Introdução à Criptografia Moderna

13 de outubro de 2024

Conteúdo

Capítulo 1

Introdução

A crise que veio à tona com as denúncias de Edward Snowden em 2013 revelou a extensão dos programas de vigilância em massa conduzidos pela NSA e outras agências de inteligência ao redor do mundo. Essas revelações mostraram como a privacidade de cidadãos comuns estava sendo sistematicamente violada, provocando uma reação significativa de diversos setores da sociedade civil.

Em resposta à crise de privacidade, grupos autônomos, em parceria com ONGs e outros agentes da sociedade civil organizada, começaram a organizar eventos conhecidos como *criptoparties* e *cryptorraves*. Esses eventos tinham como objetivo promover o uso de criptografia ponta a ponta entre a população, ensinando técnicas e ferramentas para proteger a privacidade das comunicações.

São Paulo tornou-se um importante centro para esses eventos, sediando algumas das maiores criptoparties e cryptorraves do mundo. Nesses eventos, voluntários ensinavam aos participantes como usar tecnologias de criptografia para proteger suas comunicações.

Criptografia ponta a ponta (E2EE) é uma técnica de comunicação segura onde somente as partes que estão se comunicando podem ler as mensagens. Isso significa que, mesmo que a comunicação seja interceptada, ninguém além dos destinatários pretendidos pode acessar o conteúdo.

Para entender a importância da criptografia ponta a ponta (E2EE), é fundamental compará-la com outros modelos de comunicação segura. Um desses modelos é o protocolo SSL/TLS, amplamente utilizado na web.

SSL (Secure Sockets Layer) e seu sucessor, TLS (Transport Layer Security), são protocolos de segurança que protegem as comunicações na internet.

Quando você acessa um site seguro (por exemplo, um site que começa com `https://`), seu navegador está usando SSL/TLS para estabelecer uma conexão segura com o servidor do site.

Simplificadamente o modelo SSL/TLS funciona da seguinte forma:

1. Você envia escreve uma mensagem (m) no navegador.
2. A mensagem é criptografada e se transforma em uma cifra (c_1) que é enviada para o servidor.
3. O servidor tem a chave da cifra (c_1) e a descriptografa recuperando a mensagem (m).
4. A mensagem (m) é novamente criptografada se transformando em uma nova cifra (c_2) e é enviada para o destinatário.
5. O destinatário tem a chave da cifra (c_2) e a descriptografa para recuperar a mensagem (m)

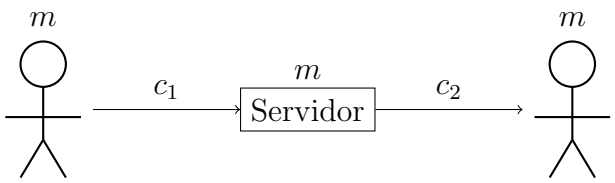


Figura 1.1: Modelo tradicional

Criptografia ponta a ponta é um modelo de segurança onde as mensagens são criptografadas no dispositivo do remetente e só podem ser descriptografadas no dispositivo do destinatário. Isso significa que nenhum intermediário, nem mesmo o servidor que transmite a mensagem, pode acessar o conteúdo da comunicação.

A criptografia ponta a ponta funciona da seguinte forma:

1. Você escreve uma mensagem (m) e a criptografa no seu dispositivo produzindo uma cifra (c).
2. A mensagem criptografada (c) é enviada ao servidor.
3. O servidor transmite a mensagem criptografada (c) ao destinatário.

4. O destinatário recebe a mensagem criptografada (c) e a descriptografa no seu dispositivo recuperando a mensagem original (m).

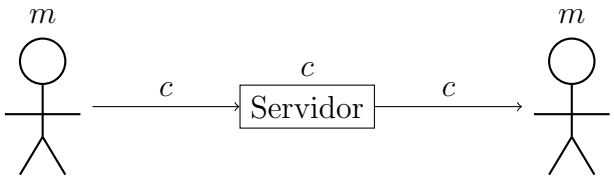


Figura 1.2: Criptografia ponta a ponta

Antes da popularização da criptografia de ponta a ponta (E2EE), a vigilância em massa abusava do acesso a servidores de grandes empresas como Microsoft, Meta, Alphabet, Apple, entre outras. A concentração das comunicações globais nos servidores dessas poucas empresas– conhecidas como big techs– cria o que, no jargão da área, é chamado de ponto único de falha. Quando uma organização como a NSA conseguia acesso a esses servidores, ela podia monitorar os dados de centenas de milhões de usuários dessas plataformas. Isso era possível porque, embora a comunicação entre o usuário e o servidor fosse segura graças ao protocolo SSL/TLS, esse protocolo protege apenas a comunicação em trânsito e não garante a segurança dos dados armazenados nos servidores. Assim, as mensagens e outros dados ficavam acessíveis aos administradores dessas plataformas ou a quem tivesse acesso aos servidores, permitindo a vigilância em massa.

A popularização da criptografia ponta a ponta obriga uma mudança significativa na abordagem da vigilância. Com E2EE, mesmo que uma organização tenha acesso aos servidores intermediários, ela não consegue ler o conteúdo das mensagens. Somente os dispositivos dos remetentes e destinatários têm as chaves necessárias para descriptografar as mensagens. Isso força as agências de vigilância a mudarem o foco de uma vigilância em massa para uma vigilância direcionada a alvos específicos, que são suspeitos de atividades ilícitas.

Nos primeiros eventos de criptoparty, as tecnologias ensinadas eram frequentemente obsoletas, como o PGP (Pretty Good Privacy). Embora o PGP garanta a criptografia ponta a ponta (E2EE), ele é complexo e de difícil uso para a maioria das pessoas, exigindo a troca manual de chaves públicas. A complexidade e a dificuldade de uso do PGP limitaram sua adoção em larga escala, evidenciando a necessidade de uma tecnologia mais acessível.

Reconhecendo essa necessidade, formou-se um consenso entre ativistas e tecnólogos de que era crucial desenvolver ferramentas de criptografia que fossem acessíveis a todos.

Foi nesse contexto que o Signal foi desenvolvido por ativistas de privacidade. O Signal oferece criptografia ponta a ponta de forma simples e intuitiva, tornando a segurança acessível para todos os usuários. A maior conquista desse movimento foi a implementação do protocolo de criptografia desenvolvido para o Signal no WhatsApp, um aplicativo com uma base de usuários de muitos milhões. A adoção do protocolo de criptografia ponta a ponta do Signal pelo WhatsApp marcou um avanço significativo na proteção da privacidade em comunicações digitais, democratizando o acesso a comunicações seguras para uma vasta audiência global.

A popularização da criptografia ponta a ponta representa um avanço crucial na proteção da privacidade individual, mas também levanta novos desafios e questões sobre a relação entre segurança, privacidade e controle. As tecnologias de criptografia estão em constante evolução, e o debate sobre sua implementação envolve diversos atores: a sociedade civil, a academia, o governo e as empresas. Cada um desses grupos tem sua própria visão e papel na busca por soluções que equilibrem a necessidade de segurança com os direitos fundamentais à privacidade.

Ao longo deste livro, apresentamos as principais primitivas criptográficas e mostramos como elas garantem, de maneira formal, diferentes noções de segurança. Nosso foco foi oferecer uma base sólida para que os leitores compreendam os fundamentos da criptografia moderna, abordando as definições de segurança e as suposições necessárias para garantir a robustez dos sistemas criptográficos.

Espero que este material ajude os estudantes e interessados a entender melhor não só os aspectos técnicos das primitivas criptográficas, mas também como essas soluções podem ser aplicadas para enfrentar problemas reais relacionados à segurança da informação. Além disso, espero que os leitores possam refletir sobre o papel que a criptografia desempenha na sociedade contemporânea e como as soluções propostas pela sociedade civil, pela academia, pelos governos e pelas empresas podem contribuir para a construção de um ambiente digital mais seguro e confiável.

Parte I

Criptografia Simétrica

Capítulo 2

Cifras Clássicas

A internet é um meio de comunicação inerentemente promíscuo. As partes que se comunicam pela rede não têm controle sobre os caminhos que suas mensagens percorrem. Essa característica, entretanto, não é exclusiva da era digital. No século XVIII, por exemplo, toda correspondência que passava pelo serviço de correios de Viena, na Áustria, era direcionada para um escritório chamado de "black chamber". Lá, o selo da carta era derretido, o conteúdo copiado, o selo recolocado e a carta reenviada ao destinatário. Todo esse processo durava cerca de três horas para evitar atrasos na entrega. Tal prática não era isolada; todas as potências europeias da época operavam suas próprias "black chambers".

Com a invenção do telégrafo e do rádio, a espionagem se tornou ainda mais sofisticada. O telégrafo permitiu a instalação de grampos nas linhas de comunicação, enquanto o rádio possibilitou a interceptação de transmissões sem fio [Kah96].

Essa capacidade de monitoramento e interceptação persiste na era digital, onde as comunicações pela internet podem ser facilmente observadas por diversos atores, sejam governamentais ou privados. Portanto, a necessidade de métodos robustos de criptografia nunca foi tão crítica.

