Técnicas Criptográficas

José Manuel E. Valença *

12 de Janeiro de 2010

*Departamento de Informática, Universidade do Minho, Campus de Gualtar Braga

3.Incerteza e Segurança

A segurança dos sistemas de informação está rigidamente ligada a duas noções essenciais: a noção de *segredo* e a noção de *confiança*.

Entendemos que "*segredo*" exprime "controlo da informação": um *agente* A (alguém, munido de conhecimento, segredos e recursos computacionais próprios) consegue intervir num *acto* do qual extrai o valor k. Aqui a noção de "acto" é bastante genérica: inclui a simples execução de uma computação mas inclui também acções mais complexas (por exemplo, esquemas ou protocolos) onde intervêm vários agentes.

O segredo pode ou não ser *partilhado*; não sendo partilhado, "acredita-se" que nenhum outro agente A' (sem acesso ao conhecimento ou recursos de A) consegue executar um acto com os mesmos efeitos. Se for partilhado, outro agente pode participar no mesmo acto e também extrair a informação k. Pode também acontecer que o segredo não seja partilhado (só um agente consegue extrair k do acto) mas o próprio acto exija a intervenção de vários agentes.

"Acreditar" prende-se com a noção de *confiança*; genericamente entendemos *confiança* do agente A em relação ao agente A', como uma medida da crença que A tem de que qualquer acto onde A' intervenha não lhe permite violar "direitos" de A.

Concretamente, os "direitos" em causa ligam-se à preservação de segredos; aqui, confiança de A em relação a A' exprime a crença de que A' não consegue intervir em nenhum acto que lhe permita extrair um segredo de A.

Para a crença de A seja bem fundada tem de existir uma medida da dificuldade computacional de A' em determinar um segredo k de A. Essa medida é a **incerteza**. A fundamentação da crença de A reside, então, no facto de a "incerteza" de k com os privilégios de A' (conhecimento, segredos e recursos computacionais) ser "demasiado grande" para que, realisticamente, A' consiga obter o segredo.

A noção de incerteza aplica-se a *eventos*: a incerteza de um evento e mede a *dificuldade de previsão* de e.

A combinação *"A intervém no acto α do qual extrai a informação k"* é o paradigma de evento que é relevante ao estudo da segurança. Para exprimir a incerteza de tal evento usaremos a notação $\vartheta(k : \alpha|A)$. Esta notação é simplificada para $\vartheta(k|A)$, se se considerar todos os "eventuais actos" onde A pode intervir, ou para $\vartheta(k : \alpha)$ se consideramos um agente universal com privilégios implicitamente definidos.

Pode-se considerar um modelo mais geral de eventos baseado nas máquinas de Turing livres de prefixos; o quadro de referência que daqui resulta designa-se por **SEM (standard event model)**.

Tome-se uma PFTM universal $\mathcal{U}$ e descreva-se um evento e por uma string que é submetida como "input" de $\mathcal{U}$; assume-se que o evento e *ocorre* quando esse "input" é aceite por $\mathcal{U}$. A incerteza $\vartheta(e)$ vai ser definida como uma medida da complexidade computacional da decisão $\mathcal{U}(e)\downarrow$.

Como medida de complexidade usa-se os limites à recursividade μ. Representamos por $\mathcal{U}^l$ a PFTM universal em que a μ-recursividade está limitada a l iterações; isto é, $\mathcal{U}^l$ obtém-se de $\mathcal{U}$ substituindo todas as construções da

forma $(\mu\, y \cdot h(y,x))$ por $(\mu\, y < l \cdot h(y,x))$. Nestas circunstâncias, no modelo SEM a incerteza define-se como

$$\vartheta(e) \;=\; \min\{\, |l| \;\mid\; \mathcal{U}^{l}(e)\!\downarrow \,\} \tag{56}$$

3.1 Circuitos

Um exemplo particularmente simples de eventos e representação de incerteza, ocorre com a noção de *circuito*.

96 Noção
Um conjunto enumerado $\mathcal{N} \subseteq \varpi$ *é uma* **família de circuitos** *quando todo* $n \in \mathcal{N}$ *é* 0, 1 *ou então é um par* $a\|b$ *com* $a, b \in \mathcal{N}$.

O **selector** $/$ *é a função primitiva recursiva* $\mathbb{B}^* \times \mathcal{N} \to \mathcal{N}$ *definida por*

$$n/\varepsilon = n \quad , \quad 0/y = 0 \quad , \quad 1/y = 1 \quad , \quad (a\|b)/(0\,y) = a/y \quad , \quad (a\|b)/(1\,y) = b/y \qquad (57)$$

Escreve-se $m \leqslant n$ *sse* $n/y = m$, *para algum* y. *A linguagem* $\Lambda(n)$ *é a menor linguagem* L *que verifica* $(\forall\, y \in L)(\forall\, u)\,[n/(y\,u) \neq 0]$.

Por indução verifica-se facilmente que $n/(y\,u) = (n/y)/u$. Essencialmente temos $y \in \Lambda(n)$ quando todos os sub-circuitos atómicos de (n/y) são 1. Escreve-se $n \ni y$, e diz-se que y é **aceite** por n, quando $n/y = 1$. Claramente, todo y aceite por n tem um prefixo em $\Lambda(n)$.

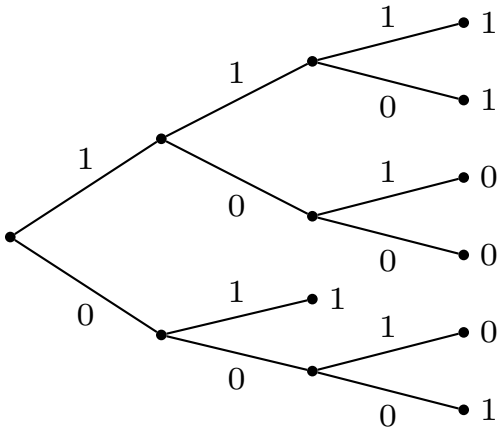
A relação $\leqslant$ é bem-fundada e, por essa razão, cada $\Lambda(n)$ é uma linguagem finita e livre de prefixos. A função $\Lambda : \varpi \to \varpi$ é, claramente, recursiva mas, normalmente, não é computável.

Ao invés, é possível definir uma função recursiva primitiva Λ^{-1} que mapeia qualquer linguagem L, finita e livre de prefixos, num circuito $n = \Lambda^{-1}(L)$ tal que $\Lambda(n) = L$. Basta fazer

$$\Lambda^{-1}(\emptyset) \simeq 0 \quad , \quad \Lambda^{-1}(\{\varepsilon\}) \simeq 1 \quad , \quad \Lambda^{-1}(A \uplus B) \simeq \Lambda^{-1}(A) \,\|\, \Lambda^{-1}(N) \tag{58}$$

Pode-se ver circuitos como árvores binárias particularmente simples: as folhas são marcadas com 0 ou com 1, e os nodos não têm qualquer marca; a sub-árvore esquerda é marcada com 0 e a sub-árvore direita com 1.

EXEMPLO 18: Considere-se o circuito $n = ((1\|0)\|1) \,\|\, ((0\|0)\|(1\|1))$.



O circuito n é representado pela árvore à esquerda.

A selecção n/y denota a sub-árvore que se obtém percorrendo n, partindo da raiz, com a string y, "virando à esquerda" ou "virando à direita" consoante o bit de y é 0 ou 1.

Se for $y = 0\,1$, tem-se $n/y = 1$. Se for $y = 0\,1\,1$ tem-se $n/y = 1/1 = 1$.

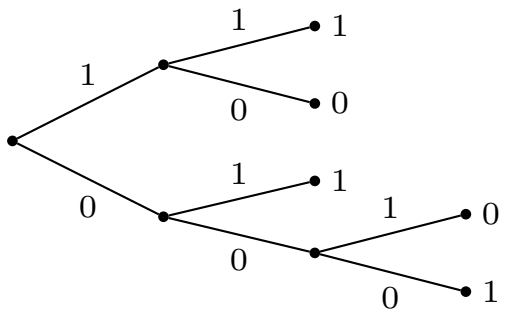
A linguagem $\Lambda(n)$ é formada pelas strings y que determinam sub-árvores n/y que só contêm folhas 1. Neste caso será $\Lambda(n) = \{\,0\,0\,0\ ,\ 0\,1\ ,\ 1\,1\,\}$.

EXEMPLO 19: Considere-se a linguagem $L = \{\,0\,0\,0\ ,\ 0\,1\ ,\ 1\,1\,\}$.

L pode ser reduzido, sucessivamente, da forma seguinte

$$\{\,000, 01, 11\,\} \rightsquigarrow \{00, 1\} \uplus \{1\} \rightsquigarrow (\{0\} \uplus \{\varepsilon\}) \uplus (\emptyset \uplus \{\varepsilon\})$$
$$\rightsquigarrow ((\{\varepsilon\} \uplus \emptyset) \uplus \{\varepsilon\}) \uplus (\emptyset \uplus \{\varepsilon\})$$

Desta forma $\Lambda^{-1}(L) = ((1\|0)\|1)\|(0\|1)$. Este circuito pode ser representado pela árvore á direita que representa, de forma mínima, a mesma informação que está associada ao circuito do exemplo anterior.



Através das linguagens $\Lambda(n)$ é possível definir várias construções em circuitos. Nomeadamente

- Dois circuitos n, m são **equivalentes** quando geram a mesma linguagem; $n \cong m \Leftrightarrow \Lambda(n) = \Lambda(m)$.
- A **redução** de n, representada por $n^{\sim}$, é o menor circuito equivalente a n; constrói-se como $\Lambda^{-1}(\Lambda(n))$. Desta forma um circuito n está reduzido quando todo $y \in \Lambda(n)$ é aceite por n. Note-se que a redução $n^{\sim}$ não está, necessariamente, contida na mesma família de circuitos que contém n.

  Equivalentemente pode-se reduzir n efectuando, sucessivamente, as substituições $(0\|0) \rightsquigarrow 0$ e $(1\|1) \rightsquigarrow 1$. Definindo $n^{\sim}$ deste modo, $\Lambda(n)$ é o conjunto de todas as strings aceites pela redução $n^{\sim}$.
- A **simultaneidade** de n, m, representada por $n \cdot m$ é o menor circuito que aceita todas as strings que são aceites simultaneamente por $n^{\sim}$ e $m^{\sim}$; isto é, $n \cdot m = \Lambda^{-1}(\Lambda(n) \cap \Lambda(m))$.

Alguns parâmetros permitem aferir a complexidade computacional destas construções.

Tendo em atenção que todo o circuito é, afinal de contas, um inteiro, o ponto de partida é o **tamanho** $|n|$ desse inteiro. Porém circuitos são também árvores binárias e, por isso, é natural ter em atenção a **profundidade** $d(n)$ da árvore n; temos $d(0) = d(1) = 0$ e $d(a\|b) = 1 + \max\{d(a), d(b)\}$.

Finalmente, o **comprimento** $\lambda(n)$, define-se[25] por $\lambda(n) = \min\{|y| \mid y \in \Lambda(n)\}$. A medida $\lambda(n)$ exprime a complexidade da decisão $(\exists\, y)\,[n \ni y]$ ou, equivalentemente, da decisão $n \geqslant 1$. Genericamente, dados circuitos n, m, o **comprimento condicional** $\lambda(n|m) = \min\{|y| \mid n/y = m\}$ exprime a complexidade computacional associada à decisão $n \geqslant m$.

Obviamente $\lambda(n) \leq \lambda(n|1)$ e, genericamente, $\lambda(n) \leq \lambda(n|m) + \lambda(m)$.

97 FACTO
Tem-se $|n| = O(2^{|d(n)|}$ *e, para circuitos que verifiquem* $n \geqslant 1$, *tem-se* $\lambda(n) = \Theta(d(n))$.

Tomando como referência a profundidade $d(n)$, o comprimento $\lambda(n)$ é da ordem de $d(n)$, enquanto que o tamanho do inteiro n é da ordem de $2^{d(n)}$. Por isso faz sentido tomar como referência para a análise de complexidade a profundidade $d(n)$.

Neste contexto procuremos analisar a complexidade das várias construções com circuitos n tomando por referência quer o seu tamanho $|n|$, a sua profundidade $d(n)$ ou o seu comprimento $\lambda(n)$.

[25] Convenciona-se que $\min\{|y| \mid y \in \emptyset\}$ é ∞.

Os dois problemas essenciais são:

1. **Aceitação:** Dados y, n verificar se y é aceite por n; isto é, se $n/y = 1$.
   Este é um problema que admite uma solução linear com $d(n)$ e com $|y|$.
2. **Verificação** Verificar se $n \geqslant 1$; equivalentemente, encontrar y tal que $n \ni y$.
   Este problema é NP ("não-determinístico polinomial"): se existir um oráculo que, em tempo constante, consiga gerar uma possível solução y, testar se y é ou não uma solução realiza-se em tempo linear.

Estes dois problemas básicos estão relacionados com vários outros problemas parcelares.

- **Cópia:** obter um "clone" de um circuito n.
  Este problema é essencial à implementação de qualquer função que receba circuitos como "input"; tem uma solução exponencial com $d(n)$.
- **Igualdade:** verificar a igualdade $n = m$ de dois circuitos.
  A definição indutiva não admite uma solução polinomial; no entanto é possível usar a solução do problema da cópia, para comparar dois circuitos em tempo constante.
- **Redução:** dado n calcular $n\tilde{}$ usando as substituições $(0\|0) \rightsquigarrow 0$ e $(1\|1) \rightsquigarrow 1$.
  Esta função tem de percorrer toda a árvore e por isso a sua complexidade é, essencialmente, exponencial.
- **Cálculo da equivalência, Λ ou simultaneidade**
  São problemas ligados directamente à redução e, por isso, têm a mesma complexidade.

O comprimento $\lambda(n)$ é uma medida base de quão difícil é prever o evento $n/y = 1$. Pode-se afirmar que $\lambda(n)$ é um limite superior à incerteza $\vartheta(y : n/y = 1)$.

Quase sempre interessa-nos analisar medidas de incerteza que se apliquem, uniformemente, a uma família de circuitos e não a um circuito específico. A procura de medidas mais de incerteza leva-nos ao conceito de *representação*

98 NOÇÃO
Seja $\mathcal{N}$ uma família de circuitos. Sejam $\{f_n\}$ uma indexação computável de funções que preservam prefixos ($x \leq y$ implica $f_n(x) \leq f_n(y)$) e h uma função computável. O par (f, h) é uma **representação** *de $\mathcal{N}$ quando, para todo $n \in \mathcal{N}$ e todo y, tem-se $h(n) \ni y \Rightarrow n \ni f(y, n)$.*

Note-se que, existindo uma representação (f, h) de $\mathcal{N}$, decidir se $n \geq 1$ pode ser resolvido indirectamente: basta verificar se se verifica $h(n) \geqslant 1$. De facto, existindo y tal que $h(n) \ni y$, então também existe $u = f(y, n)$ tal que $n/u = 1$. Se for $\lambda(h(n))$ substancialmente inferior a $\lambda(n)$, a incerteza $\vartheta(y : n/y = 1)$ é melhor traduzida pelo comprimento $\lambda(h(n))$.

É sempre possível encontrar uma representação de uma qualquer família de circuitos; trivialmente a identidade determina uma representação. No entanto só são significativas aquelas que conduzem a uma alteração substantiva da incerteza em $\mathcal{N}$. Tais representações designam-se por **ataques**.

Não sendo consensual o que pode significa "uma alteração substantiva da incerteza em $\mathcal{N}$", várias interpretações são possíveis da noção de "ataque". Assim, a representação (f, h) conduz a um ataque quando,

1 Para todo $n \in \mathcal{N}$, tem-se $\lambda(h(n)) \ll \lambda(n)$, ou

2 Existe uma sub-família $N_h \subseteq \mathcal{N}$, computável e não-desprezável, tal que $\lambda(h(n)) \ll \lambda(n)$ ocorre para todo $n \in N_h$, ou

3 Existe uma sub-família não desprezável $N_h \subseteq \mathcal{N}$ tal que $\lambda(h(n)) \ll \lambda(n)$ ocorre para todo $n \in N_h$.

Obviamente que estas percepções de "ataque" são, sucessivamente, mais fortes: **1** implica **2** que implica **3**.

No primeiro caso exige-se uma redução substancial no comprimento, para todos os circuitos na família; no último caso apenas se exige que isso se verifique para uma sub-família "não-desprezável" de $\mathcal{N}$; o segundo caso é algo intermédio: exige que seja possível computar a sub-família onde existe a referida redução de comprimentos.

A incerteza $\vartheta(y : n/y = 1)$ pode apenas ser aferida através de limites superiores. Existindo uma representação f, h de $\mathcal{N}$, tem-se

$$\vartheta(y : n/y = 1) \leq \lambda(n) \quad , \quad \vartheta(y : n/y = 1) \leq \vartheta(y : h(n)/y = 1) \tag{59}$$

□

Circuitos são modelos de computabilidade particularmente simples no que respeita à concepção de incerteza. Dada a sua simplicidade coloca-se a questão de saber se são realmente úteis na descrição das situações concretas onde essa incerteza é necessária. A resposta a esta questão procuraremos construí-la nas próximas secções, gerando objectos matemáticos que sejam representáveis por circuitos mas que tenham relevância criptográfica.

3.2 Funções Booleanas e sua Incerteza

O projecto e segurança de muitas técnicas criptográficas estão ligadas ao estudo das funções da forma $f : \mathbb{B}^n \longrightarrow \mathbb{B}$ que tomam como argumento uma sequência finita de *bits* de comprimento fixo e devolvem um simples bit.

As funções booleanas de argumento $\mathbb{B}^n$ identificam-se com os sub-conjuntos de $\mathbb{B}^n$. De facto, cada f determina um conjunto, designado por **suporte** de f, e definido por

$$\mathsf{sup}(f) \;=\; \{\, x \in \mathbb{B}^n \;\mid\; f(x) = 1 \,\}$$

Inversamente, cada $S \subseteq \mathbb{B}^n$ tem uma função característica que é, obviamente, uma função booleana; S coincide com o suporte da sua função característica. O número de funções booleanas de n variáveis booleanas é, assim, $2^{|\mathbb{B}^n|}$. Como $\mathbb{B}^n$ tem 2^n elementos, o número de funções booleanas de argumento $\mathbb{B}^n$ será 2^{2^n}.

□

Das várias representações possíveis para funções em que os argumentos e resultados têm tamanho fixo, escolhemos algumas que são particularmente importantes:

1. Funções booleanas de n variáveis booleanas; i.e.

$$f : \mathsf{GF}(2)^n \longrightarrow \mathsf{GF}(2)$$

2. Funções booleanas sobre o corpo de Galois de ordem n ; i.e.

$$f : \mathrm{GF}(2^n) \longrightarrow \mathrm{GF}(2)$$

3. Quando o resultado tem $s > 1$ bits e o tamanho do argumento é um múltiplo de s, pode-se ter

$$f\colon \mathrm{GF}(2^s)^n \to \mathrm{GF}(2^s) \qquad \text{ou} \qquad f\colon \mathrm{GF}(2^{n\times s}) \to \mathrm{GF}(2^s)$$

   Esta representação assume $\mathrm{GF}(2^s)$ como uma descrição do resultado e interpreta o argumento de dois modos possíveis: ou como uma palavra com n componentes em que cada uma é também um elemento de $\mathrm{GF}(2^s)$ ou então como um elemento de um corpo de Galois de dimensão $n \times s$.

