após a sexta confirmação. Por exemplo, a interface gráfica padrão para o bitcoin, o `Bitcoin Core`, mostra uma transação como confirmada quando a profundeza da transação atingiu 6 blocos.

Porém, esta *profundeza*, o número de confirmações para `bitcoins` *transferidos*, de 6 blocos é arbitrária, e pode mudar com cada transação. Em contraste, `bitcoins` *minerados* só podem ser gastados quando o bloco gerado tem uma profundeza de 100 blocos, isto é, foi estendido na cadeia por 100 blocos.

Referências Bibliográficas

Bach, Eric. 1984. *Discrete Logarithms and Factoring*. Computer Science Division, University of California Berkeley.

Daemen, Joan. 1995. «Cipher and hash function design strategies based on linear and differential cryptanalysis». Tese de doutoramento, Doctoral Dissertation, March 1995, KU Leuven.

Daemen, Joan, e Vincent Rijmen. 1999. «AES proposal: Rijndael». http://www.cs.miami.edu/home/burt/learning/Csc688.012/rijndael/rijndael_doc_V2.pdf.

———. 2002. *The design of Rijndael*. Information Security and Cryptography. Springer-Verlag, Berlin. https://doi.org/10.1007/978-3-662-04722-4.

Diffie, Whitfield, e Martin Hellman. 1976. «New directions in cryptography». *IEEE transactions on Information Theory* 22 (6): 644–54.

Goldwasser, Shafi, e Silvio Micali. 1984. «Probabilistic encryption». *J. Comput. System Sci.* 28 (2): 270–99. https://doi.org/10.1016/0022-0000(84)90070-9.

Gordon, Daniel M. 1993. «Discrete logarithms in GF(p) using the number field sieve». *SIAM J. Discrete Math.* 6 (1): 124–38. https://doi.org/10.1137/0406010.

Heys, Howard M. 2002. «A tutorial on Linear and Differential Cryptanalysis». *Cryptologia* 26 (3): 189–221.

Hoffstein, Jeffrey, Jill Catherine Pipher, e Joseph H Silverman. 2008. *An Introduction to Mathematical Cryptography*. Vol. 1. Springer.

Joux, Antoine. 2009. *Algorithmic cryptanalysis*. Chapman & Hall/CRC Cryptography and Network Security. CRC Press, Boca Raton, FL. https://doi.org/10.1201/9781420070033.

Joux, Antoine, e Kim Nguyen. 2003. «Separating decision Diffie-Hellmann from computational Diffie-Hellman in cryptographic groups». *J. Cryptology* 16 (4): 239–47. https://doi.org/10.1007/s00145-003-0052-4.

Lenstra, A. K., H. W. Lenstra Jr., M. S. Manasse, e J. M. Pollard. 1993. «The number field sieve». Em *The development of the number field sieve*, 1554:11–42. Lecture Notes in Math. Springer, Berlin. https://doi.org/10.1007/BFb0091537.

Maurer, Ueli M., e Stefan Wolf. 1999. «The relationship between breaking the Diffie-Hellman protocol and computing discrete logarithms». *SIAM J. Comput.* 28 (5): 1689–1721. https://doi.org/10.1137/S0097539796302749.

Menezes, Alfred J., Paul C. van Oorschot, e Scott A. Vanstone. 1997. *Handbook of applied cryptography*. CRC Press Series on Discrete Mathematics and its Applications. CRC Press, Boca Raton, FL. http://www.cacr.math.uwaterloo.ca/hac/.

Rivest, Ronald L, Adi Shamir, e Leonard Adleman. 1978. «A method for obtaining digital signatures and public-key cryptosystems». *Communications of the ACM* 21 (2): 120–26.

Shoup, Victor. 1997. «Lower bounds for discrete logarithms and related problems». Em *Advances in cryptology—EUROCRYPT '97 (Konstanz)*, 1233:256–66. Lecture Notes in Comput. Sci. Springer, Berlin. https://doi.org/10.1007/3-540-69053-0_18.

Sweigart, Albert. 2013. *Hacking Secret Ciphers with Python*. Gratis.