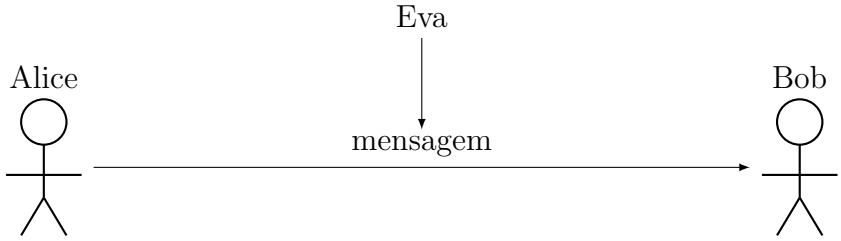
Para estudar a segurança da comunicação vamos adotar o seguinte modelo. Duas partes, o remetente e o destinatário, desejam se comunicar. Tradicionalmente, chamamos o remetente de Alice e o destinatário de Bob. O uso dos nomes Alice e Bob é comum em todos os livros de criptografia, provavelmente porque suas letras iniciais são as duas primeiras letras do alfabeto: A e B.

Nossa principal suposição é que o canal de comunicação entre Alice e Bob

é inseguro. Isso significa que assumimos que terceiros, a quem chamamos de Eva, podem observar as mensagens que trafegam pelo canal. O nome Eva vem do inglês "Eve", derivado de "eavesdrop", que significa "bisbilhotar".

Essa suposição é frequentemente referida como a "hipótese da comunicação hacker". Para os propósitos deste curso, sempre trabalharemos com essa hipótese.

A *criptografia* (do grego "escrita secreta") é a prática e o estudo de técnicas para garantir a comunicação segura na presença de terceiros, conhecidos como *adversários*. O nosso primeiro desafio neste curso é apresentar sistemas de comunicação que garantam a *confidencialidade*. Em outras palavras, qualquer mensagem enviada de Alice para Bob deve ser compreensível apenas para Alice e Bob e incompreensível para Eva, o adversário.



A importância da comunicação confidencial entre civis tem se tornado cada vez mais urgente, mas no meio militar suas origens são ainda mais antigas. Suetônio (69-141 d.C.), um historiador e biógrafo romano, descreveu, cerca de dois mil anos atrás, como o imperador Júlio César (100 a.C. - 44 a.C.) escrevia mensagens confidenciais:

> "Se ele tinha qualquer coisa confidencial a dizer, ele escrevia cifrado, isto é, mudando a ordem das letras do alfabeto, para que nenhuma palavra pudesse ser compreendida. Se alguém deseja decifrar a mensagem e entender seu significado, deve substituir a quarta letra do alfabeto, a saber 'D', por 'A', e assim por diante com as outras."

O esquema que chamaremos de cifra de César é ilustrado pelo seguinte exemplo:

```
Mensagem: transparencia
Cifra:    XUDQVSDUHQFLD
```

Como descrito por Suetônio, a regra para encriptar uma mensagem de Júlio César consistia em substituir cada letra da mensagem por aquela que está três posições à frente na ordem alfabética. Para descriptografar a cifra, substitui-se cada letra por aquela que está três posições atrás.

Aqueles que, como César, conheciam ou descobriam essa regra eram capazes de entender as mensagens. Para todos os outros, a mensagem parecia apenas uma sequência sem sentido de letras. Esse método simples, embora engenhoso para a época, tinha uma grande vulnerabilidade: uma vez que a regra de criptografia era conhecida, qualquer um podia decifrar a mensagem facilmente. Em outras palavras, o segredo é sua própria regra.

Embora técnicas de criptografia e criptoanálise existam desde o Império Romano, foi com o advento do telégrafo e sua capacidade de comunicação eficiente que o campo se estruturou. No fim do século XIX, um militar holandês chamado Auguste Kerckhoffs estabeleceu um princípio fundamental para as cifras, que ficou conhecido como o *princípio de Kerckhoffs*.

Esse princípio estabelece que a regra usada para criptografar uma mensagem, mesmo que codificada em um mecanismo, não deve ser um segredo. Ou seja, não deve ser um problema se esse mecanismo cair nas mãos do adversário ou que a regra seja descoberta.

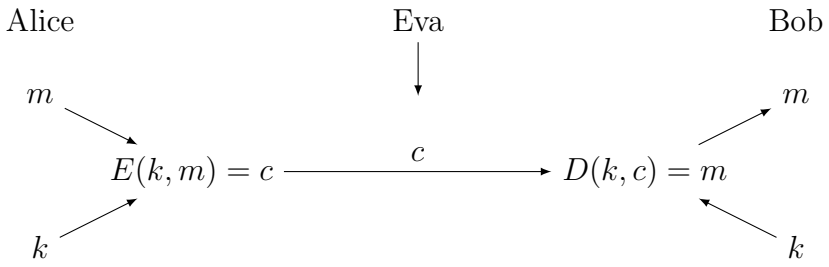
Outros criptógrafos importantes reafirmaram esse princípio de maneiras diferentes. Claude Shannon, considerado o fundador da teoria da informação, dizia que devemos sempre assumir que "o inimigo conhece o sistema". Whitfield Diffie, um dos pioneiros da criptografia moderna, afirmou que "um segredo que não pode ser rapidamente modificado deve ser interpretado como uma vulnerabilidade".

Ou seja, em uma comunicação confidencial, as partes devem compartilhar algo que possa ser facilmente modificado caso necessário. Esse segredo compartilhado é chamado de *chave* da comunicação e assumimos que ela é a única parte sigilosa do sistema.

Trazendo o debate para uma discussão mais moderna, o sigilo do código-fonte de um sistema não deve, em hipótese alguma, ser o que garante sua segurança. A segurança deve depender apenas da chave secreta, não do algoritmo ou do código-fonte.

O modelo de *criptografia simétrica* pode ser descrito da seguinte maneira: Alice (a remetente) usa um algoritmo público (E) que, dada uma chave (k), transforma uma mensagem (m) em um texto incompreensível chamado de cifra (c). A cifra é então enviada a Bob (o destinatário) através de um meio assumidamente inseguro (hipótese da comunicação hacker). Bob, por sua vez,

usa a mesma chave em um outro algoritmo (D) para recuperar a mensagem original a partir da cifra.



Este modelo é chamado de simétrico porque assume que ambas as partes, Alice e Bob, compartilham a mesma chave. Mais adiante, veremos um outro modelo chamado assimétrico.

Um sistema de criptografia simétrico é formado por três algoritmos:

- *Gen*: o algoritmo que gera uma chave secreta.
- E: o algoritmo que criptografa, ou seja, transforma uma mensagem em uma cifra.
- D: o algoritmo que descriptografa, ou seja, recupera a mensagem a partir de uma cifra.

Considere que temos um sistema Π formado por três algoritmos como esses. Digamos que uma mensagem m foi criptografada usando o algoritmo E e a chave k, produzindo a cifra c (escrevemos $E(k, m) = c$). O sistema é considerado *correto* se, ao descriptografarmos c com o algoritmo D usando a mesma chave k, recuperamos a mensagem original m (ou seja, $D(k, c) = m$).

2.1 Cifra de Deslocamento

O que chamamos na seção anterior de "cifra de César" não deve ser propriamente considerado um sistema de criptografia, pois não possui uma chave. No entanto, é possível e simples adaptar esse esquema para incorporar uma chave.

Para fazer isso, em vez de deslocar as letras sempre três casas para frente, vamos assumir que foi sorteado previamente um número k entre 0 e 25. Ou

seja, nosso algoritmo *Gen* sorteia um número de 0 a 25 Esse número será a chave da comunicação, e assumiremos que ambas as partes a compartilham.

O mecanismo para criptografar uma mensagem (E) será deslocar cada letra k posições para a direita. Para descriptografar (D), basta deslocar cada letra as mesmas k posições para a esquerda.

Exemplo 1. *Digamos que o algoritmo Gen sorteie a chave $k = 3$:*

A Tabela 2.1 mostra a posição original de cada letra e a nova posição após o deslocamento de 3 posições. Cada linha da tabela indica a letra original, sua posição no alfabeto, a nova posição após o deslocamento e a letra resultante após o deslocamento.

Para calcular a nova posição de cada letra após o deslocamento, usamos a aritmética modular*. A aritmética modular é uma maneira de lidar com valores que "reiniciam" após atingirem um certo valor, similar ao funcionamento de um relógio. No contexto da cifra de deslocamento, utilizamos a aritmética modular para assegurar que, ao deslocar uma letra, continuamos dentro dos limites do alfabeto.*

Como estamos usando o alfabeto latino básico, que possui 23 letras, para obter a posição deslocada somamos k à posição original módulo 23. Ou seja, pegamos o resto da soma quando dividido por 23.

Se a mensagem for `transparencia` *a cifra será* `XUDQVSDUHQFLD`*.*

- *E(*3,`transparencia`*)* = `XUDQVSDUHQFLD`
- *D(*3,`XUDQVSDUHQFLD`*)* = `transparencia`

A criptoanálise da cifra de deslocamento revela que ela não é segura, principalmente porque a quantidade de chaves existentes é pequena. *Criptoanálise* é a prática de analisar e decifrar sistemas criptográficos com o objetivo de encontrar vulnerabilidades ou descobrir o conteúdo de mensagens cifradas sem conhecer a chave secreta. Chamamos de *universo das chaves* o conjunto que abrange todas as chaves possíveis em um sistema de criptografia. No caso da cifra de deslocamento, a chave é um número entre 0 e 22, resultando em apenas 23 chaves possíveis. Este pequeno universo de chaves torna o sistema vulnerável a ataques de força bruta.

Um *ataque de força bruta* envolve testar todas as chaves possíveis até encontrar a correta. Dado que existem apenas 23 chaves possíveis na cifra de deslocamento, um atacante pode simplesmente tentar cada uma delas. Em média, precisaríamos testar $\frac{23}{2} = 11,5$ chaves até encontrar a correta.