Nesta secção iremos analisar a primeira destas famílias de funções. Porém, vamos necessitar de alguns operadores binários que usam palavras de bits $x, y \in \mathbb{B}^n$ como argumentos.

- A **paridade** (já referida no capítulo 1) é a função booleana definida como

$$x \cdot y \;=\; x_0\, y_0 + x_1\, y_1 + \cdots + x_{n-1}\, y_{n-1} \;=\; \sum x_i\, y_i$$

- A **mistura** $x \oplus y$ dá um resultado em $\mathbb{B}^n$ tal que

$$z \;=\; x \oplus y \quad \Leftrightarrow \quad (\forall\, i < n)\; z_i = x_i + y_i$$

- A **máscara** $x * y$ dá um resultado em $\mathbb{B}^n$ tal que

$$z = x * y \quad \Leftrightarrow \quad (\forall\, i < n)\, z_i = x_i\, y_i$$

Considere-se de novo uma função genérica $f : \mathrm{GF}(2)^n \longrightarrow \mathrm{GF}(2)$. Como o domínio de f tem exactamente 2^n elementos, a forma interpoladora de Lagrange diz-nos que a função pode ser escrita como um polinómio em n variáveis booleanas $x_0, x_1, \ldots, x_{n-1}$ cujo grau não excede n.

A forma genérica de tal polinómio será

$$f(x_0, x_1, \ldots, x_{n-1}) = \sum_{i_0<n} \sum_{i_1<n} \cdots \sum_{i_{n-1}<n} a_{i_0,i_1,\ldots,i_{n-1}}\, x_0\, x_1\, \ldots\, x_{n-1} \tag{60}$$

Por exemplo, o seguinte polinómio descreve uma função $f : \mathrm{GF}(2)^4 \to \mathrm{GF}(2)$.

$$f(x) = 1 + x_0 + x_2 + x_1\, x_2 + x_1\, x_3 + x_0\, x_1\, x_3 \tag{61}$$

A notação em (104) é extremamente complexa e isso deriva da forma como são usados os índices. Para simplificar a notação pode-se estender a noção de índice, supondo que um índice u é um sub-conjunto qualquer de $\mathbb{Z}_n$.

Com esta convenção cada monómio de f pode ser representado por um conjunto $u \subseteq \mathbb{Z}_n$. Entendo índice como conjunto vamos usar a notação x_u para representar o monómio com as variáveis definidas por u e $\bar{x}_u$ para denotar

o monómio análogo mas com as variáveis negadas. Por exemplo,

$$x_{\{0,1,3\}} = x_0\, x_1\, x_3 \quad , \quad \bar{x}_{\{0,1,3\}} = \neg x_0\, \neg x_1\, \neg x_3 = (1+x_0)\,(1+x_1)\,(1+x_3)$$

Recorde-se que a negação $\neg x$ é calculada como $(1+x)$. Formalmente

$$x_u = \prod_{i \in u} x_i \quad , \quad \bar{x}_u = \prod_{i \in u} (1+x_i) \tag{62}$$

Esta convenção também permite escrever (104) de forma compacta. Toda a função booleana de n variáveis escreve-se

$$f = \sum_{u \in \Sigma(f)} x_u \tag{63}$$

em que $\Sigma(f)$ é um apropriado sub-conjunto de $\wp(\mathbb{Z}_n)$ designado como **espectro de índices** de f.

Para a função f em (61) o espectro de índices é

$$\Sigma(f) = \{\emptyset\ ,\ \{0\}\ ,\ \{2\}\ ,\ \{1,2\}\ ,\ \{1,3\}\ ,\ \{0,1,3\}\}$$

Também se pode ver o espectro como um subconjunto de $\mathbb{B}^n$ representando cada índice por uma *string* de n bits.

$$\Sigma(f) = \{0\,0\,0\,0\ ,\ 1\,0\,0\,0\ ,\ 0\,0\,1\,0\ ,\ 0\,1\,1\,0\ ,\ 0\,1\,0\,1\ ,\ 1\,1\,0\,1\}$$

A linguagem $\Sigma(f)$ descreve completamente a função f através da construção (105). Note-se que o número de linguagens distintas é igual ao número de conjunto que se podem formar com palavras de n bits; dado que existem 2^n palavras de n bits, o número de linguagens diferentes será 2^{2^n}.

□

A outra representação possível de f, por conjuntos de palavras de n bits, seria feita via o suporte da função. Se for $S = \text{sup}(f)$, as palavras $x \in S$ são os argumentos que f mapeia em 1: o suporte S não é mais do que o conjunto $f^{-1}(1)$.

EXEMPLO 20: A representação de f através do seu suporte é muito frequente, por exemplo, quando as funções booleanas representam certas componentes das cifras designadas por **S-boxes** ("substitution boxes").

Uma *S-box* $n \times s$ é uma função $f : \mathbb{B}^n \longrightarrow \mathbb{B}^s$. Por questões de eficiência computacional e de generalidade das funções escolhidas, é normalmente implementada numa tabela com 2^n entradas, e cada entrada usa s bits.

Consome, portanto, $2^n \times s$ bits de memória; p.ex., uma *S-box* 16×16 necessita de 128 Kbytes. Este é um valor demasiado elevado para que as *S-boxes* "grandes" possam ser usadas em dispositivos computacionalmente limitados (por exemplo, *smart-cards*). Normalmente estão limitadas a 8×8.

Porém, se for $s \ll n$, pode ser mais eficiente representar f como uma *"inverse S-Box"*: uma tabela com $2^s - 1$ entradas em que a entrada respeitante ao "output" z é uma lista de palavras de n bits representando $f^{-1}(z)$.

A lista $f^{-1}(0)$ não necessita de representação explicita; assim, se muitos "inputs" forem mapeados em 0, então a "inverse S-Box" pode ser muito eficiente porque apenas um número limitado de entradas são necessárias para cada um dos restantes "outputs".

□

Para se ver como é possível construir a função f partindo do seu suporte precisamos de analisar uma função particular.

O **símbolo de Kronecker** δ_n é a função $\mathrm{GF}(2)^n \longrightarrow \mathrm{GF}(2)$ que verifica

$$\delta_n(x) = 1 \quad \Leftrightarrow \quad x = (0, 0, \cdots, 0)$$

Tem-se $\delta_n(x) = 1$ se e só, par todo i, $x_i = 0$; consequentemente tem de se verificar

$$\delta_n(x) = (1 + x_0)\,(1 + x_1)\,(1 + x_2) \cdots (1 + x_{n-1}) \tag{64}$$

O espectro $\Sigma\,(\delta_n)$ é todo o conjunto $\mathbb{B}^n$. Este facto prova-se por indução: é válido para $\delta_1(x) = (1 + x_0)$ e, atendendo que $\delta_{i+1}(x) = \delta_i(x)\,(1 + x_i)$, se for válido para δ_i é também válido para δ_{i+1}.

A dualidade entre a representação espectral e o suporte, como representações de uma função f, é bem evidente na comparação entre duas funções: o símbolo de Kronecker $\delta_n(x)$ e a função constante 1. A função $\delta_n(x)$ tem, por

suporte, o conjunto singular $\{00\ldots0\}$ mas tem um espectro com todas as 2^n *strings* geradas com n bits. Ao invés, a função constante $f(x) = 1$ tem um espectro que é o conjunto singular $\{\emptyset\}$ mas tem $\sup(f) = \mathbb{B}^n$ (todo $x \in \mathbb{B}^n$ verifica $f(x) = 1$).

No que se segue assumimos que a dimensão n está implicitamente estabelecida e representaremos o símbolo de Kronecker simplesmente por $\delta(x)$.

99 FACTO
Para qualquer $z \in \mathbb{B}^n$ *, a função* $x \mapsto \delta(x \oplus z)$ *verifica* $\delta(x \oplus z) = 1$ *se e só se* $x = z$ *.*

Prova Tem-se $x = z$ sse $x \oplus z = 0$ e $\delta(x \oplus z) = 1$ sse $x \oplus z = 0$.

Como consequência, é possível reconstruir uma função arbitrária f partindo do seu suporte

$$f \quad = \quad \sum_{z \in \mathsf{sup}(f)} \delta(z \oplus x) \tag{65}$$

As igualdades (105) e (106) ilustram as duas formas *standard* de construir uma função booleana f partindo de um conjunto de palavras $A \subseteq \mathbb{B}^n$. Se interpretarmos A como o espectro de f, constrói-se a função como $\sum_{u \in A} x_u$. Ao invés, se A for interpretado como o suporte de f, a função é gerada por $\sum_{u \in A} \delta(u \oplus x)$.

Para determinar $\delta(x \oplus z)$ e o seu espectro, pode-se usar de novo o facto 99 e generalizar (107).

$$\delta(x \oplus z) \;=\; \prod_i (1 + x_i + z_i) \;=\; x_z\, \bar{x}_{\bar{z}} \tag{66}$$

em que $\bar{z}$ denota a palavra de bits construída com os complementos dos bits de z.

EXEMPLO 21: Construir a função $f : \mathrm{GF}(2)^4 \to \mathrm{GF}(2)$ cujo suporte é $S = \{0110, 1100\}$.

Usando (66), tem-se

$$\begin{aligned} \delta(x \oplus 0110) &= x_{0110}\, \bar{x}_{1001} = x_1\, x_2\, (1 + x_0)\, (1 + x_3) \\ \delta(x \oplus 1100) &= x_{1100}\, \bar{x}_{0011} = x_0\, x_1\, (1 + x_2)\, (1 + x_3) \end{aligned}$$

Usando (106), a função f, que verifica $S = \mathrm{sup}(f)$, é a soma destas duas funções;

$$f = \delta(x \oplus 0110) + \delta(x \oplus 1100) \;=\; x_1\, x_2 + x_1\, x_2\, x_3 + x_0\, x_1 + x_0\, x_1\, x_3$$

Esta função é naturalmente representada pelo seu espectro $\Sigma(f) \;=\; \{\, 0110, 0111, 1100, 1101 \,\}$.

□

Pode parecer que o modelo das funções booleanas é particularmente limitado e que aplicações realistas (por exemplo, cifras) não são contemplados. No entanto consegue capturar muitas das propriedades de funções mais complexas.

Suponhamos que se tem um "bloco" de $n \times n$ bits, $B\colon \mathrm{GF}(2)^n \longrightarrow \mathrm{GF}(2)^n$, que se observou um "output" z e se quer encontrar um "input" x correspondente. O problema é

$$\textit{conhecidos } B \textit{ e } z\textit{, encontrar } x \textit{ tal que } B(x) = z$$

Dado que x e z são palavras de n bits, parece que as funções booleanas estão arredadas deste problema.

Mero engano !

Pode-se escrever a igualdade $B(x) = z$ como $B(x) \oplus z = 0$ e, esta última, como $\delta(B(x) \oplus z) = 1$.

Dado que B e z são conhecidos, $f(x) = \delta(B(x) \oplus z)$ é uma função booleana de x. Determinar o suporte de f é equivalente, por isso, a encontrar os valores x que verificam a equação $B(x) = z$.

□

Determinar o suporte de uma função booleana é um problema básico. Se a função f for especificada pelo seu espectro (ou por outra forma computacionalmente equivalente), a relação entre os dois conjuntos $\Sigma(f)$ e $\mathrm{sup}(f)$ é a questão computacional essencial da teoria das funções booleanas.

Também se pode colocar o problema de forma inversa: a função é especificada pelo seu suporte e pretende-se determinar o seu espectro. Normalmente o segundo problema é uma componente do primeiro: antes de determinar

o suporte de f é preciso determinar, em primeiro lugar, o seu espectro. Se f for constituído por componentes especificadas pelo respectivo suporte, pode-se determinar o espectro das componentes e combinar esses espectros parciais para formar o espectro final de f. Com esse espectro pode-se, finalmente, determinar o suporte de f.

Na maior ou menor dificuldade em resolver estes problemas, reside a noção da incerteza de f e, para a exprimir, os circuitos que vimos na secção anterior são uma ferramenta útil.

□

Suportes $\sup(f)$ e espectros $\Sigma(f)$ são linguagens finitas e livres de prefixos, com a particularidade de todos os seus elementos terem comprimento igual ao número de argumentos de f. Por isso, naturalmente, usando a função Λ^{-1} (ver (58), na página 182) ambas as linguagens são representáveis por circuitos.

Deste modo a conversão de suportes em espectros (e vice-versa) podem ser descrita por um morfismo entre circuitos. Para construir estes morfismos precisamos de duas construções recursivas adicionais. No que se segue designaremos por $\mathcal{C}$ o domínio dos circuitos ou, equivalentemente, o domínio das linguagens finitas livres de prefixos.

Soma: Dadas linguagens finitas e livres de prefixos de comprimento fixo, $A, B \subseteq \mathbb{B}^n$, define-se *soma*[26]

$$A + B \;=\; (A \cup B) \setminus (A \cap B) \tag{67}$$

[26] Na terminologia da Teoria dos Conjuntos, a soma de linguagens é a *diferença simétrica* de conjuntos, usualmente representada pelo símbolo $\ominus$; por isso, aparentemente, não se justifica uma terminologia e notação alternativas. No entanto, dada a relação próxima entre a diferença simétrica de linguagens finitas, livres de prefixos e a soma em $\mathrm{GF}(2)$, adoptamos a notação e terminologia aqui apresentadas.

A soma recolhe as bit-strings de ambas as linguagens com excepção dos elementos comuns.

EXEMPLO 22:
Se $A = \{00, 01, 11\}$ e $B = \{10, 11\}$, então $A + B = \{00, 01, 10\}$. O elemento comum 11 é "cancelado".

Uma definição equivalente seria $A + B = (A \setminus B) \cup (B \setminus A)$. É evidente que, para toda $A \subseteq \mathbb{B}^n$, tem-se $A + A = \emptyset$ e $A + \emptyset = A$. Por isso a soma $+$ e a linguagem $\emptyset$ definem em $\wp(\mathbb{B}^n)$ a estrutura de um grupo abeliano.

No domínio $\mathcal{C}$ das linguagens finitas e livres de prefixos arbitrárias, a definição da soma é um pouco mais complexa já que é definida a menos de uma relação de equivalência.

Recorde-se que, para $A \in \mathcal{C}$, $\uparrow A$ representa a classe de todos os conjuntos (ou, bit-strings infinitas) que têm um elemento de A como prefixo. Duas linguagens finitas e livres de prefixos $A, B \in \mathcal{C}$ são **equivalentes**, e escreve-se $A \cong B$, quando $\uparrow A = \uparrow B$. É simples verificar que esta equivalência ocorre se e só se as linguagens são descritas por circuitos equivalentes.

Define-se **soma** de linguagens finitas livres de prefixos, $A + B$, como a classe de todas as linguagens S que verificam $\uparrow S = (\uparrow A \cup \uparrow B) \setminus (\uparrow A \cap \uparrow B)$.

Dualidade: O dual A' de uma linguagem finita e livre de prefixos A, é a linguagem construída recursivamente por

$$\emptyset' = \emptyset \quad , \qquad \{\varepsilon\}' = \{\varepsilon\} \quad , \qquad (A \uplus B)' = A' \uplus (A' + B') \tag{68}$$

EXEMPLO 23:
Considere-se a linguagem $S = \{0110, 1100\}$ apresentada no exemplo 21 como o suporte de uma função booleana f. Vamos determinar o dual S' e compará-lo com o espectro da função, calculado no mesmo exemplo.

Tem-se $S = \{110\} \uplus \{100\} = (\emptyset \uplus \emptyset \uplus (\{\varepsilon\} \uplus \emptyset)) \uplus (\emptyset \uplus ((\{\varepsilon\} \uplus \emptyset) \uplus \emptyset))$. Aplicando recursivamente (68) a esta expansão de S obtém-se $S' = \{0110, 0111, 1100, 1101\}$ que coincide, precisamente, com o espectro de f.

Este resultado não é coincidência; de facto, para qualquer função booleana, o espectro e o suporte estão relacionados pela dualidade.

Para demonstrar esta importante relação entre as duas representações de funções booleanas, precisamos de alguns resultados prévios.

100 LEMA *Se, para algum n, se tem $A, B \subseteq \mathbb{B}^n$, então verifica-se:*

- $(A + B)' = A' + B'$
- $(A')' = A$.

Esboço de prova A prova é uma simples aplicação da indução na estrutura das linguagens livres de prefixos e finitas e da definição de dualidade.

O lema anterior não é válido, normalmente, quando A, B são linguagens que, mesmo sendo finitas e livres de prefixos, têm elementos de comprimentos diferentes. Aplica-se, no entanto, aos casos que nos interessam: espectros e suportes são linguagens formadas por *strings* todas com o mesmo comprimento. Para linguagens finitas e livres de prefixos arbitrárias $A, B \in \mathcal{C}$, este resultado estende-se usando a equivalência de linguagens

101 LEMA *Se A é uma linguagem finita e livre de prefixos, então $(A')' \cong A$ e, para qualquer outra linguagem $B \in \mathcal{C}$, verifica-se $(A + B)' \cong A' + B'$.*

As definições de espectro e de suporte de função booleana, conduzem imediatamente ao seguinte resultado.

102 LEMA *Se f, g são funções booleanas, então*

- $\Sigma(f + g) = \Sigma(f) + \Sigma(g)$ *e*
- $\mathsf{sup}(f + g) = \mathsf{sup}(f) + \mathsf{sup}(g)$.

Finalmente, o resultado que nos interessa é uma representação estrutural do algoritmo de Davis-Putman para o problema de SAT.

103 TEOREMA
Para toda a função booleana tem-se $\Sigma(f)' = \mathsf{sup}(f)$.

Esboço de Prova Pode-se tomar qualquer polinómio $f(x_0, x_1, \ldots, x_{n-1}$ e escrevê-lo como

$$f \;=\; f_0(x_1, \ldots, x_{n-1}) \;+\; x_0\, f_1(x_1, \ldots, x_{n-1})$$

Os polinómios f_0 e f_1 não contêm a variável x_0 em nenhum dos seus monómios. Pela definição de espectro temos $\Sigma(f) = \Sigma(f_0) \uplus \Sigma(f_1)$. Por simples manipulação, tem-se $f \;=\; (1+x_0)\, f_0 + x_0\, (f_0+f_1)$ donde resulta, pelo lema 102, $\sup(f) \;=\; \sup(f_0) \uplus \sup(f_0+f_1)$.

Este teorema dá-nos uma formalização simples da relação fundamental entre espectros e suportes de funções booleanas. Ambas as linguagens podem ser representadas por circuitos e, por isso, os problemas essenciais das funções booleanas podem ser expressos em termos das operações com circuitos que vimos na página 185.

1 É conhecido o suporte de f

Seja σ o circuito que representa o suporte de f; isto é $\sigma \;=\; \Lambda^{-1}(\sup(f))$. Então

1. **Cálculo de $f(x)$**
   Tem-se $f(x) \;=\; \sigma/x$. A complexidade deste problema é a do problema da aceitação; isto é, $O(n)$.
2. **Encontrar x tal que $f(x) = 1$**
   O suporte não é vazio sse $\sigma \geqslant 1$; equivalentemente, x verifica $f(x) = 1$ sse $\sigma/x = 1$. A complexidade deste problema é a do problema da verificação, i.e. limitada a $O(2^n)$.