Original		**Deslocada**	
Letra	**Posição**	**Posição**	**Letra**
a	0	3	D
b	1	4	E
c	2	5	F
d	3	6	G
e	4	7	H
f	5	8	I
g	6	9	J
h	7	10	L
i	8	11	M
j	9	12	N
l	10	13	O
m	11	14	P
n	12	15	Q
o	13	16	R
p	14	17	S
q	15	18	T
r	16	19	U
s	17	20	V
t	18	21	X
u	19	22	Z
v	20	23	A
x	21	0	B
z	22	1	C

Tabela 2.1: Tabela de deslocamento para $k = 3$

O valor esperado da quantidade de tentativas necessárias para encontrar a chave correta é sempre metade do tamanho do universo das chaves.

Embora todos os sistemas de criptografia estejam teoricamente sujeitos a ataques de força bruta, a viabilidade prática do ataque depende do tamanho do universo de chaves. Em sistemas com um grande universo de chaves, um ataque de força bruta se torna impraticável devido ao tempo e aos recursos computacionais necessários.

Exemplo 2. *Muitos dos roteadores modernos possuem um mecanismo chamado WPS (Wi-Fi Protected Setup), que supostamente simplifica o processo*

de conexão, especialmente na configuração do hardware. O WPS permite que um usuário se conecte remotamente e sem fio ao roteador, desde que possua um PIN (Personal Identification Number). Esse PIN é uma sequência de oito dígitos de 0 a 9, o que significa que o universo das chaves é $10^8 \approx 2^{27}$.

Para entender melhor, o universo das chaves, 10^8*, significa que existem 100 milhões de combinações possíveis. Em termos binários, isso é aproximadamente* 2^{27}*, que é um número relativamente pequeno para os padrões de segurança modernos.*

Devido ao tamanho limitado do universo das chaves, um ataque de força bruta é viável. Um ataque de força bruta envolve tentar todas as combinações possíveis de PIN até encontrar a correta. Neste contexto, um atacante pode testar sistematicamente cada um dos 100 milhões de PINs possíveis. Dependendo da velocidade da conexão e da capacidade de processamento do atacante, este processo pode levar entre 4 e 8 horas para ser concluído.

Esse tempo relativamente curto para realizar um ataque de força bruta expõe uma significativa vulnerabilidade no mecanismo WPS. Uma vez que um atacante consegue encontrar o PIN correto, ele pode acessar a rede Wi-Fi, comprometendo a segurança e a privacidade dos dados transmitidos.

Portanto, enquanto o WPS pode facilitar a configuração inicial do roteador, ele também introduz um risco considerável de segurança. Para mitigar esse risco, é recomendado desativar o WPS e utilizar métodos de configuração de segurança mais robustos, como a configuração manual de uma senha Wi-Fi forte e a utilização de protocolos de segurança avançados como WPA3.

Como em geral uma chave é uma sequência aleatória de bits, uma chave de n bits corresponde a um universo de 2^n chaves. O tamanho da chave seguro é algo que depende do avanço tecnológico. Nos anos 70, o padrão para criptografia usava chaves com 56 bits, que eram consideradas seguras na época. No entanto, em 1998, uma chave deste tamanho foi quebrada em um ataque de força bruta, demonstrando que a evolução tecnológica havia tornado esse tamanho de chave vulnerável.

Apesar disso, um universo de chaves de 80 bits, é tão grande que mesmo os avanços tecnológicos não devem comprometer sua segurança. Como veremos, hoje em dia, os sistemas costumam ter chaves com pelo menos 128 bits, o que torna um ataque força bruta contra esses sistemas inviável.

2.2 Cifra de Substituição

Em 1567, a residência da rainha da Escócia, foi destruída por uma explosão, resultando na morte do então rei, que era seu primo. O principal suspeito do assassinato foi dispensado da pena e, surpreendentemente, casou-se com Mary no mês seguinte. Este evento tumultuoso levou à prisão de Mary na Inglaterra.

Mary subiu ao trono da Escócia em 1542. Muitos católicos consideravam Mary a legítima herdeira do trono inglês, que estava ocupado pela protestante Elizabeth I, que reinava na Inglaterra desde 1558. Durante seu tempo na prisão, Mary conspirou com seus aliados para assassinar Elizabeth e reivindicar o trono.

Em 1587, Mary foi executada pelo que ficou conhecido como a Conspiração de Babington. A principal prova usada para sua condenação foi uma troca de cartas cifradas que foram interceptadas e decifradas pelas autoridades inglesas [Sin04].

A cifra usada pelos conspiradores é conhecida hoje como *cifra de substituição* ou *cifra monoalfabética*. Neste tipo de criptografia, cada letra ou par de letras é substituída por um símbolo, que pode ser inclusive outra letra.

A chave de uma cifra de substituição é uma tabela que associa cada letra a um símbolo. Para que a cifra funcione corretamente, a quantidade de símbolos deve coincidir com a quantidade de letras no alfabeto. Isso significa que podemos substituir cada letra por outra letra sem perder nenhuma informação.

Em outras palavras, a chave de uma cifra de substituição é uma permutação do alfabeto. Cada letra do alfabeto é substituída por outra letra de acordo com esta tabela de permutação.

Exemplo 3. *Vamos considerar que temos a seguinte Tabela de substituição 2.2.*

A partir desta chave, podemos criptografar textos substituindo cada letra pela letra correspondente na chave. Por exemplo, usando a tabela de permutação fornecida, a letra `a` *seria substituída por* `Z`*, a letra* `b` *por* `E`*, e assim por diante. Este processo transforma o texto original em um texto cifrado, onde cada letra do alfabeto original é substituída por sua contraparte definida na chave de permutação.*

```
Mensagem: transparencia
Cifra:    QOZKPMZOAKBFZ
```

Letra Original	Letra Substituída
a	Z
b	E
c	B
d	R
e	A
f	S
g	C
h	D
i	F
j	G
k	H
l	I
m	J
n	K
o	L
p	M
q	N
r	O
s	P
t	Q
u	T
v	U
w	V
x	W
y	X
z	Y

Tabela 2.2: Tabela de permutação da cifra de substituição

Para descriptografar o texto cifrado, basta reverter o processo. Isso significa substituir cada letra do texto cifrado pela letra original do alfabeto. Usando novamente a tabela de permutação, se encontrarmos a letra `Z` *no texto cifrado, substituímos de volta por* `a`*, se encontrarmos* `E`*, substituímos por* `b`*, e assim por diante. Este processo inverso reconstrói o texto original a partir do texto cifrado.*

O desfecho da história da Conspiração de Babington sugere que a cifra

monoalfabética não é muito segura. No entanto, até o desenvolvimento das primeiras máquinas de criptografar, versões das cifras monoalfabéticas eram as mais populares no mundo todo.

Um exemplo interessante é que, nos anos 70, a Editora Abril publicou no Brasil o famoso "Manual do Escoteiro Mirim" da Disney, que apresentava uma cifra monoalfabética. Este manual era um sucesso entre as crianças e ensinava técnicas básicas de criptografia usando cifras de substituição simples.

Em 2017 o curioso caso do desaparecimento de um rapaz no Acre viralizou quando seus familiares revelaram que, no seu quarto, havia uma coleção de livros que ele havia escrito de maneira criptografada. Após investigações, foi descoberto que o rapaz usara uma cifra de substituição.

A cifra de substituição é interessante porque seu universo de chaves é extremamente vasto. O universo das chaves da cifra de substituição é o conjunto de todas as permutações possíveis do alfabeto. Se estamos usando o alfabeto latino básico, com 23 letras, a análise combinatória nos indica que esse universo possui $23! \approx 4.10^{26} \approx 2^{88}$ combinações possíveis. Este número é dezenas de milhares de vezes maior do que o número de grãos de areia em todas as praias da Terra, estimado em 10^{19} e dez vezes maior do que o número de estrelas no universo observável, que é estimado em cerca de 10^{22}.

Este número colossal torna um ataque de força bruta impraticável, pois seria necessário testar um número inimaginável de chaves para encontrar a correta.

Apesar disso, como sabe qualquer pessoa que tenha jogado jogos como criptogramas, que são vendidos pelas famosas revistas Coquetel, é relativamente fácil quebrar uma cifra de substituição. Isso porque esse sistema é vulnerável a um outro tipo de ataque, o ataque de frequência.

Um *ataque de frequência* explora as características estatísticas do texto original. Em qualquer idioma, certas letras aparecem com mais frequência do que outras. Por exemplo, em textos em português, as letras `a`, `e` e `o` são muito comuns, enquanto letras como `x`, `z` e `q` são usadas com menos frequência. Um criptoanalista pode analisar a frequência das letras no texto cifrado e compará-las com as frequências conhecidas do idioma para fazer suposições educadas sobre quais letras correspondem a quais.

Para piorar – ou melhorar dependendo da perspectiva – na maioria das línguas há dígrafos particulares. Por exemplo, no português, dois símbolos repetidos provavelmente representam o `r` ou o `s`, e o `h` quase sempre vem depois do `l`, do `c` ou do `n`. Se o texto a ser decifrado for suficientemente

longo, essas pistas podem ser suficientes para quebrar a cifra.

No seguinte trecho de "O escaravelho de ouro" de Edgar Allan Poe a personagem descreve essa técnica que ela utilizou para decifrar um texto em inglês:

> "Ora, no inglês, a letra que ocorre com mais frequência é a letra `e`. Depois dela, a sucessão é: `a o i d h n r s t u y c f g l m w b k p q x z`. O e prevalece de tal maneira que quase nunca se vê uma frase isolada em que ele não seja predominante. Aqui nós temos, portanto, bem no início, uma base que permite mais do que um mero palpite. O uso que se pode fazer da tabela é óbvio, mas, neste criptograma em particular, não precisamos nos valer dela por inteiro. Como nosso caractere dominante é o `8`, começaremos assumindo que este é o `e` do alfabeto normal. (...)"

Em português, faz sentido separar as letras em cinco blocos, com frequência de ocorrência decrescente:

1. `a`, `e` e `o`
2. `s`, `r` e `i`
3. `n`, `d`, `m`, `u`, `t` e `c`
4. `l`, `p`, `v`, `g`, `h`, `q`, `b` e `f`
5. `z`, `j`, `x`, `k`, `w` e `y`

Esses blocos refletem a probabilidade de ocorrência das letras em textos em português, com as letras dos primeiros blocos aparecendo com mais frequência do que as dos últimos blocos. Essa distribuição permite que um criptoanalista faça suposições informadas sobre quais letras no texto cifrado correspondem a quais letras no texto original, facilitando a quebra da cifra. Foi uma análise assim que permitiu que o governo inglês decifrasse os textos da rainha Mary da Escócia, o que acabou lhe custando a vida.

Esse tipo de análise pode ser feito manualmente ou, hoje em dia, com a ajuda de programas de computador, e é extremamente eficaz contra cifras de substituição simples. A análise de frequência foi uma das primeiras técnicas de criptoanálise desenvolvidas e continua a ser uma ferramenta poderosa para

quebrar cifras que não introduzem complexidade suficiente para mascarar as características estatísticas do texto original.

Portanto, embora o universo de chaves de uma cifra de substituição seja vasto, a simplicidade do algoritmo o torna vulnerável a técnicas de criptoanálise como a análise de frequência.

2.3 Cifra de Vigenère

A cifra de Vigenère foi criada no século XV e, ainda no começo do século XX, era considerada inquebrável. Em 1868, o matemático e autor de "Alice no País das Maravilhas", Charles Lutwidge Dodgson (mais conhecido pelo pseudônimo Lewis Carroll), descreveu a cifra como "inquebrável". Ainda em 1917, a cifra continuava a ser amplamente reconhecida, e um artigo da revista Scientific American a descreveu como "impossível de decifrar".

Essa percepção de invulnerabilidade decorre da sofisticação da cifra de Vigenère, que pertence a uma classe de cifras chamadas de *polialfabéticas*. Diferente das cifras monoalfabéticas, que usam uma única substituição para todo o texto, as cifras polialfabéticas utilizam múltiplos alfabetos de substituição. Isso significa que a mesma letra no texto original pode ser cifrada de diferentes maneiras, dependendo de sua posição e da chave utilizada.

Em poucas palavras, a cifra de Vigenère consiste em deslocar as letras do texto original em distâncias diferentes, determinadas por uma chave. Em sua versão mais simples, a chave é uma palavra cuja primeira letra indica quantas casas devemos deslocar a primeira letra da mensagem, a segunda letra da chave indica quantas casas devemos deslocar a segunda letra, e assim por diante. Quando a mensagem ultrapassa o tamanho da chave, a chave é repetida e o processo continua.

Para facilitar a conta na hora de criptografar e descriptografar, podemos usar uma tabela que indica para cada letra da mensagem e cada letra da chave qual é a letra correspondente na cifra. Essa tabela é chamada de *tabula recta* e está representada na Figura 2.3. A tabula recta é uma matriz que alinha os alfabetos de forma que cada linha representa um alfabeto deslocado. Para cifrar, encontra-se a letra da mensagem na coluna da letra da chave correspondente, e a interseção dá a letra cifrada.

Exemplo 4. *Considere a seguinte mensagem criptografada com a chave* `senha` *usando a cifra de Vigenère:*

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**X**	**Z**
A	A	B	C	D	E	F	G	H	I	J	L	M	N	O	P	Q	R	S	T	U	V	X	Z
B	B	C	D	E	F	G	H	I	J	L	M	N	O	P	Q	R	S	T	U	V	X	Z	A
C	C	D	E	F	G	H	I	J	L	M	N	O	P	Q	R	S	T	U	V	X	Z	A	B
D	D	E	F	G	H	I	J	L	M	N	O	P	Q	R	S	T	U	V	X	Z	A	B	C
E	E	F	G	H	I	J	L	M	N	O	P	Q	R	S	T	U	V	X	Z	A	B	C	D
F	F	G	H	I	J	L	M	N	O	P	Q	R	S	T	U	V	X	Z	A	B	C	D	E
G	G	H	I	J	L	M	N	O	P	Q	R	S	T	U	V	X	Z	A	B	C	D	E	F
H	H	I	J	L	M	N	O	P	Q	R	S	T	U	V	X	Z	A	B	C	D	E	F	G
I	I	J	L	M	N	O	P	Q	R	S	T	U	V	X	Z	A	B	C	D	E	F	G	H
J	J	L	M	N	O	P	Q	R	S	T	U	V	X	Z	A	B	C	D	E	F	G	H	I
L	L	M	N	O	P	Q	R	S	T	U	V	X	Z	A	B	C	D	E	F	G	H	I	J
M	M	N	O	P	Q	R	S	T	U	V	X	Z	A	B	C	D	E	F	G	H	I	J	L
N	N	O	P	Q	R	S	T	U	V	X	Z	A	B	C	D	E	F	G	H	I	J	L	M
O	O	P	Q	R	S	T	U	V	X	Z	A	B	C	D	E	F	G	H	I	J	L	M	N
P	P	Q	R	S	T	U	V	X	Z	A	B	C	D	E	F	G	H	I	J	L	M	N	O
Q	Q	R	S	T	U	V	X	Z	A	B	C	D	E	F	G	H	I	J	L	M	N	O	P
R	R	S	T	U	V	X	Z	A	B	C	D	E	F	G	H	I	J	L	M	N	O	P	Q
S	S	T	U	V	X	Z	A	B	C	D	E	F	G	H	I	J	L	M	N	O	P	Q	R
T	T	U	V	X	Z	A	B	C	D	E	F	G	H	I	J	L	M	N	O	P	Q	R	S
U	U	V	X	Z	A	B	C	D	E	F	G	H	I	J	L	M	N	O	P	Q	R	S	T
V	V	X	Z	A	B	C	D	E	F	G	H	I	J	L	M	N	O	P	Q	R	S	T	U
X	X	Z	A	B	C	D	E	F	G	H	I	J	L	M	N	O	P	Q	R	S	T	U	V
Z	Z	A	B	C	D	E	F	G	H	I	J	L	M	N	O	P	Q	R	S	T	U	V	X

Tabela 2.3: Tabula Recta

```
Mensagem: transparencia
Chave:    senhasenhasen
Cifra:    LVNUSHEELNUMN
```

Apesar da fama de "inquebrável" que a cifra de Vigenère ostentou até o começo do século XX, desde a metade do século anterior já eram conhecidos métodos de criptoanálise capazes de derrotar esse tipo de cifra.

Em 1854, John Hall Brock Thwaites submeteu um texto cifrado utilizando uma cifra supostamente por ele inventada. Charles Babbage, o inventor das máquinas que precederam o computador moderno, mostrou que no fundo a cifra de Thwaites era equivalente à cifra de Vigenère. Após ser desafiado, Babbage decifrou uma mensagem criptografada por Thwaites duas vezes com chaves diferentes.

Em 1863, Friedrich Kasiski formalizou um ataque contra a cifra de Vigenère que ficou conhecido como *teste de Kasiski*. O ataque considera o fato de que a chave se repete com uma frequência fixa e, portanto, há uma probabilidade de produzir padrões reconhecíveis.

Exemplo 5. *Considere o exemplo extraído da Wikipédia:*

```
Mensagem: cryptoisshortforcryptography
Chave:    abcdabcdabcdabcdabcdabcdabcd
Cifra:    CSASTPKVSIQUTGQUCSASTPIUAQJB
```

Note que o padrão `CSASTP` se repete na cifra. Isso ocorre porque o prefixo "crypto" foi criptografado com a mesma chave. Em um texto suficientemente longo, um padrão como este fatalmente ocorrerá. Uma vez encontrado, é calculada a distância entre as repetições. Neste caso, a distância é 16, o que significa que o tamanho da chave deve ser um divisor de 16 (2, 4, 8 ou 16). Com esta informação, podemos aplicar um ataque de frequência nos caracteres de 2 a 2, de 4 a 4, de 8 a 8 e de 16 a 16.

Embora a cifra de Vigenère seja significativamente mais complexa do que as cifras monoalfabéticas, ela não é invulnerável. Técnicas avançadas de criptoanálise, desenvolvidas ao longo do tempo, eventualmente permitiram a quebra dessa cifra. Por exemplo, o método de Kasiski e a análise de frequência foram ferramentas essenciais na criptoanálise da cifra de Vigenère.

2.4 Máquinas de Criptografar

No final da primeira década do século XX, foram inventadas as primeiras máquinas de criptografar. A componente principal dessas *máquinas eletromecânicas* é um conjunto de *rotores*. A configuração inicial dos rotores contém a chave da criptografia. Cada vez que o operador pressiona uma tecla, os rotores embaralham as letras. Dessa forma, essas máquinas rotoras se comportam como uma sofisticada cifra polialfabética.

Para descriptografar a mensagem, o operador precisa ajustar a máquina em modo de descriptografia, configurar a máquina com a chave secreta inicial e digitar o texto cifrado. A máquina então se rearranja para produzir o texto original quando digitado.