2 É conhecido o espectro de f

Seja $\tau = \Lambda^{-1}(\Sigma(f))$ o circuito que representa o espectro de f. O problema básico, aqui, é a determinação do suporte de f. Este problema tem, essencialmente, a complexidade do problema da cópia.

3.3 Funções Boolenas de argumento GF(2^n)

SBoxes $n \times n$ são representadas por funções onde o domínio e contradomínio podem ser $\mathrm{GF}(2)^n$ ou $\mathrm{GF}(2^n)$. Nomeadamente é importante relacionar representações usando funções $\mathrm{GF}(2)^n \longrightarrow \mathrm{GF}(2)^n$ com as representações que usam as funções $\mathrm{GF}(2^n) \longrightarrow \mathrm{GF}(2^n)$.

Genericamente, em qualquer corpo finito $\mathbb{F}_q$ (nomeadamente no corpo $\mathrm{GF}(2^n)$), qualquer função $f : \mathbb{F}_q \longrightarrow \mathbb{F}_q$ é sempre descrita por um polinómio de grau inferior a $q-1$.

$$f(x) \;=\; \sum_{i=0}^{q-1} a_i\, x^i \;, \qquad a_i \in \mathbb{F}_q \tag{69}$$

Isto porque a fórmula interpoladora de Lagrange diz que quaisquer n pontos de uma função determinam um polinómio de grau $n-1$ que passa por esses N pontos. No caso particular em que N é igual à cardinalidade do domínio da função (i.e. quando $n = q$) o polinómio passa por todos os pontos da função.

Outras representações equivalentes são usadas baseadas no facto de que, em $\mathbb{F}_q$, todo $x \neq 0$ verifica $x^{q-1} = 1$. Consequentemente, se $n = m \pmod{q-1}$, tem-se $x^n = x^m$. Esta relação permite substituir monómios x^n por outros monómios x^m (desde que seja $n = m \pmod{q-1}$) e assim gerar outras representações de f.

EXEMPLO 24: Na cifra AES as operações sobre *bytes* são descritas por funções $f : \mathrm{GF}(2^8) \longrightarrow \mathrm{GF}(2^8)$. O corpo de Galois é representado numa base polinomial do tipo 0 usando o polinómio característico $\mathbf{c} = y^8+y^4+y^3+y+1$.

A definição do **AES** especifica uma função `ByteSub` que é a composição de duas funções $\text{ByteSub} = \text{Inv} \circ \text{Afim}$.

A primeira das quais é a função $\text{Inv}(x) = 2^{254}$. Note-se que, em $\text{GF}(2^8)$, todo $x \neq 0$ verifica $x^{255} = 1$; logo $x^{254} = x^{-1}$. Portanto tem-se $\text{Inv}(0) = 0$ e, para $x \neq 0$, $\text{Inv}(x) = x^{-1}$.

Afim é uma função afim definida, não sobre $\text{GF}(2^8)$, mas sim sobre sobre $\text{GF}(2)^8$; tem a forma[27]

$$\text{Afim}(x) = c \oplus (\omega_0(x), \cdots, \omega_{n-1}(x)) \quad \text{com} \quad c, \omega_i \in \text{GF}(2)^8$$

Como as duas componentes de ByteSub são definidas por funções em domínios distintos não é imediato ter uma representação algébrica para ByteSub. A única forma de obter uma representação da composição das duas componentes passa por transformar uma das representações para o domínio da outra.

□

O exemplo anterior ilustra como é importante converter uma representação linear ou afim genérica em $\text{GF}(2)^n$ numa função de domínio $\text{GF}(2^n)$. É o que faremos em seguida começando por alguns casos de funções de domínio $\text{GF}(2^n)$ que merecem referência especial:

1. **automorfismos em $\text{GF}(2^n)$**

[27] Os valores concretos das constantes c, ω_i constam na especificação oficial do AES.

Os automorfismos em $\mathrm{GF}(2^n)$, i.e. funções injectivas que preservam a estrutura do corpo, fixam necessariamente os elementos de $\mathrm{GF}(2)$. Vendo $\mathrm{GF}(2^n)$ como uma extensão de $\mathrm{GF}(2)$, o facto 86 determina que f tem a forma $\sigma^k : x \mapsto x^{2^k}$, para algum $k \in 0..n-1$. Só as potências do morfismo de Frobenius $\sigma : x \mapsto x^2$ são automorfismos neste corpo.

É possível escrever os σ^k de uma forma vectorial de modo a permitir o uso de uma álgebra matricial para representar transformações lineares.

104 NOÇÃO

A função $(\cdot)_\sigma : \mathrm{GF}(2^n) \to \mathrm{GF}(2^n)^n$ *transforma um elemento* x *do corpo de Galois no vector, sobre esse corpo, formado por todos* $\sigma^k(x)$ *(as imagens de* x *pelos vários automorfismos).*

$$x_\sigma \doteq (x\ ,\ \sigma(x)\ ,\ \sigma^2(x)\ ,\ \cdots\ ,\ \sigma^{n-1}(x)) \tag{70}$$

2. **funções booleanas $\mathrm{GF}(2^n) \to \mathrm{GF}(2)$**

Estas funções podem ser representadas pelas formas genéricas em (69) tendo em atenção que o contradomínio $\mathrm{GF}(2)$ está imerso em $\mathrm{GF}(2^n)$.

Por exemplo, a função *traço* (ver definição 87 – página 145), é definida pelo polinómio

$$\mathrm{tr}(x) = x + x^2 + x^4 + \cdots + x^{2^{n-1}}$$

3. funções lineares

As funções lineares $f : \mathrm{GF}(2^n) \longrightarrow \mathcal{K}$, sendo $\mathcal{K}$ uma qualquer extensão de $\mathrm{GF}(2)$, são funções que preservam somas e multiplicações por escalares: para todos $x, y \in \mathrm{GF}(2^n)$ e $a \in \mathrm{GF}(2)$

$$f(x+y) = f(x) + f(y) \quad , \quad f(a\,x) = a\,f(x)$$

Toda a função linear f pode ser escrita como um polinómio da forma

$$f(x) \;=\; \sum_{k=0}^{n-1} a_k\,\sigma^k(x) \qquad \text{com} \quad a_k \in \mathcal{K} \tag{71}$$

Representando por $\mathbf{a}$ o vector $(a_0, a_1, \cdots, a_{n-1})$, o polinómio (71) pode ser escrito como o produto escalar[28]

$$f(x) \;=\; \mathbf{a} \cdot x_\sigma \tag{72}$$

Como caso particular temos construções da forma $\mathrm{tr}(x \cdot y)$; pela definição verifica-se que, $\mathrm{tr}(x\,y) \;=\; x_\sigma \cdot y_\sigma$. Voltando à representação em (72), seja $y = f(x)$ com $f(x) = \mathbf{a} \cdot x_\sigma$. Então tem-se $y_\sigma \;=\; \mathbf{A}\,x_\sigma$ sendo

[28] Se $u, v \in X^n$ o produto escalar $u \cdot v$ é $u^t\,v = v^t\,u$, em que $(\cdot)^t$ representa a transposição de matrizes.

$\mathbf{A}$ a matriz cujo elemento genérico é

$$\mathbf{A}_{ij} = (a_k)^{2^i} \qquad \text{com} \quad k = j - i \pmod{n-1} \tag{73}$$

4. **funções bilineares**
São funções de dois argumentos $f : \mathrm{GF}(2^n) \times \mathrm{GF}(2^n) \longrightarrow \mathrm{GF}(2^n)$ lineares em cada um deles: $f(x, y+z) = f(x, y) + f(x, z)$, $f(x+z, y) = f(x, y) + f(z, y)$ e $f(x, a \cdot y) = a \cdot f(x, y) = f(a \cdot x, y)$. A forma mais geral das funções bilineares é

$$f(x) \;=\; x_\sigma \cdot (\mathbf{A}\, y_\sigma) \tag{74}$$

sendo $\mathbf{A} \in \mathrm{GF}(2^n)^{n \times n}$ uma matriz com elementos em $\mathrm{GF}(2^n)$.
Exemplos de funções bilineares de x, y em $\mathrm{GF}(2^n)$, com $n > 3$

$$x \cdot y \quad , \quad x^2 \cdot y^4 + x^4 \cdot y^2 + x^8 \cdot y^8$$

5. **funções quadráticas**
São funções $g : \mathrm{GF}(2^n) \longrightarrow \mathrm{GF}(2^n)$ da forma $g(x) = f(x, x)$ sendo f bilinear. Isto é,

$$g(x) \;=\; x_\sigma \cdot \mathbf{A}\, x_\sigma \tag{75}$$

para uma matriz apropriada $\mathbf{A} \in \mathrm{GF}(2^n)^{n\times n}$.

A função $g(x) = x^3$ é quadrática uma vez que se pode escrever $g(x) = f(x,x)$ com $f(x,y) = x\,y^2$ que, claramente, é bilinear. Pelo mesmo motivo qualquer monómio x^k é uma forma quadrática se for, para i,j apropriados, $k = 2^i + 2^j \pmod{2^n - 1}$.

6. Uma base $\mathcal{B}$ de $\mathrm{GF}(2^n)$ determina sempre n **projecções**; i.e., funções booleanas $p_\mu : \mathrm{GF}(2^n) \to \mathrm{GF}(2)$, uma para cada elemento $\mu \in \mathcal{B}$, de tal forma que qualquer $x \in \mathrm{GF}(2^n)$ pode ser reconstruído a partir dos elementos μ e dos valores $p_\mu(x)$

$$x = \sum_{\mu\in\mathcal{B}} p_\mu(x)\,\mu \tag{76}$$

Representemos por $(\cdot)^{\tilde{}} : \mathrm{GF}(2^n) \to \mathrm{GF}(2)^n$ a função que associa x ao vector das suas projecções. Vectorialmente, (76) é

$$x = \mathcal{B}\cdot x^{\tilde{}} \tag{77}$$

Algumas propriedades das projecções que derivam directamente de (76)

(i) Como a representação de x em $\mathcal{B}$ é única, as projecções são também únicas.

(ii) A projecção em $\mu \in \mathcal{B}$ de um outro elemento da base $\upsilon \in \mathcal{B}$ tem de ser zero porque, por definição de base, não é possível representar υ como uma soma de outros elementos da mesma base. Por isso

$$p_\mu(\upsilon) \;=\; \delta(\mu + \upsilon) \qquad \forall\, \mu, \upsilon \in \mathcal{B}$$

(iii) As projecções p_μ são sempre funções lineares: para todos $x, y \in \mathsf{GF}(2^n)$ e $a \in \mathsf{GF}(2)$.

$$p_\mu(0) = 0 \quad , \quad p_\mu(x+y) = p_\mu(x) + p_\mu(y) \quad , \quad p_\mu(a\,x) = a\,p_\mu(x)$$

(iv) Para todo $x \in \mathsf{GF}(2^n)$ existe y tal que $p_\mu(x) \neq p_\mu(y)$; isto é, as projecções são funções sobrejectivas.

Se compararmos as duas últimas propriedades com as da função traço (página 145) vemos que elas coincidem; por isso é natural que existam semelhanças entre as duas noções. De facto é relativamente simples provar

105 FACTO

Para qualquer elemento $\mu \in \mathcal{B}$ de uma base de $\mathsf{GF}(2^n)$ existe um único elemento $\mu' \in \mathsf{GF}(2^n)$, designado por **elemento dual** *de μ, tal que, para todo $x \in \mathsf{GF}(2^n)$,*

$$p_\mu(x) \;=\; \mathsf{tr}(\mu'\,x) \;=\; (\mu')_\sigma \cdot x_\sigma$$

O conjunto $\mathcal{B}' \doteq \{\, \mu' \mid \mu \in \mathcal{B} \,\}$ de todos os elementos duais forma uma nova base de $\mathsf{GF}(2^n)$ que será designada por **base dual** *de $\mathcal{B}$.*

Sugestão de prova Construa-se a $n \times n$-matriz de traços $\mathbf{T}_{\mu\upsilon} \doteq \mathrm{tr}(\mu\,\upsilon)$; as colunas da sua inversa $\mathbf{T}^{-1}$ determinam os elementos da base dual.

□

Ordenando os elementos de $\mathcal{B}$ num vector, e preservando a mesma ordem na base dual $\mathcal{B}'$, a prova do facto anterior mostra que estes vectores estão relacionados por

$$\mathcal{B}' \;=\; \mathbf{T}^{-1}\,\mathcal{B} \tag{78}$$

em que $\mathbf{T}$ designa a matriz dos traços cruzados: $\mathbf{T}_{\mu\upsilon} = \mathrm{tr}(\mu\,\upsilon)$.

EXEMPLO 25: Consideremos de novo $\mathrm{GF}(2^4)$ com uma base polinomial do tipo 0

$$\mathcal{B} = \left\{1\ ,\ \beta\ ,\ \beta^2\ ,\ \beta^3\right\}$$

Para calcular a matriz dos traços $\mathbf{T}_{ij} \doteq \mathrm{tr}(\beta^i \cdot \beta^j) = \mathrm{tr}(\beta^{i+j})$ basta calcular a lista dos traços das potências de β para expoentes entre 0 e 6.

Usando o polinómio característico $\mathbf{c}(X) = X^4 + X + 1$ facilmente se constrói a tabela

i	0	1	2	3	4	5	6
$\mathsf{tr}(\beta^i)$	0	0	0	1	0	0	1

A matriz $\mathbf{T}$ e a sua inversa $\mathbf{T}^{-1}$ são

$$\mathbf{T} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{T}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Portanto a base dual é formada pelos elementos (pela ordem dos respectivos duais em $\mathcal{B}$)

$$\mathcal{B}' = \left\{1 + \beta^3 \ , \ \beta^2 \ , \ \beta \ , \ 1\right\}$$

e as projecções respectivas serão

$$p_1(x) = \mathsf{tr}(x) + \mathsf{tr}(x\,\beta^3) \qquad , \qquad p_\beta(x) = \mathsf{tr}(x\,\beta^2)$$

$$p_{\beta 2}(x) = \mathsf{tr}(x\,\beta) \qquad , \qquad p_{\beta 3}(x) = \mathsf{tr}(x)$$

o que permite concluir a identidade

$$x = \mathsf{tr}(x) + \mathsf{tr}(x\,\beta^3) + \mathsf{tr}(x\,\beta^2)\,\beta + \mathsf{tr}(x\,\beta)\,\beta^2 + \mathsf{tr}(x)\,\beta^3$$

□

A noção de elemento dual pode ser estendida a qualquer $x \in \mathrm{GF}(2^n)$. O **elemento dual** de x na base $\mathcal{B}$, representado por x', é elemento de $\mathrm{GF}(2^n)$ que tem exactamente as mesmas componentes de x mas na base dual $\mathcal{B}'$; i.e. para todo $\mu \in \mathcal{B}$

$$p_\mu(x) = \mathrm{tr}(\mu' x) = \mathrm{tr}(\mu x') = p_{\mu'}(x') \tag{79}$$

Tomando como implícita a base $\mathcal{B}$, o operador $(\cdot)^\sim$ associa cada elemento de $\mathrm{GF}(2^n)$ ao vector das suas projecções nessa base. Então verifica-se

$$x'^\sim = \mathbf{T}^{-1} x^\sim \quad , \quad x' = \mathcal{B} \cdot x'^\sim = \mathcal{B}' \cdot x^\sim \tag{80}$$

□

O operador $(\cdot)_\sigma$ pode ser extendido para vectores de elementos $\mathrm{GF}(2^n)$. Dado $A \in \mathrm{GF}(2^n)^n$ a matriz $A_\sigma \in \mathrm{GF}(2^n)^{n\times n}$ tem, por linha de ordem i, o vector $(A_i)_\sigma$; isto é $(A_\sigma)_{ik} = \sigma^k(A_i)$.

Nestas circunstâncias

106 FACTO

Se $\mathcal{B}, \mathcal{B}'$ são um par de bases duais então:

1. $(\mathcal{B}')_\sigma = \mathbf{T}^{-1}\,\mathcal{B}_\sigma$
2. $\tilde{x} = (\mathcal{B}')_\sigma\, x_\sigma \quad e \quad x_\sigma = (\mathcal{B}_\sigma)^t\, \tilde{x}$
3. $(\mathcal{B}_\sigma)^t\, \mathcal{B}_\sigma = \mathbf{T} \quad e \quad (\mathcal{B}_\sigma)^t\, (\mathcal{B}')_\sigma = \mathbf{I}.$
4. $\mathcal{B}_\sigma\, (x')_\sigma = (\mathcal{B}')_\sigma\, x_\sigma.$

Prova O primeiro resultado é consequência da relação $\mathcal{B}' = \mathbf{T}^{-1}\,\mathcal{B}$, da linearidade dos morfismos σ^k e do facto de fixarem todos os elementos da matriz $\mathbf{T}^{-1}$.

O segundo resultado é consequência do anterior e do facto 105. Os últimos resultados são as definições da matriz dos traços cruzados $\mathbf{T}$, da base dual e elemento dual.

□

Todas estas noções são essenciais quando se pretende estudar funções booleanas que requerem "mudança de representação"; isto ocorre quando uma função é formada pela composição de uma sequência de funções que manipulam as palavras de *bits* alternadamente na representação $\mathrm{GF}(2^n)$ e na representação $\mathrm{GF}(2)^n$.

É o caso da SBox do AES (introduzida na exemplo 24) que é determinada pela composição da função $x \mapsto x^{-1}$ em $\mathrm{GF}(2^8)$ seguida de uma transformação afim em $\mathrm{GF}(2)^8$.

O problema essencial é o seguinte:

Dada uma base de representação $\mathcal{B}$ *em* $\mathsf{GF}(2^n)$, *dada uma função* $f^{\tilde{}}$ *de domínio* $\mathsf{GF}(2)^n$, *construir a função* f *de domínio* $\mathsf{GF}(2^n)$ *que verifica* $f^{\tilde{}}(x^{\tilde{}}) = f(x)^{\tilde{}}$ *para todo* x.
Ou, inversamente, dada a função f, *construir* $f^{\tilde{}}$.

Note-se que: $f^{\tilde{}}(x^{\tilde{}})$ denota a função de palavras de bits $f^{\tilde{}}$ aplicada ao vector de bits que representa x (visto como elemento do corpo de Galois); $f(x)^{\tilde{}}$ é o vector de bits que representa o resultado $f(x)$; a relação entre as duas funções diz-nos que estes dois vectores são iguais.

Vamos procurar resolver este problema para algumas formas particulares de funções $f^{\tilde{}} : \mathsf{GF}(2)^n \to \mathsf{GF}(2)^n$ ou $f^{\tilde{}} : \mathsf{GF}(2)^n \to \mathsf{GF}(2)$.

$f^{\tilde{}}(x^{\tilde{}}) = x^{\tilde{}} \oplus c^{\tilde{}}$ com $c \in \mathsf{GF}(2^n)$.