Essas máquinas foram um grande avanço na criptografia, proporcionando um nível de segurança muito maior do que as cifras anteriores. O uso de rotores e a capacidade de alterar a configuração inicial significavam que cada letra do texto original poderia ser cifrada de várias maneiras diferentes, dependendo da posição dos rotores, tornando a criptoanálise extremamente difícil.

As máquinas rotoras mais conhecidas são da série *Enigma*. Elas foram criadas por um inventor alemão no final da Primeira Guerra Mundial e versões mais modernas foram extensamente usadas durante a Segunda Guerra pelo exército nazista. As versões mais simples da máquina possuíam três rotores capazes de gerar $26^3 \approx 17.500$ possíveis configurações iniciais. Além disso, era possível trocar a ordem dos rotores, multiplicando por 6 o número de combinações possíveis, chegando a um total de cerca de 105 mil possibilidades. A versão utilizada pelo exército nazista, porém, permitia cerca de 150 trilhões de possibilidades.

Em 1939, Alan Turing, matemático britânico e pioneiro da computação, desenvolveu uma máquina eletromecânica chamada *Bombe*, capaz de decifrar algumas cifras de máquinas Enigma com 3 rotores. Posteriormente, a Bombe foi melhorada para decifrar mensagens de máquinas Enigma mais sofisticadas. Este trabalho de decifração foi crucial para os esforços dos Aliados durante a Segunda Guerra Mundial e marcou um avanço significativo na criptoanálise.

A história da computação esbarra na história da criptografia neste ponto. Poucos anos antes da guerra, Alan Turing demonstrara que a satisfatibilidade da lógica de primeira ordem é um problema indecidível. Para tanto, ele propôs um modelo computacional que hoje chamamos de Máquinas de Turing. Diferente dos modelos computacionais anteriores, como o cálculo

lambda de Church ou as funções recursivas de Gödel, o modelo de Turing era intuitivo. Além disso, Turing mostrou que era possível construir com seu modelo uma Máquina Universal capaz de simular qualquer outra Máquina de Turing. Esse resultado magnífico é o que dá origem à computação moderna.

O primeiro modelo de computador desenvolvido por Turing e sua equipe em Bletchley Park foi batizado de *Colossus* e tinha como principal propósito quebrar outra cifra usada pelos nazistas durante a guerra, a *cifra de Lorenz*. A cifra de Lorenz é uma versão do que estudaremos com o nome de *cifra de fluxo*.

Para decifrar os códigos das máquinas Enigma e da cifra de Lorenz, os ingleses tiveram que contar não apenas com o texto criptografado que interceptavam sem grandes dificuldades, mas também com uma série de cifras cujas mensagens eles conheciam previamente. Veremos mais para frente a importância desta informação.

A capacidade dos aliados de decifrar as mensagens de seus adversários foi central para sua vitória na guerra. O trabalho de Turing e de sua equipe em Bletchley Park, utilizando máquinas como a Bombe e o Colossus, foi fundamental para quebrar as cifras alemãs e garantiu uma vantagem estratégica crucial aos Aliados.

O começo do século XX marcou o surgimento das primeiras máquinas de criptografar e das primeiras máquinas de criptoanálise. Na metade do século, começaram a surgir os primeiros computadores. Nos anos 70, a comunicação seria revolucionada pelo advento da internet, mas antes disso já ficara claro que era necessário compreender melhor o que faz uma cifra ser segura.

2.5 Exercício

Exercício 1. *Considere a seguinte mensagem:*

`privacidadepublicatranparenciaprivada`

- *Criptografe essa mensagem utilizando a cifra de deslocamento com* $k = 3$.
- *Criptografe essa mensagem utilizando a cifra de substituição com a seguinte permutação de letras:* `ZEBRASCDFGHIJKLMNOPQTUVWXY`

- *Criptografe essa mensagem utilizando a cifra de Vigenère com chave* `senha`.

Exercício 2. *Mostre que a operação de adição + modulo n é um* anel *para qualquer valor de n. Ou seja, para qualquer* $a, b, c, n \in \mathbb{Z}$ *temos que:*

- associatividade*:* $(a+b)+c \equiv a+(b+c)\ mod\ n$ *e* $(ab)c \equiv a(bc)\ mod\ n$
- elemento neutro*:* $a+0 \equiv a\ mod\ n$ *e* $a.1 \equiv a\ mod\ n$
- inverso*: existe* $-a$ *tal que* $a+(-a) \equiv 0\ mod\ n$
- distributividade*:* $a(b+c) \equiv ab+ac\ mod\ n$

Exercício 3. *Mostre que se* $n|a$ *e* $n|b$ *então* $n|(ra+sb)$ *para quaisquer* $r, s \in \mathbb{Z}$.

Exercício 4. *Dizemos que a é o* inverso multiplicativo *de b em* $\mathbb{Z}_n$ *sse* $ab \equiv 1\ mod\ n$.

- *Mostre que* 2 *é o inverso multiplicativo de* 5 *em* $\mathbb{Z}_9$.
- *Mostre que* 6 *não possui inverso multiplicativo em* $\mathbb{Z}_{12}$

Exercício 5. *Proponha um sistema de criptografia simétrica e argumente porque ele é mais seguro do que os sistemas que vimos até aqui.*

Exercício 6. *Calcule o tamanho do universo das chaves em uma cifra de Vigenère da forma como usada normalmente (escolhendo um palavra) e na forma como apresentamos formalmente (sequência aleatória com tamanho fixo l)?*

Exercício 7. *Construa um script que extraia um corpus do português moderno (por exemplo, textos da wikipedia) e calcule a frequência de ocorrência das letras do alfabeto.*

Exercício 8. *Em 2017 um rapaz que ficou conhecido como menino do Acre ficou dias desaparecido e deixou uma serie de livros criptografados com cifra de substituição em seu quarto. Na Figura 2.1 está reproduzida uma página de um desses livros. Utilize a análise de frequência para decifrar o texto.*

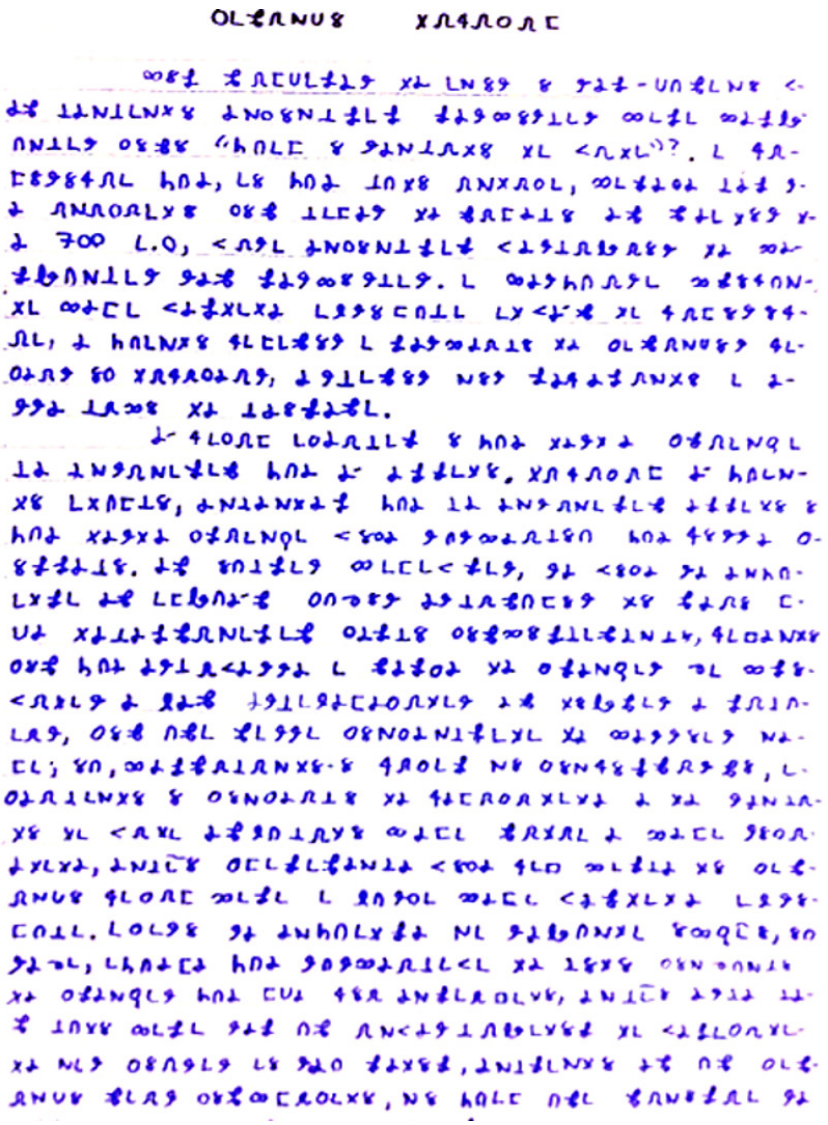


Figura 2.1: Texto criptografado pelo Menino do Acre

Capítulo 3

Criptografia Moderna

No final dos anos 40, com o desenvolvimento dos primeiros computadores e a experiência da quebra das cifras mecanicamente produzidas por poderosas máquinas desenvolvidas pelo esforço de guerra do nazismo, alguns cientistas se voltaram para um problema central no campo da criptografia: o que torna um sistema de criptografia seguro? As cifras que vimos até agora são conhecidas como *cifras clássicas* exatamente porque elas precedem esse debate moderno, e não é à toa que foram todas derrotadas cedo ou tarde. Informalmente, poderíamos dizer que o problema dos esquemas clássicos de criptografia é que eles mantêm muita informação sobre a mensagem original, como a frequência das letras, dos dígrafos, letras duplas, entre outros. Esses padrões permitem que criptanalistas descubram o texto original sem conhecer a chave.