Esta é a função *branqueamento* ("whitening") que ocorre quase sempre nos andares das cifras. A função de domínio $\mathsf{GF}(2^n)$ equivalente é

$$f(x) \;=\; x + c$$

$f^{\tilde{}}(x^{\tilde{}}) = c^{\tilde{}}(x^{\tilde{}}) = \mathsf{Tr}(c^{\tilde{}} * x^{\tilde{}})$ com $c \in \mathsf{GF}(2^n)$

Forma genérica da função booleana linear; a função de domínio $\mathsf{GF}(2^n)$ equivalente é

$$f(x) \;=\; \mathsf{tr}(c'\,x) \;=\; (c')_\sigma \cdot x_\sigma$$

Prova: Simples utilização das identidades no facto 106

$$f^{\tilde{}}(x^{\tilde{}}) = c^{\tilde{}} \cdot x^{\tilde{}} = \quad (\mathcal{B}_\sigma \, (c')_\sigma) \cdot (\mathcal{B}')_\sigma \, x_\sigma = (((\mathcal{B}')_\sigma)^t \mathcal{B}_\sigma \, (c')_\sigma) \cdot x_\sigma = (c')_\sigma \cdot x_\sigma$$

$f^{\tilde{}}(x^{\tilde{}}) = \mathbf{H}\, x^{\tilde{}}$ com $\mathbf{H} \in \mathrm{GF}(2)^{n \times n}$.

Esta é a forma genérica da SBox linear onde cada *bit* à saída é uma combinação linear dos *bits* de entrada.
A função de domínio $\mathrm{GF}(2^n)$ equivalente é o polinómio

$$f(x) = \quad \mathbf{h} \cdot x_\sigma \tag{81}$$

em que $\mathbf{h} = (\mathcal{B}')_\sigma^t \, \mathbf{H}^t \, \mathcal{B}$.

Prova

$$f(x) = \mathcal{B} \cdot f^{\tilde{}}(x^{\tilde{}}) = \quad \mathcal{B} \cdot (\mathbf{H} \, (\mathcal{B}')_\sigma \, x_\sigma) = ((\mathcal{B}')_\sigma^t \cdot \mathbf{H}^t \, \mathcal{B}) \cdot x_\sigma$$

É importante ter-se em conta que (81), apesar da forma aparentemente complexa, é uma função linear. porque tem a forma prevista em (71).
Pode-se interpretar $\mathbf{H}^t \, \mathcal{B}$ como o vector determinado pelas colunas de $\mathbf{H}$ vendo cada coluna como os coeficientes de um elemento de $\mathrm{GF}(2^n)$ na base $\mathcal{B}$.

Este vector (e, consequentemente, a matriz $\mathbf{H}$) pode ser recuperado de $\mathbf{h}$ por

$$\mathbf{H}^t\,\mathcal{B} \;=\; \mathcal{B}_\sigma\,\mathbf{h}$$

Isto permite fazer a conversão em sentido inverso: dada a função linear de domínio GF(2^n), obter a matriz que determina a função equivalente de domínio GF$(2)^n$.

A relação entre $\mathbf{H}$ e $\mathbf{h}$ pode ainda ser vista de outra forma; note-se que se podia escrever

$$\mathbf{h}\cdot x_\sigma = \;(\mathbf{H}^t\,\mathcal{B})\cdot(\mathcal{B}')_\sigma\,x_\sigma = (\mathbf{H}^t\,\mathcal{B})\cdot x^{\tilde{}}$$

Finalmente pode-se ver

$$\mathbf{h} \;=\; (\mathbf{H}\,\mathcal{B}')^t_\sigma\,\mathcal{B}$$

$\mathbf{H}\,\mathcal{B}'$ representa ver as linhas da matriz como vectores representados na base $\mathcal{B}'$; isto é, se ω_i representar o elemento de GF(2^n) representado pela linha de ordem i da matriz **H**, então $\mathbf{H}\,\mathcal{B}' = (\omega'_0,\cdots,\omega'_{n-1})$ é o vector dos elementos duais.

EXEMPLO 26: Regressando à cifra AES e ao exemplo 24, a transformação afim $g^{\tilde{}}(y^{\tilde{}}) = \mathbf{b}\oplus\mathbf{H}\,y^{\tilde{}}$ é definida,

na especificação da cifra, por

$$\mathbf{b} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \qquad \mathbf{H} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

ou, representando cada *byte* em base hexadecimal e a matriz como um vector de colunas,

$$\mathbf{b} = 63 \qquad \mathbf{H} = (\texttt{8F} \ , \ \texttt{C7} \ , \ \texttt{E3} \ , \ \texttt{F1} \ , \ \texttt{F8} \ , \ \texttt{7C} \ , \ \texttt{3E} \ , \ \texttt{1F})$$

A especificação do AES define o polinómio característico

$$\mathbf{c}[X] = 1 + X + X^3 + X^4 + X^8$$

Para uma base polinomial do tipo 0 gerada por este polinómio, e usando a mesma metodologia de representação,

temos a matriz dos traços cruzados, a sua inversa e $\mathcal{B}_\sigma$

$$\mathbf{T} = (05\ ,\ 0A\ ,\ 15\ ,\ 2B\ ,\ 57\ ,\ AE\ ,\ 5C\ ,\ B9)$$

$$\mathbf{T}^{-1} = (94\ ,\ 0D\ ,\ 1A\ ,\ A0\ ,\ 65\ ,\ CA\ ,\ 25\ ,\ 2A)$$

$$\mathcal{B}_\sigma = \begin{bmatrix} 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 \\ 02 & 04 & 10 & 1B & 5E & E4 & 4D & FA \\ 04 & 10 & 1B & 5E & E4 & 4D & FA & 02 \\ 08 & 40 & AB & B3 & E8 & 1D & 4A & EF \\ 10 & 1B & 5E & E4 & 4D & FA & 02 & 04 \\ 20 & 6C & 97 & 94 & 91 & 80 & 9A & C5 \\ 40 & AB & B3 & E8 & 1D & 4A & EF & 08 \\ 80 & 9A & C5 & 20 & 6C & 97 & 94 & 91 \end{bmatrix}$$

Os elementos de $\mathcal{B}_\sigma$ são polinómios; nesta representação são apresentados os seus coeficientes como vectores de bits, começando no grau mais elevado. Por exemplo $E4 = 11100100$ denota o polinómio $X^7 + X^6 + X^5 + X^2$.

Com estas três matrizes é possível computar todos os elementos necessários; por exemplo,

$$(\mathcal{B}')_\sigma = \mathbf{T}^{-1}\, \mathcal{B}_\sigma$$

que é a componente essencial para calcular $\mathbf{h}$ a partir de $\mathbf{H}$.

Feitas as contas conclui-se

$$\mathbf{h} = (\texttt{05}\ ,\ \texttt{09}\ ,\ \texttt{F9}\ ,\ \texttt{25}\ ,\ \texttt{F4}\ ,\ \texttt{01}\ ,\ \texttt{B5}\ ,\ \texttt{8F})$$

Conclui-se portanto que a transformação afim do AES é

$$\begin{aligned} g(y) = \quad & \texttt{63} + \texttt{05} \cdot y + \texttt{09} \cdot y^2 + \texttt{F9} \cdot y^4 + \texttt{25} \cdot y^8 + \\ & + \texttt{F4} \cdot y^{16} + \texttt{01} \cdot y^{32} + \texttt{B5} \cdot y^{64} + \texttt{8F} \cdot y^{128} \end{aligned}$$

A SBox do AES resulta da composição dessa função ao resultado da transformação

$$x \mapsto x^{254}$$

que mapeia 0 em si próprio e qualquer $x \neq 0$ em x^{-1}.

A transformação final abtém-se então substituindo, em $g(y)$, a variável y por x^{254} e reduzindo os expoentes módulo 255 (uma vez que $x^{255} = 1$ se $x \neq 0$).

Por exemplo

$$y^{128} \quad \mapsto \quad (x^{254})^{128} \quad \mapsto \quad x^{(254*128 \mod 255)} \quad \mapsto \quad x^{127}$$

Genericamente y^k reduz-se a x^{255-k}. O resultado final é

$$f(x) = \begin{array}{l} \texttt{63} + \texttt{05} \cdot x^{254} + \texttt{09} \cdot x^{253} + \texttt{F9} \cdot x^{251} + \texttt{25} \cdot x^{247} + \\ + \texttt{F4} \cdot x^{239} + \texttt{01} \cdot x^{223} + \texttt{B5} \cdot x^{191} + \texttt{8F} \cdot x^{127} \end{array} \tag{82}$$

□

Finalmente vamos examinar a representação das **funções bilineares**

$$f(x, y) = x_\sigma \cdot \mathbf{A}\, y_\sigma \tag{83}$$

ou equivalentemente, com $\boldsymbol{\alpha} \doteq \mathcal{B}_\sigma\, \mathbf{A}\, \mathcal{B}_\sigma^t$

$$f(x, y) = x^{\tilde{}} \cdot \boldsymbol{\alpha}\, y^{\tilde{}} \tag{84}$$

Usando as identidades $x_\sigma = \mathcal{B}_\sigma^t\, x^{\tilde{}}$ e $\boldsymbol{\alpha} = \mathcal{B}_\sigma\, \mathbf{A}\, \mathcal{B}_\sigma^t$ tem-se

$$f(x, y) = (\mathcal{B}_\sigma^t\, x^{\tilde{}}) \cdot \mathbf{A}\, \mathcal{B}_\sigma^t\, y^{\tilde{}} = x^{\tilde{}} \cdot \mathcal{B}_\sigma\, \mathbf{A}\, \mathcal{B}_\sigma^t\, y^{\tilde{}} = x^{\tilde{}} \cdot \boldsymbol{\alpha}\, y^{\tilde{}}$$

Seja $\boldsymbol{\alpha}_k$ a matriz $\mathrm{GF}(2)^{n\times n}$ que selecciona a componente k de cada um dos elementos de $\boldsymbol{\alpha}$; seja z_k a componente correspondente de $f(x,y)$; então

$$z_k = x^{\tilde{}} \cdot \boldsymbol{\alpha}_k \, y^{\tilde{}}$$

Desta forma é possível calcular as componentes individuais de $f(x,y)$; expandindo resulta uma função booleana com monómios da forma $x_i \, y_j$.

$$z_k = \sum_{ij} \boldsymbol{\alpha}_k^{ij} \, x_i \, y_j$$

EXEMPLO 27: Um exemplo de tal função, em $\mathrm{GF}(2)^4$ seria definida pelo sistema de equações

$$\begin{bmatrix} z_0 = & x_0\,y_0 + x_1\,y_0 + x_1\,y_1 \\ z_1 = & x_1\,y_0 + x_1\,y_2 + x_2\,y_0 \\ z_2 = & x_3\,y_3 \\ z_3 = & x_2\,y_0 + x_0\,y_2 \end{bmatrix}$$

As matrizes $\boldsymbol{\alpha}_k$ são

$$\boldsymbol{\alpha}_0 = \begin{bmatrix} 1000 \\ 1100 \\ 0000 \\ 0000 \end{bmatrix} \qquad \boldsymbol{\alpha}_1 = \begin{bmatrix} 0000 \\ 1010 \\ 1000 \\ 0000 \end{bmatrix} \qquad \boldsymbol{\alpha}_2 = \begin{bmatrix} 0000 \\ 0000 \\ 0000 \\ 0001 \end{bmatrix} \qquad \boldsymbol{\alpha}_3 = \begin{bmatrix} 0010 \\ 0000 \\ 1000 \\ 0000 \end{bmatrix}$$

A matriz $\boldsymbol{\alpha}$ congrega estas componentes individuais em vectores de *bits*

$$\boldsymbol{\alpha} = \begin{bmatrix} 1000 & 0000 & 0001 & 0000 \\ 1100 & 1000 & 0100 & 0000 \\ 0101 & 0000 & 0000 & 0000 \\ 0000 & 0000 & 0000 & 0010 \end{bmatrix}$$

Conhecida a matriz $\boldsymbol{\alpha}$, é possível recuperar a matriz $\mathbf{A}$,

$$\boldsymbol{\alpha} = \mathcal{B}_\sigma \, \mathbf{A} \, \mathcal{B}_\sigma^t \quad \Longrightarrow \quad \mathbf{A} = (\mathcal{B}')_\sigma^t \, \boldsymbol{\alpha} \, (\mathcal{B}')_\sigma \quad \text{com} \quad (\mathcal{B}')_\sigma = \mathbf{T}^{-1} \, \mathcal{B}_\sigma$$

Tomemos o polinómio característico $\mathbf{c}[X] = X^4 + X + 1$ e a base polinomial de tipo 0 apresentada no exemplo 25 (página 209).

A base dual calculada nesse exemplo foi

$$\mathcal{B}' = \left\{ 1 + \beta^3 \ , \ \beta^2 \ , \ \beta \ , \ 1 \right\}$$

As matrizes $(\mathcal{B}')_\sigma$ e $\mathbf{A}$ que daí resultam são

$$(\mathcal{B}')_\sigma = \begin{bmatrix} 9 & 4 & 2 & 1 \\ D & 3 & 4 & 1 \\ E & 5 & 3 & 1 \\ B & 2 & 5 & 1 \end{bmatrix} \qquad \mathbf{A} = \begin{bmatrix} A & 5 & B & A \\ 3 & D & D & B \\ 2 & D & 7 & 1 \\ 5 & 5 & 7 & D \end{bmatrix}$$

As **funções quadráticas** têm uma representação que deriva directamente da representação das funções bilineares: se for $z = g(x) = x_\sigma \cdot \mathbf{A} x_\sigma$ então teremos

$$z \;=\; \tilde{x} \cdot \boldsymbol{\alpha}\, \tilde{x}$$

com $\boldsymbol{\alpha} = \mathcal{B}_\sigma\, \mathbf{A}\, \mathcal{B}_\sigma^t$. Deste modo, a componente z_k do resultado é

$$z_k \;=\; \sum_{ij} \boldsymbol{\alpha}_k^{ij}\, x_i\, x_j \tag{85}$$

em que $\boldsymbol{\alpha}_k^{ij}$ denota a componente de ordem k do elemento de índices i, j da matriz $\boldsymbol{\alpha}$.

O valor booleano $\boldsymbol{\alpha}_k^{ij}$ determinam se o monómio $x_i\, x_j$ ocorre ou não na função booleana que calcula z_k. Assim, em termos de representações em $\mathrm{GF}(2)^n$, as formas quadráticas produzem (como seria de esperar) funções booleanas de grau 2; todos os monómios são da forma $x_i\, x_j$.

3.4 Cifras Primitivas, Segurança e Computabilidade

Cifras são as mais antigas e, de certo modo, as mais extensivamente usadas de todas as técnicas criptográficas. Em termos gerais, o seu objectivo é a confidencialidade da comunicação da informação em meios hostis (canais sujeitos a escuta e/ou interferência). Porém, dado o papel fulcral das cifras nestas circunstância, é fundamental formalizar com precisão o que se entende serem esses objectivos e o os critérios que avaliam se os objectivos são cumpridos.

O estudo da **segurança** das cifras é, fundamentalmente, uma formalização de objectivos e critérios e, para uma cifra específica, uma fundamentação da crença que tais critérios são, nessa cifra, verificados.

Nesta secção vamos considerar apenas uma forma particular de cifras: as que podem ser descritas completamente por uma função computável que preserva comprimentos.

107 Noção (Cifra Primitiva)
Uma função computável f *é uma* ***cifra primitiva*** *quando, para cada índice* k*, a função* f_k *é injectiva e tem inversa computável.*

Se f é uma cifra primitiva, numa relação $f(x,k) = y$, os inteiros k, x, y designam-se, respectivamente, por *chave*, *mensagem* ou *texto* e *criptograma*. A chave k identifica f_k e, por isso, quando $y = f(x,k)$ e f está implícito, diremos que k *cifra* o texto x no criptograma y. Pelo mesmo motivo k^{-1} representa a função inversa ${f_k}^{-1}$. Representemos por $\kappa(x,y) = \{\, k \mid f(x,k) = y \,\}$, o conjunto das chaves que cifram x no criptograma y.

A **segurança** de uma cifra primitiva coloca-se, fundamentalmente, em torno de três problemas básicos: a *inversão da chave* (dados texto x e criptograma y, computar a chave k), a *inversão do criptograma* (dado o criptograma y, computar o respectivo texto x), a *distinguibilidade das chaves* (dados vários pares texto e criptograma, decidir se foram gerados pela mesma chave) e a *distinguibilidade das mensagens* (dadas duas mensagens e o criptograma de uma delas, decidir a que mensagem esse criptograma corresponde).

Para os descrever precisamos de outra noção

108 NOÇÃO

*Seja f uma cifra primitiva; uma sequência de pares $S = \{x_n \| y_n\}$ é é f-**coerente** se existe uma chave k (designada **chave de coerência**) tal que, $(\forall\, n)\,[y_n = f(x_n, k)\,]$. Usaremos a notação $S \in \kappa(x, y)$ quando, sendo k a chave de coerência de S, se verifica $y = f(x, k)$.*

Pode-se agora formalizar os **problemas básicos da criptoanálise**:

Inversão da Chave

Dada uma sequência f-coerente S, determinar uma ordem n e uma máquina de Turing probabilística e computável (CPTM) que, sob "inputs" n e $(S \restriction n)$, calcule a chave de S.

Equivalentemente, vendo S como um oráculo:

Determinar uma ordem n e uma CPTM T tais que, sob "input" 0, $T^{S,n}$ (i.e. a máquina T recorrendo ao oráculo S não mais que n vezes) calcula a chave de S.

Inversão do Criptograma

Dada uma sequência S que seja f-coerente, determinar uma ordem n e uma CPTM que, sob "inputs" n, $S\restriction n$ e y, computa uma mensagem x tal que $S \in \kappa(x, y)$.

Vendo S como um oráculo esta formalização é equivalente a:

Determinar uma CPTM T e uma ordem n tais que $T^{S,n}$, sob "input" y, computa x tal que $S \in \kappa(x, y)$.

Distinguibilidade das Chaves

Dada uma sequência f-coerentes S determinar uma ordem n e uma máquina de Turing probabilística e computável que, sob "inputs" $(S\restriction n)$, $x\|y$, decida se $S \in \kappa(x, y)$.

Vendo S como oráculo, a formalização será:

Determinar uma ordem n e uma CPTM T tais que $T^{S,n}$, sob "input" $x\|y$, determina se $S \in \kappa(x, y)$.

Como caso particular de um oráculo f-coerente com chave k, pode-se considerar a própria função f_k. Isto significa

que, neste caso, a máquina de Turing pode escolher qualquer sequência finita de textos $\{x_i\}$ e recolher do oráculo os respectivos criptogramas $\{y_i\}$ (com $y_i = f(x_i, k)$), desde que o número de consultas não excede o limite n .

A designação dos três primeiros problemas básicos (inversão do criptograma, inversão de chave e distinguibilidade de chaves) usando f_k como oráculo acrescenta o qualificativo *"com oráculo de cifra"*.

Algumas considerações gerais sobre estes três problemas e a sua redutibilidade:

1. Em todos os problemas, subentende-se, dado que as TM que se pretende determinar são probabilísticas e que todas as computações são realizadas a menos de uma probabilidade de erro desprezável.
2. Todos os problemas assumem o acesso a um oráculo S que seja f-coerente e todos determinam uma ordem n que limita o número de acessos a esse oráculo.
3. Os problemas estão apresentados pela ordem de redução; isto é, uma solução para o primeiro (inversão da chave) implica uma solução para o segundo (inversão do criptograma) que, por seu turno, implica uma solução para o último (distinguibilidade das chaves).
   De facto, se existir um modo de calcular a chave k, a inversa ${f_k}^{-1}$ permite determinar um texto x tal que $k \in \kappa(x, y)$. Da mesma forma, se existir alguma forma de determinar tal $x' \in {f_k}^{-1}(y)$ (mesmo que sem conhecer k), atendendo ao facto de f_k ser uma bijecção, tem-se $k \in \kappa(x, y)$ sse $x' = x$.

□

Note-se que o objectivo de ua cifra é preservar a confidencialidade de uma mensagem. Por isso o problema

da segurança de uma cifra é, essencialmente, é um problema de transmissão de informação. A relação entre complexidade computacional e informação faz com que a intratabilidade computacional dos três primeiro problemas não implique, necessariamente, que a cifra é segura, no sentido em que essa confidencialidade é preservada.

Considere-se a seguinte situação:

1. Num sistema de voto electrónico, um um agente *Mesa* recolhe os votos de uma comunidade de eleitores e, com o objectivo de preservar a confidencialidade desses votos, envia-os cifrados para um agente *Escrutinador*.
2. Para esse fim, os agentes *Mesa* e *Escrutinador* usam uma cifra primitiva onde não é tratável fazer a inversão do criptograma. Adicionalmente usam a mesma chave para todas as trocas de mensagens.
3. A votação é referendária: existem apenas dois votos possíveis: `sim` e `nao`. Por este motivo só existem dois criptogramas possíveis que representamos por `{sim}` e `{nao}`.
4. Se um *Intruso*, recolhendo os criptogramas dos votos, puder estabelecer a associação entre cada eleitor e_i e o criptograma y_i do seu voto, então pelo menos ele pode, comparando os respectivos criptogramas, determinar quem são os eleitores que votam do mesmo modo.
5. Adicionalmente, dado que o *Intruso* é, ele próprio, um eleitor, sabe qual é o seu voto e, recolhendo o seu criptograma, determina o voto de todos os restantes eleitores.

Note-se que, mesmo sendo intratável inverter computacionalmente o criptograma, o *Intruso* consegue preverter a confidencialidade da informação pela forma como a cifra está a ser usada.

Para resolver este anomalia, qualquer sistema de votação electrónica, não permite que um *Intruso* estabeleça a relação entre um eleitor e o respectivo criptograma de voto. São usadas técnicas criptográficas adicionais cujo objectivo é preservar, para além da *confidencialidade*, o *anonimato* (ou *privacidade*) do voto.

Neste caso as técnicas de anonimato ocultam a relação entre um eleitor e o respectivo criptograma de voto; genericamente, essas técnicas ocultam a relação entre uma mensagem e a meta-informação a ela associada (p.ex: a identidade do titular, o receptor ou emissor, a data e hora, etc.).

□

A conclusão que podemos tirar do exemplo anterior é que, para segurança, não basta garantir a não-invertibilidade do criptograma ou da chave ou ainda a indistinguibilidade das chaves. É necessário algo mais que descreva a forma como duas mensagens podem ou são ser distinguíveis através dos respectivos criptogramas. Temos assim um problema adicional da criptoanálise que designaremos por *distinguibilidade das mensagens*.

Os três primeiros problemas (invertibilidade da chave, invertibilidade do criptograma e distingibilidade das chaves) usam, como fonte básica de informação, um oráculo f-coerente; isto é, um oráculo determinado por uma chave, que é desconhecida de quem resolve o problema (o atacante) mas que é sempre a mesma. O quarto problema (distinguibilidade das mensagens) estende estes oráculos assumindo que os "inputs" são pares de mensagens e que, à chave, de adiciona um "bit de escolha". Fixemos uma cifra primitiva f

109 Noção (Oráculo com escolha)
Uma sequência de triplos $\Sigma = \{x_n \| x'_n \| y_n\}$ *, determina um* **f-oráculo com escolha** *se existe uma chave* k

tal que uma, e só uma, das seguintes condições é válida: $(\forall\, n)\,[y_n = f(x_n, k)]$ *ou* $(\forall\, n)\,[y_n = f(x'_n, k)]$ *.*
A **escolha** *de* Σ *é* 0 *se a primeira condição é válida e é* 1 *se é válida a segunda condição;* k *é a chave de* Σ*.*

Sem perda de generalidade pode-se assumir que $x_n \neq x'_n$, para todo n. Essencialmente, para cada par de mensagens $x_n \| x'_n$, y_n é o criptograma de x_n, se a escolha for 0, ou é o criptograma de x'_n, se a escolha for 1. A TM que recolhe a resposta do oráculo, não sabe quais das duas condições ocorreu porque, tal como a chave k, a escolha é, supostamente, desconhecida e difícil de calcular. Obviamente, conhecido k, é trivial computar a escolha.

Distinguibilidade das mensagens (versão genérica)

Dada uma cifra primitiva f *e um* f*-oráculo de escolha* Σ*, determinar uma CPTM* T *e uma ordem* n *tal que, sob "input"* 0*, a máquina* $T^{\Sigma,n}$ *computa a escolha de* Σ*.*

Como caso particular de oráculo com escolha, defina-se o par de funções

$$F(x\|x', k) \;=\; f(x, k) \quad , \qquad S(x\|x', k) \;=\; f(x', k) \qquad \text{com} \quad x \neq x' \tag{86}$$

Com o oráculo $\Sigma = F_k$ ou $\Sigma = S_k$ a máquina de Turing pode escolher qualquer par de mensagens $x\|x'$ e recolher o criptograma y seleccionado pelo tipo de função (F ou S) e calculado com a chave k.

Para além deste oráculo particular, considere-se também um limite fixo $n = 1$; isto é, é permitida uma única consulta ao oráculo. Nestas circunstâncias, o problema de distinguibilidade das mensagens exprime-se

Distinguibilidade de mensagens (versão estrita)

Dada uma cifra primitiva f, determinar uma CPTM T que recebe como "inputs" um par de mensagens x, x' e o criptograma y de uma delas e decide de qual das mensagens y é criptograma.

A eventual redutibilidade do problema da distinguibilidade das mensagens aos três primeiros problemas (inversão da chave, do criptograma ou distinguibilidade da chave) tem de ser analisado tendo em vista que os dados do problema são distintos: a distinguibilidade das mensagens usa um oráculo com escolha Σ enquanto que os restantes problemas usam um oráculo coerente.

Obviamente se, para além do acesso a um oráculo com escolha Σ, a máquina T tivesse acesso a um oráculo S coerente com a mesma chave que Σ, então o problema de determinar a escolha de Σ é muito simplificado; nomeadamente se a máquina tiver acesso a um oráculo de cifra, determinar essa escolha seria trivial.

Por exemplo, tomemos o caso particular da versão estrita do problema da distinguibilidade das mensagens; se, de alguma forma, for possível inverter o criptograma y então este problema fica trivialmente resolvido.

Parece razoável tentar descrever a segurança de uma cifra primitiva em termos da complexidade computacional destes problemas: a cifra é segura quando qualquer um dos problemas básicos é computacionalmente intratável.

Esta abordagem não é simples porque, nomeadamente, não existe uma forma única de formalizar essa "intratabilidade". Por isso, historicamente, foram seguidas outras metodologias de formalização onde a dificuldade computacional pode ter uma expressão aferível e que iremos apresentar nas secções seguintes.

3.5 Segurança Perfeita de Cifras Primitivas

Nesta secção vamos apresentar a formalização clássica de segurança das cifras, tal como foi descrita por Shannon, e que recorre aos princípios de metodologias da Teoria da Informação. A intratabilidade das computações exprime-se pela baixa probabilidade de elas terem sucesso.

A Teoria da Informação considera que chaves, mensagens e criptogramas são representados por variáveis aleatórias, cada uma das quais descrita pela sua distribuição de probabilidade, e estabelece relações entre probabilidades.

Assumem-se três variáveis aleatórias K, X e Y. A variável K designa-se por *distribuição de chaves* enquanto as variáveis X e Y são, respectivamente, as distribuições de mensagens e de criptogramas. Assume-se que K e X são variáveis aleatórias independentes (i.e., para todo x, k, $\text{Prob}[K = k, X = x] = \text{Prob}[K = k]\text{Prob}[X = x]$). A variável Y está determinada pela função f e pelas variáveis X e K, através das relações

$$\text{Prob}[K = k, X = x, Y = y] = \begin{cases} \text{Prob}[K = k, X = x] & \text{se } y = f(x, k) \\ 0 & \text{em caso contrário} \end{cases} \quad (87)$$

$$\text{Prob}[K = k, X = x] = \text{Prob}[K = k, Y = f(x, k)] \quad (88)$$

A relação (87) traduz o asserção que, quando K e X tomam os valores k, x, a variável Y só pode ser igual a $f(x, k)$. A relação (88) traduz a injectividade de f_k: para cada possível valor k de K, Y só assume o valor $f(x, k)$ quando X tem o valor x.

Escolhidas as distribuições K e X representamos por $\mathcal{X} = \{ x \mid \text{Prob}[X = x] > 0 \}$ o conjunto de mensagens que têm probabilidade não nula e por $\mathcal{K}$ o conjunto de chaves de probabilidade não-nula. Do mesmo modo $\mathcal{Y} = \{ f(x,k) \mid x \in \mathcal{X}, k \in \mathcal{K} \}$ denota o conjunto de criptogramas de probabilidade não-nula.

Quando $x \in \mathcal{X}$ e $k \in \mathcal{K}$, as relações (87) e (88) podem ser escritas com probabilidades condicionais

$$\text{Prob}[Y = y | K = k, X = x] = \begin{cases} 1 & \text{se } y = f(x,k) \\ 0 & \text{em caso contrário} \end{cases} \tag{89}$$

$$\text{Prob}[Y = f(x,k) | K = k] = \text{Prob}[X = x] \tag{90}$$

As relações (89) e (90) podem ser reescritas, integrando ambos os membros em relação à distribuição de chaves.

$$\text{Prob}[Y = y | X = x] = \text{Prob}[K \in \kappa(x,y)] \tag{91}$$

$$\text{Prob}[X = x] = \sum_k \text{Prob}[X = k^{-1}(y)] \, \text{Prob}[K = k] \tag{92}$$

EXEMPLO 28: Considere-se um caso particular: a distribuição X é uniforme com n bits e a distribuição K é uniforme com m bits. Isto significa que, para todo x, k, se tem $\text{Prob}[X = x] = 2^{-n}$, $\text{Prob}[K = k] = 2^{-m}$, e, porque as duas distribuições são independentes, $\text{Prob}[K = k, X = x] = 2^{-n-m}$. Agora

$$\text{Prob}[X = x, Y = y] = |\kappa(x,y)| \, 2^{-m-n} \quad , \quad \text{Prob}[Y = y] = \sum_x |\kappa(x,y)| \, 2^{-n-m}$$

Por outro lado, dado que qualquer f_k é injectiva, para todos k e y tem-se $\text{Prob}[Y = y|K = k] = \text{Prob}[X = {f_k}^{-1}(y)] = 2^{-n}$. Consequentemente, para qualquer y, tem-se $\text{Prob}[Y = y] = 2^{-n}$ e $\sum_x |\kappa(x, y)| = 2^m$.

Suponhamos que Y e X são variáveis independentes; i.e., para todos x, y, $\text{Prob}[X = x, Y = y] = \text{Prob}[X = x]\,\text{Prob}[Y = y]$. Então

$$\text{Prob}[X = x, Y = y] = \text{Prob}[X = x]\,\text{Prob}[Y = y] \quad \Rightarrow \quad |k(x, y)|\,2^{-m-n} = 2^{-n}\,2^{-n} \quad \Rightarrow \quad |\kappa(x, y)| = 2^{m-n}$$

Consequentemente terá de ser $m \geq n$.

Em resumo, supondo K e X uniformemente distribuídos com m e n bits, respectivamente, a assumpção de que X e Y são variáveis independentes implica que tem de ser $m \geq n$ e que $|\kappa(x, y)| = 2^{m-n}$ independentemente de x ou y.

Este exemplo motiva a seguinte noção de segurança,

110 NOÇÃO (SEGURANÇA PERFEITA DE SHANNON)
Seja K uma distribuição de chaves e $\mathcal{M}$ um domínio de mensagens. A distribuição de textos X é **adequada** *a $K, \mathcal{M}$ se associa uma probabilidade não nula a cada $x \in \mathcal{M}$ e é independente de K.*

A cifra primitiva f tem **segurança perfeita para** *$K, \mathcal{M}$ quando, para toda a distribuição de mensagens X adequada a $K, \mathcal{M}$, a variável Y definida por* (87) *é independente de X.*

A cifra f tem **segurança perfeita** *para $\mathcal{M}$ se existe uma distribuição de chaves K para a qual f tem segurança perfeita para $K, \mathcal{M}$.*

Essencialmente a segurança perfeita ocorre quando as a probabilidade $\text{Prob}[X = x, Y = y]$ é sempre igual à probabilidade $\text{Prob}[X = x] \cdot \text{Prob}[Y = y]$. Para os criptogramas y de probabilidade não-nula ($\text{Prob}[Y = y] > 0$) e para todo x, esta afirmação é equivalente à asserção

$$\text{Prob}[X = x | Y = y] \;=\; \text{Prob}[X = x] \tag{93}$$

Uma formulação alternativa é dada pela seguinte generalização

111 PROPOSIÇÃO
A cifra f é perfeitamente segura para $K, \mathcal{M}$ se e só se, para toda a distribuição de textos X adequada a $K, \mathcal{M}$, todo o criptograma y par de textos $x, x' \in \mathcal{M}$,

$$\text{Prob}[Y = y | X = x] \;=\; \text{Prob}[Y = y | X = x'] \tag{94}$$

A prova é uma simples aplicação da definição de probabilidade condicional. As conclusões do exemplo 28 podem ser estendidas para distribuições de chaves e texto arbitrários.

112 PROPOSIÇÃO
Seja K uma distribuição de chaves; a cifra f é perfeitamente segura para $K, \mathcal{M}$ se e só se, toda a distribuição de textos X adequada a $K, \mathcal{M}$ e todo y de probabilidade não-nula e todo $x \in \mathcal{M}$,

$$\text{Prob}[K \in \kappa(x, y)] \quad = \quad \sum_k \text{Prob}[X = k^{-1}(y)] \, \text{Prob}[K = k] \tag{95}$$

Prova Consequência de (91) e (92).

O aspecto mais importante deste lema é o facto de o lado esquerdo depender de x e o lado direito não; ao invés, o lado direito é determinado pela probabilidade da variável aleatória X gerar o valor $k^{-1}(y)$. O lado esquerdo denota a probabilidade de um gerador K produzir uma chave que seja uma "chave correcta" para o par (x, y); o lado direito mede a expectativa (na distribuição K) da probabilidade de o gerador X gerar o "criptograma correcto" $k^{-1}(y)$. A definição de segurança perfeita exige que estas duas quantidades sejam iguais.

Como caso particular

113 COROLÁRIO *Se $\mathcal{M}$ é finito, então f é perfeitamente segura para $K, \mathcal{M}$ quando, para todo $x \in \mathcal{M}$ e todo o y de probabilidade não-nula, verifica-se* $\text{Prob}[K \in \kappa(x, y)] = |\mathcal{M}|^{-1}$. *Adicionalmente, se $\mathcal{K}$ for finito, tem-se* $|\mathcal{K}| \geq |\mathcal{M}|$.

Prova Considere-se um espaço de mensagens $\mathcal{M}$ finito e uma a distribuição de textos X uniforme em $\mathcal{M}$; assim tem-se $\text{Prob}[X = x] = |\mathcal{M}|^{-1}$ para todo $x \in \mathcal{M}$. Se a cifra f for perfeitamente segura para a distribuição de chaves K, o lado direito de (95) resume-se a $|\mathcal{M}|^{-1}$. Como consequência, $\text{Prob}[K \in \kappa(x, y)] = |\mathcal{M}|^{-1}$ implica, para todo $k \in \mathcal{K}$, $\text{Prob}[K = k] \leq |\mathcal{M}|^{-1}$. Integrando, $\sum_{k \in \mathcal{K}} \text{Prob}[K = k] \leq \sum_{k \in \mathcal{K}} |\mathcal{M}|^{-1}$; donde $1 \leq |\mathcal{K}|\,|\mathcal{M}|^{-1}$, e $|\mathcal{K}| \geq |\mathcal{M}|$.

Um exemplo paradigmático de cifra é a chamada **cifra de Vernam** ou **one-time-pad (OTP)**. Sem perda de generalidade pode-se definir do seguinte modo:

(i) O espaço de mensagens é o conjunto das palavras de n bits: $\mathcal{M} = \mathbb{B}^n$.

(ii) O criptograma é calculado por $y = x \oplus (k^{\infty}) \restriction |x|$, onde k^{∞} denota a repetição infinitas vezes da chave k.

Note-se que, se for $|k| \geq |x| = n$, a máscara $(k^{\infty}) \restriction |x|$ reduz-se a $k \restriction n$.

Sem perda de generalidade pode-se considerar a distribuição de mensagens X uniforme no espaço das palavras com n bits, e uma distribuição de chaves K tal que $\text{Prob}[K = k] = 0$, se $|k| \neq n$, e $\text{Prob}[K = k] = 2^{-n}$, se $|k| = n$. Desta forma, para todas as chaves de probabilidade não-nula, o criptograma é, simplesmente, $y = x \oplus k$.

Temos $|\mathcal{M}|^{-1} = 2^{-n}$ e também $\text{Prob}[K \in \kappa(x, y)] = \text{Prob}[K = x \oplus y] = 2^{-n}$. Usando o corolário 113 concluímos

114 PROPOSIÇÃO
A cifra de Vernam é perfeitamente segura.

□

A noção de segurança perfeita tem muitas limitações e está longe de ser "perfeita". Nomeadamente

1. O corolário 113 diz que, sendo f perfeitamente segura para $K, \mathcal{M}$, tem-se $|\mathcal{K}| \geq |\mathcal{M}|$. Uma cifra realista não tem possibilidade de satisfazer esta condição já que, por exemplo, para cifrar uma mensagem de `1 Mbyte` seria necessário uma chave com, pelo menos, o mesmo tamanho.

2. Ainda como consequência do corolário 113, e da observação anterior, constatamos que, para que uma cifra seja perfeitamente segura, uma chave k usada para cifrar uma mensagem x não pode ser reutilizada para cifrar uma outra mensagem x' sem que tal envolva perda de segurança.

   Isto porque, cifrar sucessivamente r mensagens $x_1, x_2, \cdots, x_r$ com a mesma chave k, é equivalente a cifrar uma só vez, com essa chave k, a mensagem $x_1 \| x_2 \| \cdots \| x_r$; agora, o espaço desta mensagem composta é $\mathcal{M}^r$ enquanto o espaço das chaves continua a ser $\mathcal{K}$. Mesmo que seja $|\mathcal{K}| \geq |\mathcal{M}|$, eventualmente para algum r, será $|\mathcal{K}| < |\mathcal{M}|^r$. [29]

Ambas estas constatações tornam irrealista, quase sempre, usar uma cifra que seja perfeitamente segura. Adicionalmente a noção de segurança perfeita assenta na hipótese de que o conhecimento que um adversário tem sobre a chave se resume à sua distribuição de probabilidade. Esta hipótese também não é realista e isso pode-se constatar quando vemos a forma como cifras "perfeitamente seguras" (por exemplo, a cifra de Vernan) se comportam face aos problemas básicos da criptoanálise que enunciamos no inicio desta secção: inversão do criptograma, inversão da chave, distinguibilidade das chaves e distinguibilidade das mensagens.