Não é uma coincidência, portanto, que a primeira tentativa de formalizar o conceito de segurança tenha sido proposta por Claude Shannon, o fundador da teoria da informação. Shannon introduziu conceitos fundamentais que ajudaram a definir o que faz um sistema de criptografia ser verdadeiramente seguro. Sua abordagem foi a base para o desenvolvimento de métodos mais robustos e seguros de criptografia.

3.1 Sigilo Perfeito

Shannon definiu o que hoje chamamos de sigilo perfeito. Um sistema de criptografia garante o *sigilo perfeito* se a probabilidade da mensagem original for independente da probabilidade da cifra.

Vamos relembrar o que são eventos independentes. A probabilidade da mensagem original ser m dado que a cifra é c deve ser igual à probabilidade da mensagem ser m independentemente da cifra. Em termos matemáticos, isso significa que:

$$Pr[M = m|C = c] = Pr[M = m]$$

É nesse sentido preciso que dizemos que a cifra c não guarda nenhuma informação sobre a mensagem m. Quando essa condição é satisfeita, mesmo que um interceptador obtenha a cifra c, ele não obtém nenhuma pista sobre qual era a mensagem m. Todas as mensagens possíveis são igualmente prováveis, tornando a criptoanálise impossível. Ou seja, podemos dizer que um sistema de criptografia garante o sigilo perfeito se as cifras produzidas por ele não guardam informação da mensagem original.

De forma equivalente, um sistema de criptografia garante o sigilo perfeito se uma cifra c qualquer pode ter sido produzida com igual probabilidade por qualquer mensagem m. Dada uma cifra c e duas mensagens m_1 e m_2 quaisquer, temos que:

$$Pr[C = c|M = m_0] = Pr[C = c|M = m_1]$$

A cifra de substituição não garante o sigilo perfeito. Vamos ver um exemplo simples que mostra isso.

Exemplo 6. *Suponha que nossa cifra é* `OVO`*. Vamos calcular a probabilidade de diferentes mensagens originais terem gerado essa cifra.*

Se a mensagem original for `ana`*, e usamos uma cifra de substituição específica que mapeia* `a` *para* `O` *e* `n` *para* `V`*, a mensagem cifrada resultante seria* `OVO`*. Se a chave, que nesse caso é uma permutação, foi escolhida de forma aleatória, temos 1 chance em 23 que a letra* `a` *da mensagem original tenha sido substituída por* `O` *na cifra. A chance de* `n` *ser substituído por* `V` *é então 1 em 22, porque* `n` *não pode ser substituído por* `O` *que já foi escolhido. Portanto, a probabilidade da mensagem original* `ana` *gerar a cifra* `OVO` *é* $\frac{1}{22.23} = \frac{1}{650}$.

Agora, considere outra mensagem possível, como `eva`*. Se a letra* `e` *fosse substituída por* `O`*, isso não seria possível porque* `a` *também precisaria ser substituída por* `O`*. No entanto, na cifra de substituição, uma letra só pode ser mapeada para uma outra única letra, não duas vezes para a mesma letra. Portanto, a probabilidade de a mensagem original ser* `eva` *e gerar a cifra* `OVO` *é 0.*

Esse exemplo simples mostra mensagens distintas com probabilidades diferentes de gerar uma mesma cifra. Em outras palavras, a cifra de substituição não garante o sigilo perfeito, pois a probabilidade da cifra ser gerada não é independente da mensagem original.

Outra forma de definir o sigilo perfeito é por meio de um jogo entre o sistema de criptografia Π e um adversário que quer decifrá-lo. O jogo funciona da seguinte maneira:

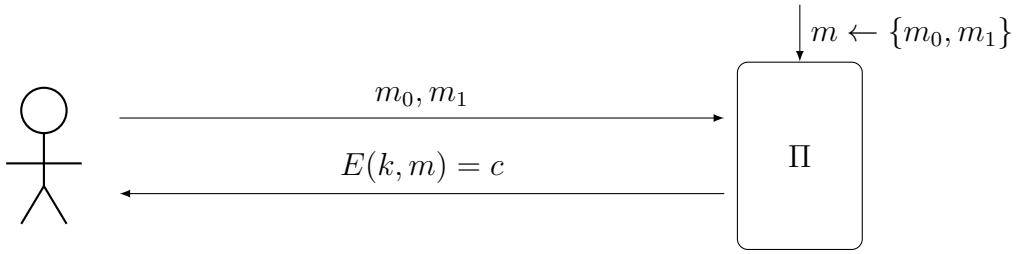
1. O adversário escolhe duas mensagens quaisquer m_0 e m_1. A única restrição é que elas devem ter o mesmo tamanho. O adversário então envia essas duas mensagens para o sistema.

2. O sistema então sorteia uma chave k usando o algoritmo Gen e escolhe aleatoriamente uma das duas mensagens.

3. O sistema criptografa a mensagem escolhida com a chave gerada ($E(k, m) = c$) e devolve a cifra resultante c para o adversário.

4. O adversário pode utilizar todos os meios ao seu dispor para descobrir qual das duas mensagens foi criptografada.

Um sistema de criptografia garante o sigilo perfeito se, ao final do jogo, o adversário não tiver nenhuma vantagem em descobrir qual mensagem foi criptografada.



Note que mesmo sem nenhuma informação, o adversário pode acertar qual das duas mensagens foi cifrada com 50% de probabilidade, simplesmente chutando uma das duas. O sistema Π garante sigilo perfeito se isso for o melhor que o adversário pode fazer. Ou seja, se a probabilidade de que ele vença o jogo for exatamente $1/2$.

Apresentar esse conceito como um jogo enfatiza que a única informação que o adversário possui é a cifra que foi gerada pelo sistema. Para que o

sistema garanta sigilo perfeito, o conhecimento sobre a cifra não deve dar nenhuma vantagem ao adversário para que ele consiga distinguir qual das mensagens foi criptografada.

3.2 One Time Pad

Temos agora uma definição formal de segurança. Vimos que a cifra de substituição não satisfaz essa definição, mas na verdade nenhuma das cifras clássicas a satisfaz. As cifras clássicas, como a cifra de deslocamente, a cifra de substituição e a cifra de Vigenère, possuem vulnerabilidades inerentes que permitem que adversários determinem informações sobre a mensagem original a partir da cifra.

Não seria desejável que essas cifras satisfizessem a definição de sigilo perfeito, pois vimos no capítulo anterior que nenhuma das cifras clássicas é segura. Todas elas podem ser derrotadas se o adversário tiver acesso a uma cifra de tamanho suficientemente grande. Por exemplo, a análise de frequência pode ser usada para quebrar cifras de substituição, enquanto técnicas como o método de Kasiski podem ser aplicadas para quebrar a cifra de Vigenère.

A definição de sigilo perfeito nos fornece um padrão elevado de segurança, onde a cifra não revela absolutamente nenhuma informação sobre a mensagem original. Este é um objetivo ideal para sistemas de criptografia, pois garantiria que mesmo um adversário com poder computacional ilimitado não pudesse descobrir a mensagem original a partir da cifra.

Ficamos então com o desafio de encontrar algum sistema que satisfaça essa definição, caso tal sistema exista. Esta busca nos leva a explorar técnicas e métodos mais avançados de criptografia.

O conceito do One-Time Pad (OTP), surgiu como uma solução inovadora no início do século XX. Em 1917, Gilbert Vernam, um engenheiro americano trabalhando para a AT&T, propôs uma cifra baseada em uma fita perfurada que se combinava com o texto claro utilizando a lógica binária. Essa ideia inicial, conhecida como a *cifra de Vernam*, já trazia uma inovação ao permitir a combinação de uma chave com o texto da mensagem, mas ainda não era totalmente segura, pois a repetição das chaves poderia levar à sua quebra.

A virada para a inviolabilidade da cifra veio com a contribuição de Joseph Mauborgne, um oficial do Corpo de Sinalização do Exército dos Estados Unidos. Mauborgne sugeriu uma modificação essencial: usar uma chave aleatória, tão longa quanto a própria mensagem, e garantir que essa chave

nunca fosse reutilizada. Essa chave, composta por uma sequência de números ou caracteres verdadeiramente aleatórios, era usada uma única vez e depois descartada. Essa abordagem eliminava qualquer padrão no texto cifrado que pudesse ser analisado por um adversário, garantindo que, mesmo que a cifra fosse interceptada, ela não revelaria absolutamente nenhuma informação sobre o texto original.

No que segue, apresentaremos um sistema chamado One Time Pad, e mostraremos que ele garante o sigilo perfeito, o que historicamente ocorreu nos anos 40 com os estudos de Shannon. A partir deste ponto, conforme começarmos a investigar sistemas a serem implementados computacionalmente, consideraremos que o universo das mensagens (assim como o universo das cifras) será representado não mais como sequências de letras, mas como sequências de bits.

No caso específico do OTP, assumiremos que as mensagens e as cifras possuem um tamanho fixo. Mais importante é o fato de que o universo das chaves é também um conjunto de sequências de bits *do mesmo tamanho*. Em outras palavras, tanto as mensagens quanto as chaves e as cifras serão representadas por sequências de n bits.