[29] A menos que $\mathcal{M}$ seja vazio, o que corresponde, obviamente, a um caso sem interesse.

3.6 Cifras e Distinguibilidade Probabilística

A noção de segurança perfeita recorre a distribuições de chaves, mensagens e criptogramas, caracterizando cada uma delas por uma distribuição de probabilidades. O modo como essa probabilidade foi estabelecida não foi determinada.

Por outro lado, as noções de segurança de cifras (nomeadamente a caracterização que fizemos dos problemas fundamentais da criptoanálise) requerem noções de computabilidade. Para ser viável estabelecer uma relação entre a visão da segurança perfeita e a visão computacional é indispensável definir com mais precisão as noções probabilísticas.

Por esta razão, variáveis aleatórias vão ser descritas por geradores (isto é, máquinas de Turing livres de prefixos) e as distribuições de probabilidade serão determinadas pelas probabilidades de paragem de tais geradores.

Recordemos que, para um gerador G, e para um eventual "output" x, a probabilidade de paragem da TM $[G = x]$ era $\mu[G = x] = \sum_{G(u) \simeq x} 2^{-|u|}$. Recordemos também que esta probabilidade de paragem é definida a menos de uma constante multiplicativa já que os comprimentos $|u|$ são definido a menos de uma constante aditiva.

Note-se, agora, que o "range" do gerador G (o conjunto dos eventuais "outputs" de G) pode ser visto como $\text{rng}(G) = \{\, x \mid \mu[G = x] > 0 \,\}$. Se o domínio $\text{rng}(G)$ for computável, então é bem determinada a probabilidade $\mu[G \in \text{rng}(G)]$ que vai coincidir com $\sum_x \mu[G = x]$.

Como referimos atrás, a probabilidade de paragem está definida a menos de uma constante multiplicativa. Por outro lado, a distribuição de probabilidade associada a uma variável aleatória X adequada a um domínio $\mathcal{M}$, deve verificar $\sum_{x\in\mathcal{M}} \text{Prob}[X = x] = 1$. Assim, para uma variável aleatória X adequada a um domínio $\mathcal{M}$ deve ser descrita por uma PFTM G tal que $\text{rng}(G)$ é computável, verifica $\mathcal{M} \subseteq \text{rng}(G)$ e tem-se

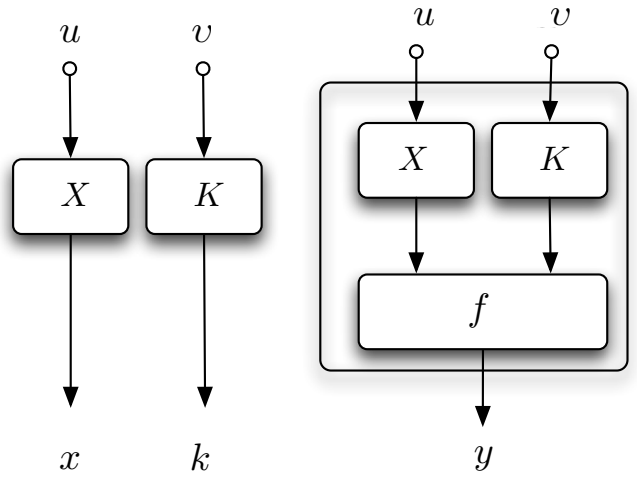
$$\text{Prob}[X = x] \doteq \mu[G = x]/\mu[G \in \mathcal{M}] \tag{96}$$

Da mesma forma, uma distribuição de probabilidades K adequada a um domínio de chaves $\mathcal{K}$ pode ser descrito por um gerador H tal que $\mathcal{K} \subseteq \text{rng}(H)$ e $\text{Prob}[K = k] = \mu[H = k]/\mu[H \in \mathcal{K}]$.

No que se segue, sem receio de ambiguidades, iremos usar a mesma notação X (ou K ou Y) para representar a variável aleatória ou o gerador que a descreve; o contexto onde o símbolo X (ou K ou Y) ocorre encarrega-se de eliminar as eventuais ambiguidades e determina se o símbolo refere a variável aleatória ou o gerador.

O gerador Y, que descreve a distribuição de criptogramas, é determinado pelos geradores X e K e, por isso, recebe dois "inputs" aleatórios

$$Y(u, v) = f(X(u), K(v)) \tag{97}$$



Para $x \in \text{rng}(X)$ defina-se o gerador $Y_x(v) = Y(u, v)$ sabendo que $X(u) = x$; i.e. $Y_x(v) = f(x, K(v))$.

115 LEMA *Com Y definido por* (97) *tem-se:*

(i) $\mu[Y = y] = \sum_{u,v:Y(u,v)\simeq y} 2^{-|u|-|v|}$.

(ii) $\mu[Y = y, X = x] = \sum_{X(u)=x \wedge Y(u,v)\simeq y} 2^{-|u|-|v|} = \mu[X = x]\,\mu[Y_x = y]$

Representemos por Y^X a indexação de geradores $\{Y_x\}$. Sejam $X \cdot Y^X$ e $X\|Y$ os geradores

$$X \cdot Y^X : (u, v) \mapsto X(u)\|Y(u, v) \tag{98}$$

$$X\|Y : (u, u', v) \mapsto X(u)\|Y(u', v) \tag{99}$$

116 LEMA *Verifica-se*

(i) $\mu[X \cdot Y^X = x\|y] = \mu[Y = y, X = x]$

(ii) $\mu[X\|Y = x\|y] = \mu[Y = y]\,\mu[X = x]$

A **segurança perfeita** verifica-se quando as variáveis aleatórias Y e X são independentes. Com a definição de probabilidade que apresentámos a segurança perfeita para o espaço de mensagens $\mathcal{M}$ é equivalente a

Existe um gerador K *tal que, para todo o gerador* X *tal que* $\mathsf{rng}(X) \supseteq \mathcal{M}$ *com o gerador* Y *determinado por* (97) *e para todo* $x \in \mathcal{M}$ *e todo* $y \in \mathsf{rng}\,(Y)$, *verifica-se*

$$\mu[Y = y, X = x] \quad = \quad \mu[X = x]\,\mu[Y = y] \tag{100}$$

Isto significa que a segurança perfeita pode ser expressa em termos de probabilidades de paragem de geradores em vez da probabilidade de distribuições probabilísticas. As vantagens deste formalismo resultam de, agora, se poder usar as propriedades e as relações que definimos nos geradores.

Ambos os lados da equação (100) podem ser vistos como probabilidades de paragem de geradores que têm, como "output", pares mensagens criptograma $x\|y$. A equação pode ser reescrita atendendo aos geradores $X \cdot Y^X$ e $X\|Y$ e ainda ao lema 116, como

$$\mu[X \cdot Y^X = x\|y] \;=\; \mu[X\|Y = x\|y] \qquad \forall\, x, y$$

Relembrando a noção de equivalência forte de geradores (ver noção 39, página 80), concluímos que a segurança perfeita é simplesmente expressa pela equivalência forte

$$X \cdot Y^X \;\equiv\; X\|Y \tag{101}$$

A figura 9 ilustra esses geradores (abstraindo-nos dos geradores X e K que produzem x e k). Quais são esses geradores? A igualdade (101) fornece uma resposta.

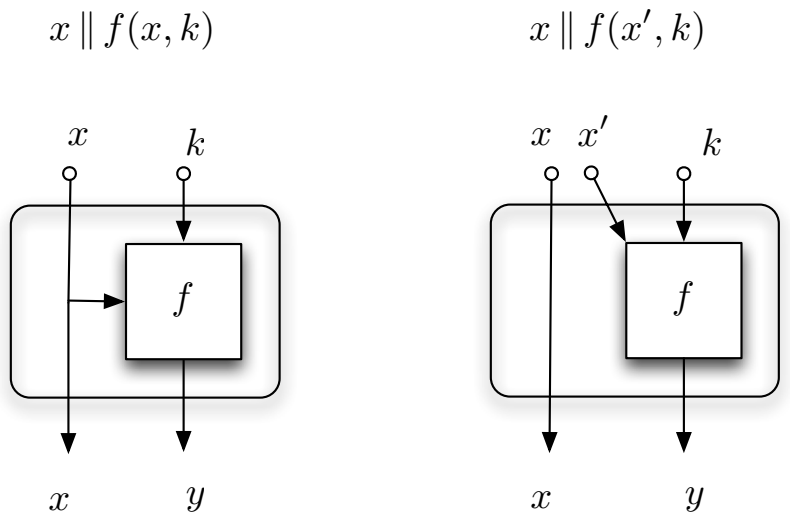


Figura 9: Segurança Perfeita e Indistinguibilidade

No gerador da esquerda, associado à probabilidade de paragem $\mu[Y = y, X = x]$, a mensagem x que aparece no "output" é a mesma que é usada como argumento da cifra f. O gerador é $X \cdot Y^X$.

No gerador da direita, o gerador $X \| Y$ associado à probabilidade de paragem $\mu[X = x]\,\mu[Y = y]$, o "output"

x é completamente independente do "output" y; por isso y é determinado por uma outra qualquer mensagem x'.

Vimos que a noção de segurança perfeita conduz a cifras que estão longe de ser realistas já que, para satifazer esta condição, a chave tem de ter um tamanho igual ou superior ao da mensagem e, adicionalmente, não pode ser usada numa outra mensagem.

Com estas exigências os problemas da criptoanálise nem sequer fazem sentido uma vez que não é possível construir dois pares distintos (mensagem,criptograma) com a mesma chave sem que isso viole a condição de segurança perfeita.

Para ser viável ter uma visão computacional da segurança é necessário, por isso, relaxar a exigência expressa em (101) (ou, originalmente, (101)), admitindo que as duas probabilidades de paragem podem não ser, sempre, exactamente iguais desde que a diferença entre elas seja "desprezável".

A condição em (100) compara as probabilidades de paragem dos geradores $X \cdot Y^X$ e $X \| Y$ para todos os pares mensagem criptograma $x \| y$ exigindo que sejam iguais. A relaxação desta condição vai exigir, apenas, que os dois geradores sejam indistinguíveis.

117 Noção (Segurança Probabilística)
A cifra f é **probabilisticamente segura** *para o espaço de mensagens $\mathcal{M}$ quando existe um gerador K tal que, para todo o gerador X que verifique $\mathcal{M} \subseteq \mathsf{rng}(X)$ e com o gerador Y definido em* (97)*, verifica-se*

$$X \cdot Y^X \quad =_{\mathsf{ind}} \quad X \| Y$$

3.7 Cifras, Aleatoriedade e Incerteza

4.Criptografia Simétrica

Neste capítulo procuraremos dar uma visão introdutória da família de técnicas criptográficas cujo uso requer o conhecimento, partilhado por todos os legítimos intervenientes, de uma única chave. Esta família de técnicas designa-se, genericamente, por **criptografia simétrica**.

Dada a limitação no número de chaves, a gama de técnicas que é possível definir é bastante limitada; essencialmente são cifras, funções de "hash" e combinações destes dois tipos base. Em contrapartida estas técnicas fornecem:

. **Elevada eficiência computacional**
Um parâmetro fundamental para aferição de uma técnica criptográfica é o seu **"throughput"**; isto é, a quantidade de informação que consegue processar numa unidade de tempo. As técnicas simétricas são as que possuem o maior "throughput" tendo capacidade para processar, em tempo real, sinais de áudio e vídeo. Nomeadamente permitem implementações em "hardware" dedicado, o que é factor essencial para aumentar o "throughput". contribui fortemente para aumentar a eficiência computacional.

. **Elevada eficiência de segurança**
A incerteza quando à viabilidade de um ataque com sucesso, expresso em bits, mede o *grau de segurança* de uma

técnica criptográfica. Nas técnicas simétricas a incerteza está sempre limitada pelo comprimento da chave[30] mas pode, no entanto, ser menor; a diferença entre o tamanho da chave e o grau de segurança da técnica, mede a *eficiência* de segurança da técnica.

As técnicas simétricas têm uma elevada eficiência de segurança já que (pelo menos, nas boa técnicas) o grau de segurança está muito próximo do tamanho da chave; por exemplo, mesma na sua versão mais limitada, o AES usa chaves de 128 bits e tem um grau de segurança muito próximo desse valor. Em contraste uma cifra assimétrica como o RSA precisa de uma chave de 1024 bits para ter um grau de segurança da ordem dos 80 bits.

. **Elevada flexibilidade e robustez**

É frequente que vários contextos de processamento de informação imponham restrições e condicionalismos de segurança específicos a uma mesma técnica criptográfica. As técnicas simétricas têm a capacidade de se adaptar a estas condições através de modos de funcionamento específicos. Também têm a flexibilidade de se metamorfosear adaptando-se às necessidades do contexto; por exemplo, as mesmas primitivas criptográficas básicas podem ser usadas para cifras de blocos, cifras sequenciais e funções de "hash".

Outro aspecto essencial é a sua robustez a ataques; quando existe a hipótese de um ataque é possível, quase sempre, efectuar ligeiras modificações à técnica básica de modo a o tornar inviável. A evolução do DES para o 3DES, que estendeu efectivamente o tempo de vida útil do DES para 30 anos, foi uma resposta simples ao

[30] Existe sempre um ataque trivial, o chamado **ataque por força bruta**, que consiste em percorrer todo o espaço de chaves até que uma delas satisfaça uma determinada condição; por exemplo, verificar se o criptograma obtido com a chave teste coincide com um determinado valor observado.

aparecimento das técnicas de criptoanálise; do mesmo modo, quando foi detectado um ataque ao SHA logo no início da sua existência, uma ligeira alteração conduziu ao SHA-1 que é, já há 15 anos, a função de "hash" mais popular.

Por estas razões as técnicas simétricas são os alicerces de todo o edifício de segurança num sistema de informação. Especificações de segurança complexas e subtis exigem, normalmente, técnicas assimétricas; no entanto, dado que as técnicas assimétricas, devido à sua elevada complexidade computacional, conseguem processar apenas pequenas quantidades de informação (chaves, "hashs", etc.), o tratamento de elevadas quantidades de dados exige sempre uma combinação de uma técnica assimétrica com uma técnica simétrica capaz de processar o volume de dados.

4.1 Nomenclatura de Cifras

Permutação

Uma permutação (também designada por **substituição simples**) é uma função bijectiva de domínio finito; isto é, uma função $f : \mathbb{B}^n \longrightarrow \mathbb{B}^n$ que seja injectiva.

As permutações são a construção invertível mais simples e estão na base das cifras ditas **determinísticas**: a permutação, aplicada duas vezes, ao mesmo "input" produz sempre o mesmo "output".

Transformação holomórfica

Uma função $h : \mathbb{N} \longrightarrow \mathbb{N}$ tal que a família de conjuntos $\left\{h^{-1}(n)\right\}_{n\in\mathbb{N}}$ é uma cobertura disjunta de $\mathbb{N}$: isto é, os conjuntos são disjuntos dois a dois e a sua união cobre todo $\mathbb{N}$.

A transformação é usada para construção das cifras homomóficas: um algoritmo probabilístico que, sob "input" x, produz um $y \in h^{-1}(x)$. A função h é usada para decifrar o criptograma y.

Cifras por blocos

São primitivas criptográficas representáveis por funções boolenas $f : \mathbb{B}^t \times \mathbb{B}^n \longrightarrow \mathbb{B}^n$ tais que, para cada $k \in \mathbb{B}^t$, a função $f(k, \cdot) : \mathbb{B}^n \longrightarrow \mathbb{B}^n$ é uma permutação. Os inteiros n e t são os comprimentos, respectivamente, do **bloco** e da **chave**.

Cifras iteradas

São cifras por blocos em que a função f é construída por composição de um número finito p de cifras por blocos $r_1, r_2, ..., r_p : \mathbb{B}^t \times \mathbb{B}^n \longrightarrow \mathbb{B}^n$, designadas por **"rounds"**, da seguinte forma:

1. Existe uma função $g : \mathbb{B}^t \longrightarrow \left(\mathbb{B}^t\right)^p$ chamada **programador de chaves** que, a cada chave $k \in \mathbb{B}^t$, associa um vector com p chaves $g(k) = (k_1, k_2, ..., k_p)$ em $\mathbb{B}^t$.
2. Para uma determinada **chave** k e **texto** x, o **criptograma** correspondente $y = f(x, k)$ é calculado através da sequência
$$y_0 = x \quad , \quad y_i = r_i(k_i, y_{i-1}) \quad i = 1, ..., t \quad , \quad y = y_t$$
em que $k_i = g(k)_i$.

Desta forma a função f é definida por uma iteração de funções $f_0, f_1, \cdots, f_t$

$$f_0(x, k) = x \quad , \quad f_i(x, k) = r_i(f_{i-1}(x, k), g(k)_i) \quad i = 1, \cdots, t \quad , \quad f = f_t$$

Numa cifra iterada o acto de decifrar é também iterado: dado o criptograma y, a sequência y_i é reconstruída iterativamente do fim para o princípio,

$$y_t = y \quad , \quad y_{i-1} = r_i^{-1}(y_i, k_i) \quad i = t, ..., 1 \quad , \quad x = y_0$$

Cifras sequenciais ("stream ciphers"):

São primitivas criptográficas descritas por funcionais $f : \mathbb{B}^t \times \mathbb{B}^\infty \to \mathbb{B}^\infty$ tais que:

(i) Para toda a chave k e todo i, o valor $f(k,u) \restriction i$ depende apenas de $u \restriction i$; isto é, se for $u \restriction i = v \restriction i$, então será $f(k,u) \restriction i = f(k,v) \restriction i$.

(ii) Para toda a chave k e todo i, a função $x \in \mathbb{B}^i \mapsto f(k, \uparrow x) \restriction i$ é uma permutação.

Este conceito de cifra sequencial efectua um processamento sequencial de mensagens *orientado ao bit*. Em cada estado constrói-se o bit de ordem i do criptograma a partir dos bits do texto de ordem $j \leq i$.

Nomeadamente, uma forma comum de cifra sequencial orientada ao bit, é descrita por uma função f da forma

$$f(k,u) \;=\; s(k) \oplus u \tag{102}$$

sendo $s : \mathbb{B}^m \to \mathbb{B}^\infty$ um **gerador pseudo-aleatório** de bits.

Neste caso o bit de ordem i do criptograma depende apenas do bit com a mesma ordem i do texto. Cifras com esta propriedade designam-se por **mono-alfabéticas**. Cifras onde o bit de ordem i depende efectivamente de todos os bits do texto de ordem $j \leq i$ e, eventualmente, de alguns de ordem $j > i$, designam-se por **poli-alfabéticas**.

Cifras sequenciais por blocos

Por vezes é necessário generalizar o conceito de cifra sequencial para o de uma cifra sequencial *orientada ao bloco*. Neste modelo tanto o texto como o criptograma são vistos como sequências infinitas de blocos de tamanho fixo n; o cálculo do bloco de ordem i do criptograma recorre ao conhecimento de todos os blocos de ordem $j \leq i$ do texto.

Formalmente, uma cifra sequencial orientada a blocos de tamanho n, com chaves de tamanho m, é uma funcional $f : \mathbb{B}^t \times \mathbb{B}^* \longrightarrow \mathbb{B}^*$ tal que:

(i) Para toda a chave k e todo i múltiplo de n, se for $u \restriction i = v \restriction i$, então $f(k,u) \restriction i = f(k,v) \restriction i$.