O OTP é o seguinte sistema de criptografia:

- *Gen* sorteia uma chave k de n bits.
- Para *criptografar* uma mensagem m, que também é uma sequência de n bits, realizamos uma operação bit a bit (bitwise) chamada de XOR (ou exclusivo) entre a mensagem m e a chave k:

$$c = k \oplus m$$

- Para *descriptografar* a cifra c, aplicamos novamente a operação XOR entre a cifra c e a mesma chave k:

$$k \oplus c$$

A operação XOR entre dois bits é definida como:

$$\begin{aligned} 0 \oplus 0 &= 0 \\ 1 \oplus 0 &= 1 \\ 0 \oplus 1 &= 1 \\ 1 \oplus 1 &= 0 \end{aligned}$$

Ou seja, o resultado da operação é 0 se os bits forem iguais e 1 se eles forem diferentes.

A operação XOR entre sequências de bits possui algumas propriedades que serão úteis. Deixamos como exercício mostrar essas quatro propriedades da operação:

(idempotência) $m \oplus m = 0$

(identidade) $m \oplus 0 = m$

(comutatividade) $m \oplus n = n \oplus m$

(associatividade) $(m_1 \oplus m_2) \oplus m_3 = m_1 \oplus (m_2 \oplus m_3)$

Essas propriedades são suficientes para mostrar que o sistema é correto. Ou seja, se criptografamos uma mensagem (m) produzindo uma cifra ($E(k, m) = c$) e depois descriptografamos essa cifra (c) com a mesma chave $D(k, c)$ recuperamos a mensagem original.

$E(k, m) = k \oplus m = c$

$D(k, c) = k \oplus c = k \oplus (k \oplus m)$

$D(k, c) = (k \oplus k) \oplus m$ por associatividade

$D(k, c) = 0 \oplus m$ por idempotência

$D(k, c) = m$ por identidade

Exemplo 7. *Considere uma mensagem* $m = 101010$ *e uma chave* $k = 010001$*. Usando o sistema One Time Pad a cifra produzida é a seguite:*

$$\begin{array}{ccccc} m & \oplus & k & = & c \\ 101010 & \oplus & 010001 & = & 111011 \end{array}$$

Como antecipado, é possível, e relativamente simples provar que o OTP possui sigilo perfeito.

Teorema 1. *O sistema de criptografia* One Time Pad *possui sigilo perfeito.*

Demonstração. Para provar o teorema note dada uma cifra c que esiste uma única chave k tal que $E(k, m = c)$, a saber $k = c \oplus m$.

Agora, a probabilidade de escolher essa chave específica entre todas as possíveis chaves (sequências de bits de comprimento n) é $\frac{1}{2^n}$.

Isso significa que a probabilidade de obter a cifra c a partir de qualquer mensagem m é a mesma, independente de qual mensagem foi escolhida originalmente. Essa probabilidade é sempre $\frac{1}{2^n}$.

Portanto, a cifra c é igualmente provável para qualquer mensagem m. Em outras palavras, o conhecimento da cifra c não fornece ao adversário nenhuma informação adicional sobre qual mensagem foi originalmente enviada, garantindo assim o sigilo perfeito. □

O One Time Pad (OTP) possui duas severas limitações que afetam sua aplicação prática.

A primeira limitação é indicada pelo próprio nome do sistema. O OTP pressupõe que a chave de criptografia k seja *usada exatamente uma vez*. Isso significa que cada chave deve ser única e exclusiva para uma única mensagem. Se a mesma chave k for reutilizada para criptografar duas mensagens distintas m_1 e m_2, o sistema se torna completamente inseguro.

Para ilustrar essa limitação considere que duas cifras c_0 e c_1 foram produzidas usando a mesma chave k. Assim temos que $c_0 = k \oplus m_0$ e $c_1 = k \oplus m_1$. Note o que acontece quando aplicamos o ou exclusivo entre as duas cifras eliminamos a chave:

$$\begin{aligned} c_0 \oplus c_1 &= (k \oplus m_0) \oplus (k \oplus m_1) \\ &= (k \oplus k) \oplus (m_0 \oplus m_1) \\ &= m_0 \oplus m_1 \end{aligned}$$

Isso significa que o adversário pode obter a operação XOR das duas mensagens originais, mesmo sem conhecer a chave k.

Com essa informação, o adversário pode realizar uma análise de frequência ou outras técnicas de criptoanálise para tentar deduzir as mensagens originais, especialmente se as mensagens tiverem algum padrão ou conteúdo previsível. Esse tipo de ataque é conhecido como *ataque de reutilização de chave*, e mostra que a segurança do OTP depende criticamente do uso único de cada chave.

Portanto, a reutilização de uma chave no OTP compromete a segurança do sistema, transformando-o de um sistema teoricamente perfeito em um sistema vulnerável e facilmente quebrável.

A segunda e mais crítica limitação do OTP é o tamanho de sua chave. A suposição que fizemos é que *o tamanho da chave deve ser tão grande quanto a mensagem a ser cifrada*. Há uma série de problemas práticos com isso. Computacionalmente, não é possível gerar chaves aleatórias muito grandes, o que limita o tamanho das mensagens que podemos cifrar.

Além disso, assumimos que as chaves são compartilhadas entre as partes. Deixamos os detalhes sobre a distribuição de chaves para o Capítulo 9, mas por ora podemos adiantar que se nossa chave é tão grande quanto a mensagem, por que não enviamos a mensagem pelo mesmo canal que enviaríamos a chave? Esse é um ponto crítico, pois, se conseguimos enviar a chave de forma segura, poderíamos, em teoria, enviar a mensagem diretamente pelo mesmo canal, tornando o uso da chave redundante.

Enfim, um sistema cuja chave seja tão grande quanto a mensagem é de muito pouca utilidade prática. A dificuldade de gerar, distribuir e gerenciar chaves de tamanho adequado para cada mensagem limita significativamente a aplicabilidade do OTP em contextos reais.

Encerramos este capítulo mostrando que esta segunda limitação do OTP infelizmente não é uma peculiaridade do sistema. Na verdade todo sistema que possua sigilo perfeito está fadado a ter chaves tão grandes ou maiores do que a mensagem. Esse resultado negativo foi proposto e demonstrado pelo próprio Shannon ainda nos anos 40.

Teorema 2 (Shannon)**.** *Se um sistema que garante o sigilo perfeito, então seu universo de chaves deve ser pelo menos tão grande quanto seu universo de mensagens, ou seja, o tamanho da chave deve ser maior ou igual ao tamanho da mensagem.*

Demonstração. Consideremos M_c como o conjunto de todas as mensagens que podem produzir a cifra c.

Note que se uma mesma chave pudesse produzir c a partir de mensagens diferentes, a descriptografia não funcionaria corretamente, pois não poderíamos determinar qual mensagem foi originalmente cifrada. Portanto, o tamanho de M_c deve ser menor ou igual ao número de chaves disponíveis.

Agora, imagine que o número de chaves seja estritamente menor do que o número de mensagens. Teríamos então $|M_c| \leq |K| < |M|$. Nesse caso,

haveria pelo menos uma mensagem m que não estaria em M_c, ou seja, uma mensagem que não pode gerar a cifra c.

Isso significa que a probabilidade das mensagens não seria independente da probabilidade das cifras. Para essa mensagem m, teríamos:

- A probabilidade de escolher m não seria zero: $Pr[M = m] \neq 0$
- A probabilidade de m produzir c seria zero: $Pr[M = m|C = c] = 0$

Ou seja, se houver mais mensagens do que chaves, haveria mensagens que não poderiam produzir certas cifras, tornando a probabilidade das mensagens dependente das cifras. Portanto, para garantir o sigilo perfeito, o número de chaves deve ser pelo menos tão grande quanto o número de mensagens. □

Durante a Guerra Fria, o One-Time Pad (OTP) foi utilizado para proteger comunicações altamente sensíveis entre as superpotências. Um dos exemplos mais conhecidos foi o "telefone vermelho", a linha direta que conectava a Casa Branca e o Kremlin, usada para evitar conflitos nucleares por meio de uma comunicação segura e rápida. O OTP foi escolhido para cifrar essas conversas porque sua chave aleatória e única para cada mensagem tornava impossível a interceptação e decifração por parte de espiões ou adversários. A complexidade e a segurança incomparável do OTP tornaram-no uma escolha ideal para momentos críticos, como a Crise dos Mísseis em Cuba, onde a confidencialidade era absolutamente vital.

3.3 Criptografia Moderna

A definição de segurança proposta por Claude Shannon foi a primeira tentativa séria de formalizar a segurança dos sistemas de criptografia. No entanto, o próprio Shannon demonstrou as limitações dessa definição.

Apesar do fracasso desta primeira tentativa de formalização, não abandonaremos a ideia geral. A abordagem da *criptografia moderna* [GM84], que utilizaremos nesta apostila, segue três princípios básicos:

1. *definições formais:* As noções de segurança utilizadas serão apresentadas de maneira formal por meio de definições claras. Essas definições nos ajudam a comparar diferentes sistemas de criptografia e avaliar sua segurança com base em critérios previamente estabelecidos.

2. *suposições explícitas:* Muitas vezes, precisamos fazer suposições sobre os sistemas de criptografia que não podemos provar. Essas suposições devem ser explicitadas de maneira clara e formal. Mesmo sem provas definitivas, podemos validar essas suposições empiricamente. Grande parte do trabalho na criptografia moderna envolve testar essas suposições e buscar as mais simples e básicas.