(ii) Para toda a chave k e todo i múltiplo de n, a função $x \in \mathbb{B}^i \mapsto f(k, \uparrow x) \restriction i$ é uma permutação.

Essencialmente, em relação à definição inicial de cifra sequencial, nesta definição substitui-se o quantificador "*todo* i" por "*todo* i *múltiplo de* n". Isto implica que um bit do criptograma de ordem i arbitrária pode depender de bits do texto que têm ordems superiores.

Uma cifra sequencial mono-alfabética orientada ao bloco de tamanho n, pode ser definida à custa de uma qualquer cifra de blocos $h : \mathbb{B}^t \times \mathbb{B}^n \longrightarrow \mathbb{B}^n$ e de um gerador de chaves $s : \mathbb{B}^t \longrightarrow (\mathbb{B}^t)^\infty$.

Para tal texto e o criptograma são sequências de blocos e define-se a funcional $f : \mathbb{B}^t \times (\mathbb{B}^n)^\infty \longrightarrow (\mathbb{B}^n)^\infty$ por

$$f(k\,,\,v) \quad = \quad h(s_0, x_0)\; h(s_1, x_1)\; \cdots\; h(s_i, x_i)\; \cdots \tag{103}$$

em que $v = x_0\, x_1 \cdots x_i \cdots$ é a sequência de blocos de "input" e $s(k) = s_0\, s_1 \cdots s_i \cdots$ é a sequência de chaves geradas a partir de k. Ou seja, o bloco de ordem i no criptograma é determinado por

$$y_i \quad = \quad f(k, v)_i \quad = \quad h(s_i, x_i) \tag{104}$$

Para construir uma cifra poli-alfabética necessita-se de uma estrutura um pouco mais complexa. Uma forma frequente, chamada **cifra com "trail" l** usa o mesmo gerador de chaves $s : \mathbb{B}^t \to (\mathbb{B}^t)^\infty$ da cifra mono-alfabética, mas introduz uma nova função $H : \mathbb{B}^* \times \mathbb{B}^n \to \mathbb{B}^n$ que é invertível no 2º argumento; isto é, para todo y, $H(y, \cdot)$ é uma permutação.

Então a sequência $\{y_i\}$ de blocos do criptograma é gerada por

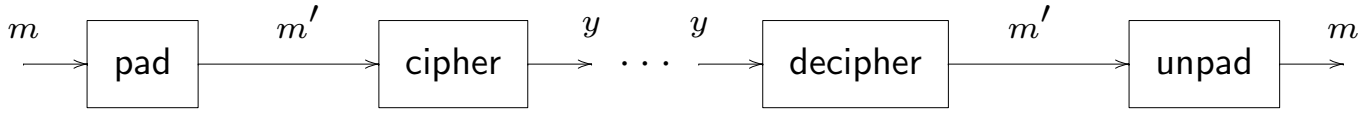
$$y_i \;=\; h(s_i, u_i) \qquad \text{com} \qquad u_i \;=\; H(y_{i-1} \| \cdots \| y_{i-l}\,,\; x_i) \tag{105}$$

Os l blocos do criptograma, anteriores a i, funcionam como chave que, com H, cifra x_i. O valor resultante u_i é, posterormente, cifrado (usado h) com a chave s_i gerada a partir de k.

Este tipo de construção levanta a questão de se saber como calcular os primeiro l blocos do criptograma. Note-se que, para calcular y_0, por exemplo, precisariamos de conhecer um "pseudo"-bloco y_{-l}. O problema resolve-se introduzindo l constantes que, independentemente do texto v, funcionam como blocos $y_{-l} \cdots y_{-1}$. Estas constantes são designadas por **vectores iniciais (iv)**.

"Padding"

Quando uma mensagem m, de comprimento arbitrário, está destinada a ser cifrada por uma cifra de blocos de tamanho fixo n, há necessidade de embeber a mensagem numa outra mensagem m' cujo comprimento é um múltiplo exacto do tamanho do bloco. Se o tamanho da nova mensagem m' for $t \cdot n$, então pode-se cifrar m' invocando t vezes a cifra de bloco n.



Este esquema exige uma função injectiva para embeber m em m' e, no final, uma função que recupere m sem ambiguidades. O mecanismo de "padding" exige duas funções:

1. Uma função injectiva pad : $\mathbb{B}^* \longrightarrow \mathbb{B}^*$ tal que, para todo $u \in \mathbb{B}^*$, $|\text{pad}(u)| = t \cdot n$, para algum $t > |u|/n$.
2. Uma função parcial unpad : $\mathbb{B}^* \rightharpoonup \mathbb{B}^*$ tal que $\text{unpad}(\text{pad}(u)) = u$, para todo u.

Modos de Cifras por blocos

Considere-se o problema de cifrar bit-strings infinitas usando uma cifra de blocos $h : \mathbb{B}^t \times \mathbb{B}^n \longrightarrow \mathbb{B}^n$ com chaves de tamanho k e blocos de tamanho n.

A forma mais simples de construir uma tal cifra, consiste em ver o "input" e o "output" como strings infinitas de blocos. e usar as construções (104) e (105) escolhendo formas particulares do gerador de chaves s e do "trail" H.

Formalmente queremos construir, usando h, uma funcional $f : \mathbb{B}^t \times (\mathbb{B}^n)^\infty \to (\mathbb{B}^n)^\infty$ que se comporte como uma cifra sequencial orientada a um bloco de tamanho n.

Numa cifra da forma (103) e (104) considere-se um gerador de chaves que se limita a repetir a chave k *ad infinitum*; isto é $s(k) = k^\infty$. Neste caso diz-se que a cifra h está em modo **electronic code book (ecb)**. No modo **ecb**, o bloco de orden i no criptograma é determinado por

$$y_i = h(k, x_i)$$

Em alternativa pode-se usar uma construção da forma (105), com o mesmo gerador de chaves $s(k) = k^\infty$, e um "trail" de tamanho 1, dado por $H(y_{i-1}, x_i) = y_{i-1} \oplus x_i$. Nesta construção diz-se que a cifra h está em modo **cipher-block chaining (cbc)**; bloco y_i, no modo **cbc**, é calculado por

$$y_i = h(k, x_i \oplus y_{i-1})$$

Uma abordagem alternativa consiste em ver o "input" e o "output" como "streams" de blocos com um tamanho m que é menor do que o tamnaho n do bloco usado por h. Os blocos y_i são gerados pela dupla recorrência

$$z_i = H(y_{i-1} \| z_{i-1}) \quad , \quad s_i = h(k, z_i) \restriction m \quad , \quad y_i = x_i \oplus s_i$$

em que $H : \mathbb{B}^* \to \mathbb{B}^n$ é uma qualquer função de "hash". Modos baseados nesta estrutura chamam-se **"feedback modes"**.

4.2 Arquitectura de Cifras por Blocos

Na maioria das cifras por blocos iteradas, os diversos "rounds", com eventual excepção do primeiro e do último, são iguais. São também formados por algumas componentes "standard". Dado que uma característica essencial das cifras por blocos é a sua eficiência computacional, essas componentes têm de ser de implementação muito simples.

O desafio essencial de qualquer arquitectura de cifra é o de construir um edifício extremamente seguro a partir de componentes muito simples e muito eficientes. Individualmente, cada uma das componentes pode não ser particularmente segura, mas a sua combinação produz a segurança pretedida.

A seguir indicam-se algumas das componentes básicas que ocorrem na construção de cifras.

Branqueamento ("**whitening**")

Sabe-se que, se $k \in \mathbb{B}^n$ for, nalgum sentido do termo, aleatório então todo $x \oplus k$ é também aleatório. Por isso a cifra $f(k, x) = x \oplus k$ dá segurança perfeita para ataques sem texto conhecido; não é possível conhecer x a partir de $y = x \oplus k$ sem conhecer a chave k.

No entanto, num ataque de texto conhecido (onde tanto x como y são conhecidos) determinar a chave k é trivial; basta fazer $k = y \oplus x$.

A cifra $f(k,x) \doteq x \oplus k$ é extremamente eficiente já que o `xor` de duas palavras de bits é directamente implementado num ciclo de máquina.

Funções Lineares

O "branqueamento" $x \oplus k$ determina, para cada k, uma função linear. Como sabemos, uma função $f : \mathbb{B}^n \longrightarrow \mathbb{B}^n$ é linear se verifica, para todo $x, y \in \mathbb{B}^n$ e $b \in \mathbb{B}$

$$f(x \oplus y) \;=\; f(x) \oplus f(y) \qquad , \qquad f(b\,x) \;=\; b\,f(x)$$

Funções lineares têm muitas representações; nomeadamente em $\mathrm{GF}(2)$ a função f é representada por uma matriz de bits $F \in \mathrm{GF}(2)^{n\times n}$; nessa representação tem-se $f(x) = F\,x$ usando as operações de álgebra matricial no corpo $\mathrm{GF}(2)$.

Também nesta representação, se a função $f(\cdot)$ for injectiva, então a matriz F tem uma inversa F^{-1} que é a representação matricial da função inversa f^{-1}

Normalmente uma função linear f é de implementação mais eficiente do que uma não linear e algumas formas particulares de funções lineares (como o "branqueamento") são mesmo extremamente eficientes.

Uma cifra $f(k,x)$, em que $f(k,\cdot)$ fosse linear para algum k, é vulnerável a um ataque muito simples. Sopunhamos que se conhecem vários criptogramas $y_i = f(k,x_i)$ e os respectivos textos x_i; seja $x = a_1\,x_1 \oplus a_2\,x_2 \cdots \oplus a_l\,x_l$,

com $a_i \in \mathrm{GF}(2)$, uma qualquer combinação linear dos diversos x_i. Então, mesmo sem conhecer a chave k, é possível calcular $y = f(k, x)$ usando a mesma combinação linear nos criptogramas; isto é, $y = a_1\, y_1 \oplus a_2\, y_2 \cdots \oplus a_l\, y_l$.

Nomeadamente, se os x_i formarem uma base do espaço vectorial $\mathrm{GF}(2)^n$, então o conhecimento de n pares (x_i, y_i) permite construir o criptograma de todos os 2^n possíveis pares (texto,criptograma), sem necessidade de conhecer a chave.

No entanto as funções lineares têm uma grande importância como componente de cifras porque uma construção da forma $y = f(x)$, sendo f linear, permite preservar a aleatoriedade de x em y.

□

Alguns exemplos de funções lineares de implementação muito eficiente.

Linear Feedback Shift Register (LFSR)

Sejam x_i, y_i os bits de ordem i dos vectores $x, y \in \mathrm{GF}(2)^n$; defina-se

$$y_0 = a_0\, x_0 + a_1\, x_1 + \cdots + a_{n-1}\, x_{n-1} \quad , \quad y_i = x_{i-1} \quad \text{para} \quad i = 1, \cdots, n-1$$

Esta transformação traduz-se no produto $y = F\, x$ em que a matriz F verifica $F_{1,i} = a_i$, $F_{i,i-1} = 1$ e $F_{i,j} = 0$ para os restantes índices. O vector $(a_0, \cdots, a_{n-1}) \in \mathrm{GF}(2)^n$ desina-se por **vector característico** do LFSR.

LFSR são uma componente muito importante de cifras sequenciais e têm a vantagem de ser implementadas em poucos ciclos máquina e directamente implementáveis em *hardware*.

Foram, durante muito tempo, usadas como componente de geradores de bit-strings pseudo-aleatórios. Um exemplo simples usa um LFSR definido por uma matriz F, do tipo atrás descrito, e a função traço $\mathrm{tr}(\cdot)$.

Gera-se uma sequência de bits b_i a partir da recorrência linear

$$x_0 \;=\; k \qquad , \qquad x_i \;=\; F\, x_{i-1} \quad \text{para } i = 1, \cdots \qquad , \quad b_i = \mathrm{tr}(x_i)$$

Esta construção pode-se generalizar usando t LFSR´s definidos por matrizes $F_1, \cdot, F_t$ e uma função booleana não-linear $\sigma : \mathrm{GF}(2)^t \longrightarrow \mathrm{GF}(2)$. Cada LFSR actua sobre um estado próprio e os traços dos diversos estados são combinados pela função σ.

A sequência de bits b_i é gerado pela recorrência

$$\begin{cases} x_{0,j} \;=\; k_j & j = 1, \cdots, t \\ x_{i,j} \;=\; F_j\, x_{i-1,j} & j = 1, \cdots, t \qquad i = 1, \cdots \\ b_i \;=\; \sigma(\mathrm{tr}(x_{i,1}), \cdots, \mathrm{tr}(x_{i,t})) \end{cases}$$

Transformações Pseudo-Hadamard (pHT)

Pode-se ver palavras $x, y \in \mathbb{B}^{2n}$ como pares de inteiros $(x_1, x_2), (y_1, y_2) \in \mathbb{Z}_n \times \mathbb{Z}_n$.

A transformação PHT define-se por $(y_1, y_2) = (2\,x_1 + x_2, x_1 + x_2)$ em que a adição $+$ e a multiplicação são feitas em $\mathbb{Z}_n$.

Assim $\begin{bmatrix} y_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ e esta transformação é, normalmente, efectuada num único ciclo máquina.

□

Vamos agora examinar funções não-lineares $f\mathbb{B}^n \longrightarrow \mathbb{B}^n$.

"Substitution boxes"

Funções não lineares genéricas $f : \mathbb{B}^n \longrightarrow \mathbb{B}^m$ são, normalmente, designadas por **substituion boxes** (ou, simplesmente, **S-boxes**).

Para um mesmo tamanho de bloco n, uma S-box $n \times n$ é muito mais difícil de implementar que uma função linear com a mesma dimensão. Por isso S-boxes são normalmente usadas com pequenos valores de n; enquanto que é relativamente simples implementar uma transformação linear para um bloco de 128 bits ou superior, as S-boxes não ultrapassam os 16 bits. Aliás, quase sempre as S-boxes são de 8 ou menos bits.

Quando n é pequeno é fácil implementar uma função f genérica como uma tabela em memória ROM; constrói-se uma memória com n linhas para o endereço de célula e cada célula contém m bits. Usando x como endereço de acesso à memória, e supondo que a célula de endereço x contém o valor $f(x)$, então, com um simples acesso à memória, calcula-se $f(x)$.

Obviamente que, atendendo à pequena capacidade de memória dos dispositivos que têm de executar estas cifras, esta estratégia só é viável se n for pequeno; por exemplo, uma S-box 8×8 requer `256 bytes` de memória; se for 16×16 requer `128 Kbytes`; se for 24×24 requer `48 Mbytes`; etc.

□

Quando são realmente necessárias funções não-lineares com elevados valores de n, então as únicas implementações possíveis são aquelas que aproveitam a forma particular destas funções.

Funções "bricklayer"

Considere-se palavras $x, y \in \mathbb{B}^n$ e considere-se um valor l que é divide n; seja $p = n/l$. Pode-se representar cada um dos elementos de $\mathbb{B}^n$ como um vector com p componentes em $\mathbb{B}^l$; isto é, usa-se o isomorfismos $\mathbb{B}^n \sim \left(\mathbb{B}^l\right)^p$.

Considere-se agora p S-boxes $f_1, \cdots, f_p \colon \mathbb{B}^l \longrightarrow \mathbb{B}^l$. Então, pode-se definir uma função não linear por

$$f \;:\; x_1 \| x_2 \| \cdots \| x_p \quad \mapsto \quad f_1(x_1) \| f_2(x_2) \| \cdots \| f_p(x_p) \tag{106}$$

Neste tipo de função as S-boxes têm dimensão $l \times l$ e podem ser razoavelmente implementáveis. Cada uma destas transformações actua apenas sob uma das componentes do "input" x e produz a componente, com a mesma ordem, no "output" y. O "output" é construído bloco a bloco e não é de estranhar que este tipo de funções se designe por "bricklayer".

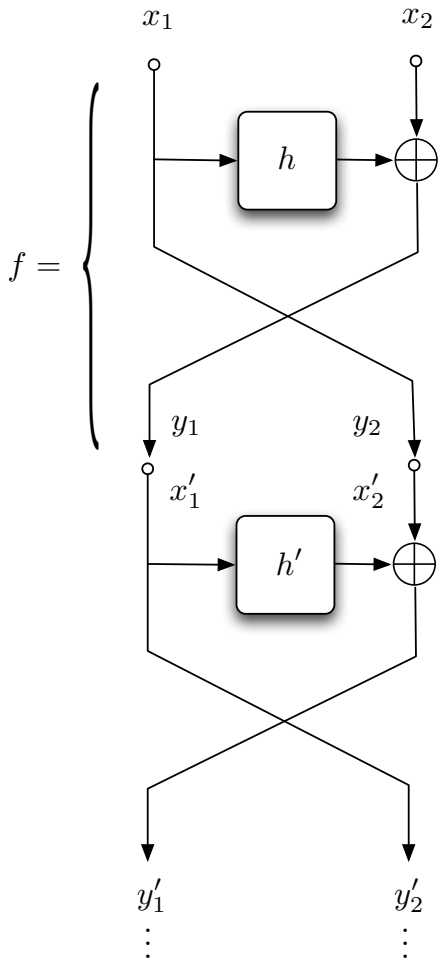
Suponhemos que f é esta função "bricklayer" e que um atacante conhece x e y e procura determinar k tal que $y = f(x \oplus k)$. Normalmente, para uma função não-linear bem escolhida, este problema teria complexidade exponencial com n. Porém, para uma função "bricklayer", o problema resume-se a p problemas diferentes de tamanho l. Ou seja, passamos de uma complexidade $O((2^l)^p)$ para uma complexidade $O(p\,2^l)$.

Por este motivo, funções "bricklayer" não podem ser usadas isoladamente como "rounds" de cifras. O seu "output" têm de ser pós-processado (normalmente por uma transformação linear) de modo que, no final, se tenha uma função $n \times n$ cujo comportamento se aproxime de uma função ideal.

Circuitos de Feistel

Uma dificuldade importante na escolha de funções não-lineares em "rounds" de cifras está no cumprimento do requisito de que os "rounds" têm de ser invertíveis. Construir e implementar eficientemente funções não-lineares invertíveis é um problema difícil.

Com o DES (que serviu de standard criptográfico no sistema financeiro durante quase 30 anos) surgiu uma construção que tomou o nome do principal autor do DES: os *circuitos de Feistel*.



Vamos tomar uma função $h\colon \mathbb{B}^n \longrightarrow \mathbb{B}^n$, não-linear mas não necessaramente invertível. A função $f\colon \mathbb{B}^{2n} \longrightarrow \mathbb{B}^{2n}$ é construída do seguinte modo

$$f \quad : \quad (x_1, x_2) \quad \mapsto \quad (x_2 \oplus h(x_1)\ ,\ x_1) \tag{107}$$

A 1ª componente do "input" é transferida para a 2ª componente do "output" sem qualquer transformação; a 2ª componente do "input" é misturado com $h(x_1)$ para produzir a 1ª componente do "output". Das equações

$$y_1 \ = \ x_2 \oplus h(x_1) \qquad , \qquad y_2 \ = \ x_1$$

concluímos

$$x_1 \ = \ y_2 \qquad , \qquad x_2 \ = \ y_1 \oplus h(y_2)$$

Desta forma a inversa de f é a função

$$f^{-1} \quad : \quad (y_1, y_2) \quad \mapsto \quad (y_2\ ,\ y_1 \oplus h(y_2)) \tag{108}$$

que é uma função análoga a f. De facto, com a ajuda da função "troca" $t\colon (x, y) \mapsto (y, x)$, vemos que $f^{-1} \ = \ t \circ f \circ t$.

A figura representa a composição de dois circuitos de Feistel formando um designado **ciclo de Feistel**. Agora todo o bit do input é transformado, pelo menos, uma vez; ao invés no circuito de Feistel onde x_1 é passado para o "output" sem modificações.

4.3 Rijndael-AES

Em 2/10/00 o *National Institute of Standards and Technology* (NIST) do *US Department of Commerce* recomendoa adopção do algoritmo designado por **Rijndael**como **Advanced Encryption Standard** (AES).

O AES foi adoptado em 2002 como *standard* para criptografia simétrica e substituiu *standard* designado por DES (Data Encryption Standard) que, na sua versão inicial ou na versão modificada designada como TRIPLEDES, tem vindo a ser usado desde 1977 na segurança de todo o sector financeiro bem como em inúmeras outras situações.

□

Alguns pontos fundametais sobre a arquitectura do AES

1. O Rijndael é uma cifra iterada com blocos de tamanho variável ($N_b \times 32$ bits, com $N_b = 4, 6$ ou 8) e chaves de tamanho variável ($N_k \times 32$ bits, com $N_k = 4, 6$ ou 8). O algoritmo tem 10 iterações (*rounds*) quando $N_b = N_k = 4$, tem 14 iterações quando $N_b = 8$ ou $N_k = 8$ e tem 12 iterações nos restantes casos.

2. O Rijndael **não é** um circuito de Feistel. É uma cifra orientada ao byte em que cada iteração (*round*) é uma substituição composta por três transformações invertíveis designadas por **camadas** (*layers*):
   - **Camada não linear (*non-linear layer*)**
     aplicação paralela de S-*boxes* destinada a evitar correlações cruzadas,

- **Camada linear (*linear mixing layer*)**
  garante a difusão dos bits transformados,
- **Mistura de chaves (*key addition layer*)**
  mistura das sub-chaves por simples operações **XOR**

3. Antes da primeira iteração é aplicada uma mistura de chaves prévia. A motivação para este tipo de operação (conhecido por *whitening*) reside no facto que qualquer transformação invertível que ocorra antes da primeira mistura de chaves pode ser retirada sem afectar as propriedades de segurança da cifra; é, portanto, redundante.
   De modo a melhorar a fase de decifragem a camada linear da última iteração é diferente da dos restantes.

Para facilitar a apresentação a descrição do algoritmo será aqui feita para o caso mais simples: blocos e chaves de 128 bits ($N_b = N_k = 4$) e 10 iterações.

Round do Rijndael

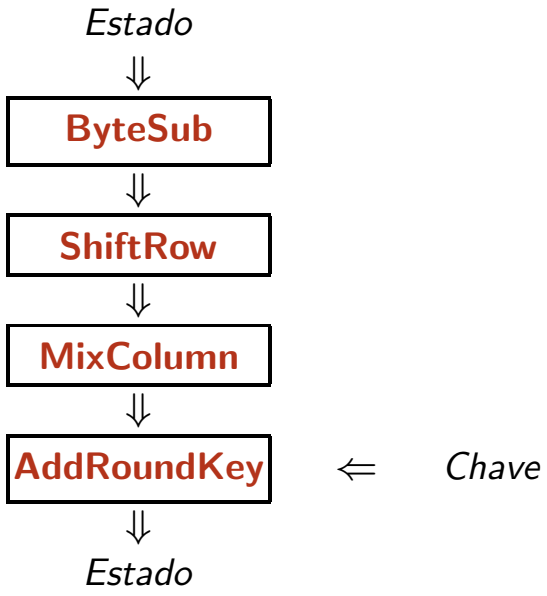
A definição de *round* é uma transformação sobre um **estado** de 128 bits organizado como uma matriz 4×4 de bytes. Cada *round* usa uma única sub-chave de 128 bits também organizada como uma matriz 4×4 de bytes. Ambas matrizes são sequencialmente organizadas "em colunas": i.e. a posição (i, j) da matriz corresponde à posição $i + 4 \times j$ num vector uni-dimensional.

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$

Estado

$k_{0,0}$	$k_{0,1}$	$k_{0,2}$	$k_{0,3}$
$k_{1,0}$	$k_{1,1}$	$k_{1,2}$	$k_{1,3}$
$k_{2,0}$	$k_{2,1}$	$k_{2,2}$	$k_{2,3}$
$k_{3,0}$	$k_{3,1}$	$k_{3,2}$	$k_{3,3}$

Chave



Cada uma dos primeiros 9 *rounds* do Rijndael é uma sequência de 4 transformações sobre o estado acima representadas. O último *round* é análogo mas não tem a transformação **MixColumn**. Adicionalmente, antes do 1º *round* há uma transformação **AddRoundKey**.

Globalmente a sequência de transformações é

$$\begin{array}{l}\texttt{AddRoundKey} \Rightarrow \\ (\ \texttt{ByteSub} \Rightarrow \texttt{ShiftRow} \Rightarrow \texttt{MixColumn} \Rightarrow \texttt{AddRoundKey} \Rightarrow\ ) \quad \times 9 \\ \texttt{ByteSub} \Rightarrow \texttt{ShiftRow} \Rightarrow \texttt{AddRoundKey}\end{array}$$

As duas transformações ByteSub e ShiftRow formam a *Non Linear Layer*. A transformação MixColumn forma a *Linear Mixing Layer* enquanto que AddRoundKey implementa a *Key Addition Layer*.

Representação dos *bytes*

Cada byte (organizado do bit mais significativo para o menos significativo) $b_7\, b_6\, \cdots\, b_2\, b_1\, b_0$ é interpretado como um polinómio em Y do 7º grau

$$b_7 \times Y^7 + b_6 \times Y^6 + \cdots + b_2 \times Y^2 + b_1 \times Y + b_0$$

São definidas operações de soma e multiplicação de bytes do seguinte modo:

Soma $A \oplus B$ calcula-se fazendo o **XOR** bit-a-bit dos respectivos coeficientes.

Multiplicação $A \otimes B$ calcula-se multiplicando os polinómios e reduzindo o resultado módulo $Y^8+Y^4+Y^3+Y+1$.

EXEMPLO

$$\begin{aligned}\texttt{ff} \otimes \texttt{03} &= (Y^7+Y^6+Y^5+Y^4+Y^3+Y^2+Y+1) \otimes (Y+1) = \\ &= (Y^8+1) \pmod{Y^8+Y^4+Y^3+Y+1} = Y^4+Y^3+Y = \\ &= \texttt{1a}\end{aligned}$$

As operações de multiplicação e similares (como o cálculo da inversa) podem ser realizadas usando uma *tabela de logaritmos*: cada byte $b \neq 0$, pode ser escrito na forma

$$b = g^e \pmod{Y^8+Y^4+Y^3+Y+1} \qquad \text{com} \quad g = \texttt{ff}$$

para um único expoente $e \in \{0 \cdots 254\}$ determinado pela tabela 4 apresentada na página 277. Para $e > 254$, tem-se $g^e \equiv g^{e \bmod 255}$.

EXEMPLO

Vamos usar a tabela para calcular o produto $\texttt{ff} \otimes \texttt{03}$ e o inverso $\texttt{aa}^{-1}$.

Vemos que $\texttt{ff} = g^1$ e $\texttt{03} = g^{73}$. Logo $\texttt{ff} \otimes \texttt{03} = g^1 \otimes g^{73} = g^{74} = \texttt{1a}$.

Vemos que $\texttt{aa} = g^{223}$; logo $\texttt{aa}^{-1} = g^{-223 \pmod{255}}$. Como $-223 \equiv (255-223) \equiv 32 \pmod{255}$ o inverso será $\texttt{aa}^{-1} = g^{32} = \texttt{12}$.

Uma tabela de logaritmos e a sua inversa podem ser facilmente armazenadas numa S-Box (que se resume a um vector de 256 bytes). A implementação de operações de multiplicação ou calculo de inversas resumem-se a algumas **XOR** de bytes e procuras na tabela; deste modo são computacionalmente muito eficientes.

Transformação `ByteSub`

Actua em paralelo e de modo uniforme sobre cada um dos bytes da matriz do estado transformando-o do modo seguinte

$$b(Y) \quad \Longrightarrow \quad u(Y) \oplus v(Y) \times b^{-1}(Y) \pmod{Y^8+1}$$

$$\text{com} \quad u \equiv \texttt{c6} \quad \text{e} \quad v \equiv \texttt{f1}$$

Notas

1. A característica não linear da cifra é basicamente implementada na transformação $b \Longrightarrow b^{-1}$.
2. A multiplicação desta inversa por v usa um polinómio (Y^8+1) diferente do usado nas operações algébricas gerais sobre os bytes (aumentando a não-linearidade intrínseca). v é escolhido de modo que $v^{-1} \pmod{Y^8+1}$ exista.
3. A transformação pode ser implementada eficientemente numa S-Box 8×8 invertível. Na tabela 4 (página 276) é apresentada essa S-Box. A transformação inversa implementa-se usando esta mesma S-Box mas em sentido oposto.

4. A constante u é escolhida de modo que a S-Box não tenha pontos fixos (pontos b onde $S(b) = b$) nem anti-pontos fixos (pontos b onde $S(b) = \bar{b}$).

Transformação `ShiftRow`

As linhas da matriz que representa o estado sofrem uma rotação circular: a primeira não é rodada, a segunda roda uma posição, a terceira roda duas posições e a última roda 3 posições

$$\begin{array}{|c|c|c|c|} \hline a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \\ \hline \end{array} \Longrightarrow \begin{array}{|c|c|c|c|} \hline a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,3} & a_{1,0} & a_{1,1} & a_{1,2} \\ a_{2,2} & a_{2,3} & a_{2,0} & a_{2,1} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,0} \\ \hline \end{array}$$

A transformação inversa efectua-se rodando as linhas o mesmo número de posições mas em sentido inverso.

Transformação `MixColumn`

Nesta transformação do estado cada coluna da matriz que representa o estado é interpretada como um polinómio em X do 3º grau

$$\mathbf{a}_j(X) \equiv a_{0,j} + a_{1,j} \times X + a_{2,j} \times X^2 + a_{3,j} \times X^3$$

Nestes polinómios a soma é efectuada componente a componente e a multiplicação é efectuada módulo (X^4+1). As operações básicas são a multiplicação de cada $\mathbf{a}(X)$ por uma matriz $\mathbf{C}$ ou pela sua inversa $\mathbf{C}^{-1}$ dadas por

$$\mathbf{C}\ \mathbf{a}_j \equiv \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \otimes \begin{bmatrix} a_{0,j} \\ a_{1,j} \\ a_{2,j} \\ a_{3,j} \end{bmatrix} \qquad \mathbf{C}^{-1}\ \mathbf{a}_j \equiv \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \otimes \begin{bmatrix} a_{0,j} \\ a_{1,j} \\ a_{2,j} \\ a_{3,j} \end{bmatrix}$$

A operação **MixColumn** multiplica por **C** cada uma das 4 colunas do estado

$$\begin{bmatrix} \mathbf{a}_0 & \mathbf{a}_1 & \mathbf{a_2} & \mathbf{a}_3 \end{bmatrix} \implies \begin{bmatrix} \mathbf{C}\,\mathbf{a}_0 & \mathbf{C}\,\mathbf{a}_1 & \mathbf{C}\,\mathbf{a}_2 & \mathbf{C}\,\mathbf{a}_3 \end{bmatrix}$$

A operação inversa de **MixColumn** é análoga mas usa a matriz $\mathbf{C}^{-1}$.

Transformação `AddRoundKey`

É efectuado um **XOR** byte-a-byte do estado e da sub-chave

$$\begin{array}{|cccc|} \hline a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \\ \hline \end{array} \implies \begin{array}{|cccc|} \hline a_{0,0} \oplus k_{0,0} & a_{0,1} \oplus k_{0,1} & a_{0,2} \oplus k_{0,2} & a_{0,3} \oplus k_{0,3} \\ a_{1,0} \oplus k_{1,0} & a_{1,1} \oplus k_{1,1} & a_{1,2} \oplus k_{1,2} & a_{1,3} \oplus k_{1,3} \\ a_{2,0} \oplus k_{2,0} & a_{2,1} \oplus k_{2,1} & a_{2,2} \oplus k_{2,2} & a_{2,3} \oplus k_{2,3} \\ a_{3,0} \oplus k_{3,0} & a_{3,1} \oplus k_{3,1} & a_{3,2} \oplus k_{3,2} & a_{3,3} \oplus k_{3,3} \\ \hline \end{array}$$

A transformação AddRoundKey coincide com a sua inversa.

A cifra inversa

Como cada transformação de um *round* é invertível, a sua inversa constrói-se por aplicação das inversas das diversas componentes pela ordem inversa. Notando que $\texttt{AddRoundKey}^{-1} = \texttt{AddRoundKey}$ a transformação será

$$\texttt{AddRoundKey} \Rightarrow \texttt{MixColumn}^{-1} \Rightarrow \texttt{ShiftRow}^{-1} \Rightarrow \texttt{ByteSub}^{-1}$$

O primeiro *round* não tem a transformação $\texttt{MixColumn}^{-1}$ e o último *round* é seguido de uma aplicação extra de AddRoundKey. As sub-chaves são usadas pela ordem inversa da usada na cifragem.

Note-se que

1. A ordem relativa das transformações ByteSub e ShiftRow é indiferente porque ByteSub do mesmo modo para qualquer byte e ShiftRow apenas muda a ordem relativa dos bytes.
2. A ordem relativa de MixColumn e AddRoundKey é indiferente porque MixColumn é uma transformação linear, AddRoundKey é uma soma e as transformações lineares $f(\cdot)$ verificam $f(a+k) = f(a) + f(k)$.

Se notar-mos que o 1º *round* não tem a transformação $\texttt{MixColumn}^{-1}$ vemos então (usando estas duas transposições)

que a decifragem é dada pela sequência exactamente análoga à da cifragem

$$\begin{array}{l}\texttt{AddRoundKey} \Rightarrow \\ (\ \texttt{ByteSub}^{-1} \Rightarrow \texttt{ShiftRow}^{-1} \Rightarrow \texttt{MixColumn}^{-1} \Rightarrow \texttt{AddRoundKey} \Rightarrow\ ) \quad \times 9 \\ \texttt{ByteSub}^{-1} \Rightarrow \texttt{ShiftRow}^{-1} \Rightarrow \texttt{AddRoundKey}\end{array}$$

4.4 Programador de Chaves

O programador de chaves gera, a partir da chave principal, tantas sub-chaves (com o mesmo tamanho da chave principal) quantos os *rounds* mais 1.

Note-se que cada *round* contém uma transformação **AddRoundKey** e, adicionalmente, antes do primeiro round existe uma operação **AddRoundKey** extra.

Vamos ilustrar a geração das sub-chaves para o caso em que $N_b = N_k = 4$ e são usados 10 *rounds*. Os princípios gerais são os seguintes:

- A `RoundKey` é um vector unidimensional de palavras de 32 bits (4 bytes) e de tamanho 4×11 . Sequencialmente são usadas 4 palavras deste vector para formar cada uma das sub-chaves.
- A chave principal (designada `CipherKey`) é expandida para a `RoundKey` segundo o algoritmo seguinte.
  1. O vector `RoundKey` é inicializado com 11 cópias do vector `CipherKey`.
  2. Para $i = 4$ até 44 a palavra na posição i é substituída pelo **XOR** da palavra na posição $i - 4$ e pela palavra w calculada da seguinte forma.
     (a) Se i não é múltiplo de 4 usa-se para w a palavra na posição $i - 1$,
     (b) Se i é múltiplo de 4 e se for $k = i/4$ e s a palavra na posição $i - 1$, então

$$w = \texttt{ByteSub4}(\texttt{rot4}(s)) \oplus \texttt{cons}(k)$$

em que:

i. $\texttt{cons}(k)$ é uma palavra constante

$$(Y^k, 00, 00, 00)$$

em que o 1º byte é representado pelo polinómio Y^k e os restantes três bytes são 00.

ii. $\texttt{rot4}(s)$ roda os 4 bytes de s uma posição:

$$(b_0, b_1, b_2, b_3) \Rightarrow (b_1, b_2, b_3, b_0)$$

iii. $\texttt{ByteSub4}(\cdot)$ aplica a S-Box ByteSub a cada um dos 4 bytes da palavra.

Tabelas AES

S-Box da cifra Rijndael

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

Tabela de logaritmos de gerador $g = \texttt{ff}$ em GF(2^8) com polinómio carcaterístico $Y^8 + Y^4 + Y^3 + Y + 1$.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	-	0	40	73	80	146	113	174	120	247	186	197	153	34	214	219
1	160	37	32	2	226	93	237	107	193	89	74	65	254	15	4	64
2	200	137	77	208	72	245	42	88	11	69	133	162	22	138	147	90
3	233	116	129	110	114	75	105	78	39	180	55	192	44	21	104	166
4	240	239	177	27	117	139	248	94	112	86	30	253	82	10	128	8
5	51	68	109	189	173	183	202	14	62	151	178	131	187	238	130	148
6	18	125	156	210	169	26	150	228	154	161	115	12	145	19	118	63
7	79	163	220	3	95	132	232	211	84	176	61	142	144	235	206	231
8	25	49	24	230	217	252	67	53	157	207	179	249	33	215	134	106
9	152	140	126	236	70	76	38	35	122	29	50	98	168	97	48	205
a	91	111	108	198	149	28	229	143	213	46	223	225	242	199	54	99
b	102	136	191	195	218	123	171	92	227	121	23	234	170	85	188	241
c	58	244	165	57	196	100	250	36	209	167	66	175	190	9	13	212
d	194	81	201	45	155	96	52	83	185	6	59	159	158	71	103	243
e	119	221	203	31	5	41	43	56	135	127	172	224	17	204	251	60
f	124	47	216	141	101	7	182	222	184	87	20	164	246	181	16	1

Na talela 4 a A entrada $(\texttt{d}_1, \texttt{d}_2)$ contém o expoente e tal que $b = g^e$ se escreve, em hexadecimal $\texttt{d}_1\texttt{d}_2$.

4.5 Outras Cifras por Blocos

Nos últimos anos, para além da cigra Rijndael adoptada como cifra AES, outras cifras mereceram atenção. Nomeadamente

Twofish cifra patrocinada por Bruce Schneier ao concurso AES e, durante bastante tempo, considerada como favorita. É uma cifra de construção que se pode classificar como "tradicional". Usa o conhecimento experimental para propor uma arquitectura de Feistel dupla.

Kasumi Retirado do "ETSI 3rd Generation Partnership Project (3GPP) Tecnical Specification"

> *"The 3GPP Confidentiality and Integrity Algorithms f8 & f9 have been developed through the collaborative efforts of the European Telecommunications Standards Institute (ETSI), the Association of Radio Industries and Businesses (ARIB), the Telecommunications Technology Association (TTA), the T1 Committee. The f8 & f9 Algorithms Specifications may be used only for the development and operation of 3G Mobile Communications and services."*

Kasumi

Na terminologia KASUMI, `f8` designa o algoritmo que implementa 8 "rounds" de cifra enquanto que `f9` é uma função de *hash* derivada de `f8`.

KASUMI é uma cifra de Feitsel com 8 "rounds" que usa blocos de 64 bits e chaves de 128 bits.