3. *demonstrações formais:* Quando conseguimos formalizar nossas suposições e definir a segurança desejada, podemos eventualmente provar que um sistema que atende a essas suposições garante uma certa noção de segurança. Esse tipo de demonstração reduz o problema da segurança às suposições do sistema, que devem ser mais simples e mais fáceis de validar empiricamente. Essa abordagem permite substituir um sistema se suas suposições forem refutadas, antes que ele seja quebrado.

As definições de segurança em criptografia geralmente possuem dois componentes:

Garantia de Segurança: Especifica o que pode ser considerado um ataque bem-sucedido.

Modelo de Ameaças: Define o que o adversário pode ou não pode fazer.

Por exemplo, na definição de sigilo perfeito, a garantia de segurança é que nenhuma informação sobre a mensagem esteja contida na cifra. Isso significa que a probabilidade de ocorrência da cifra deve ser independente da probabilidade de ocorrência da mensagem. O modelo de ameaças assume que o adversário tem acesso apenas ao texto cifrado e nada mais. Este modelo de ameaça é destacado na definição que demos, simulando um jogo entre o sistema de criptografia e um adversário. No jogo, o adversário tenta adivinhar qual mensagem foi cifrada com base apenas na cifra recebida, sem qualquer informação adicional.

Os modelos de ameaças que estudaremos na apostila incluem:

Ataque apenas ao texto cifrado (ciphertext-only): Este é o modelo assumido na definição de sigilo perfeito. Nele, supomos que o adversário tem acesso apenas a um texto cifrado de tamanho arbitrário.

Ataque com escolha de texto plano (ataque chosen-plaintext): Neste modelo, além de assumir que o adversário tem acesso à cifra, supomos

que ele é capaz de escolher algumas mensagens e ver como elas seriam cifradas pelo sistema. Isso dá ao adversário mais informações para tentar descobrir a chave de criptografia ou a mensagem original.

Ataque com escolha de texto cifrado (ataque chosen-ciphertext): Neste modelo, assumimos que o adversário é capaz de escolher certas cifras e ver como elas seriam decifradas pelo sistema. Esse é um modelo de ameaça mais poderoso, pois permite ao adversário interagir com o sistema de criptografia de maneira mais direta e potencialmente descobrir vulnerabilidades.

Esses modelos de ameaças são progressivamente mais fortes, o que significa que cada um assume uma capacidade de ataque cada vez maior por parte do adversário. É importante notar que nossos modelos de ataque não fazem suposições sobre a estratégia específica que o adversário pode usar. Eles simplesmente definem que informações assumimos que o adversário pode acessar. Isso nos permite avaliar a segurança de um sistema de maneira abrangente e rigorosa, sem depender de suposições específicas sobre o comportamento do adversário.

Afirmamos que o sigilo perfeito é garantido se nenhum adversário que só tem acesso a cifra for capaz de distinguir qual mensagem foi encriptada pelo sistema. Mostramos que essa definição exige que o número de chaves seja pelo menos tão grande quanto o número de mensagens, o que não é prático.

Para contornar essas limitações, enfraqueceremos a garantia de segurança de duas formas:

Adversário eficiente: Assumiremos que o adversário deve ser eficiente, ou seja, ele deve usar sua estratégia em um tempo razoável. Na prática, isso significa que, embora um adversário possa eventualmente derrotar o sistema se lhe for dado tempo suficiente. A ideia é que, se o tempo necessário para o adversário quebrar o sistema é impraticavelmente longo, o sistema ainda pode ser considerado seguro.

Probabilidade pequena: Permitiremos que o adversário possa eventualmente derrotar o sistema, mas apenas com uma probabilidade muito pequena. Em outras palavras, a cifra pode conter alguma informação sobre a mensagem original, desde que essa quantidade de informação seja insignificante. Isso significa que, embora não possamos garantir que a cifra seja completamente independente da mensagem, a informação

que pode ser extraída deve ser tão pequena que não ofereça uma vantagem significativa ao adversário.

Ao enfraquecer a definição de segurança dessa maneira, movemos de um ideal teórico inatingível para uma abordagem mais prática que ainda oferece um grau satisfatório de segurança.

3.4 Abordagem Assintótica

Em termos práticos, buscamos uma noção de segurança que imponha requisitos severos ao sucesso de um adversário. Especificamente, queremos que um adversário precise rodar seu algoritmo por um *intervalo de tempo muito grande* para ter sucesso. Além disso, esperamos que o adversário possa eventualmente derrotar o sistema, mas com uma *probabilidade extremamente baixa*.

Intervalo bem grande de tempo: O que consideramos um tempo razoavelmente longo para resistir a ataques muda com o tempo. O que era considerado seguro há uma década pode não ser mais seguro hoje. Assim, definir um intervalo de tempo "bem grande" é complicado, pois deve se ajustar ao ritmo acelerado de avanços tecnológicos.

Probabilidade muito baixa: Determinar uma probabilidade de sucesso muito baixa para um ataque também é desafiador. Uma probabilidade aceitável hoje pode não ser aceitável amanhã à medida que as técnicas de ataque evoluem. Além disso, a definição de "muito baixa" deve ser precisa e quantificável, o que pode ser difícil de estabelecer de forma universal.

Em vez de estabelecer valores fixos para esses limites, adotamos uma *abordagem assintótica*, como fazemos ao analisar algoritmos ou avaliar a complexidade de um problema de computação. Vamos supor que, ao gerar a chave, o algoritmo Gen recebe um parâmetro de segurança n e estabelecemos nossos limites em função deste parâmetro.

Para os efeitos desta apostila, podemos assumir que o parâmetro está relacionado ao tamanho da chave a ser gerada e que é de conhecimento público. O parâmetro de segurança permite que as partes ajustem seu sistema para

o nível desejado de segurança. Aumentar o parâmetro de segurança geralmente resulta em um aumento no tempo de processamento do sistema e no tamanho da chave.

A capacidade de ajustar o parâmetro de segurança tem vantagens práticas. Ela permite que as partes defendam seus sistemas contra adversários com poder computacional mais forte à medida que o tempo passa e a tecnologia avança.

Com essa abordagem, podemos definir de maneira mais clara nossas garantias de segurança. Vamos exigir que nenhum adversário rodando um *algoritmo polinomial* em relação ao tamanho do parâmetro de segurança n seja capaz de distinguir as mensagens, exceto com uma probabilidade desprezivelmente maior que $1/2$. Por "desprezivelmente maior", queremos dizer que esse valor cresce mais lentamente do que qualquer polinômio em relação a n.

Em termos práticos, isso significa que, *conforme aumentamos o parâmetro de segurança, o tempo de processamento do adversário se torna impraticável e sua probabilidade de sucesso se torna insignificante.* Essa abordagem nos permite criar sistemas que são adaptáveis e capazes de garantir a segurança, mesmo diante de adversários com crescente poder computacional.

Exemplo 8. *Considere um ataque força-bruta contra uma chave de n bits. Neste caso o universo de chaves teria tamanho 2^n e o valor esperado do tempo do ataque força-bruta seria 2^{n-1}. Assim o tempo de processamento desse ataque cresce exponencialmente, ou seja, mais rápido do que qualquer polinômio em relação a n. A probabilidade de se chutar uma chave encontrar a chave correta é 2^{-n}. Assim, essa probabilidade descrece mais rápido do que qualquer polinômio.*

Ou seja, sendo n o parâmetro de segurança, um ataque força-bruta não viola a definição de segurança porque esse tipo de ataque toma tempo exponencial e a probabilidade de sucesso é desprezível.

3.5 Exercícios

Exercício 9. *Mostre que o 0 é elementro neutro na operação $\oplus$, ou seja, que para todo $x \in \{0,1\}^*$ temos que $x \oplus 0 = 0 \oplus x = x$.*

Exercício 10. *Mostre que a operação $\oplus$ é* associativa*, ou seja, que para todo $x, y, z \in \{0,1\}^*$ temos que $x \oplus (y \oplus z) = (x \oplus y) \oplus z$.*

Exercício 11. *Mostre que a operação* $\oplus$ *é* comutativa*, ou seja, que para todo* $x, y \in \{0,1\}^*$ *temos que* $x \oplus y = y \oplus x$*.*

Exercício 12. *Mostre que para qualquer sequência de bits* $x \in \{0,1\}^*$ *temos que* $x \oplus x = 0$*.*

Exercício 13. *Sejam* ε_1 *e* ε_2 *duas funções desprezíveis. Mostre que* $\varepsilon_1 + \varepsilon_2$ *é desprezível.*

Exercício 14. *Seja* ε *uma função desprezível e* p *um polinômio. Mostre que* $p\varepsilon$ *é desprezível.*

Exercício 15. *Quais as vantagens da abordagem assintótica na definição da garantia de segurança?*

Exercício 16. *Por que representamos o parâmetro de segurança em notação unária?*

Exercício 17. *Considere um sistema* Π *seguro contra ataques* ciphertext only *cujo parâmetro de segurança tem* 128 *bits (*$n = 128$*) e um adversário polinomial que derrota o sistema com probabilidade* $\frac{1}{2} + \frac{1}{2^{n/4}}$*. Com que probabilidade esse adversário derrotaria o sistema se dobrassemos* n*?*

Exercício 18. *Considere um sistema* Π *seguro contra ataques* ciphertext only *cujo parâmetro de segurança tem* 128 *bits (*$n = 128$*) e um adversário que roda um algoritmo que derrota o sistema com probabilidade* 1 *em* $2^{n/8}$ *passos. Quantos passos seriam necessários para esse algoritmo derrotar um sistema se dobrássemos* n*? E se quadrupicássemos* n*?*

Capítulo 4

Cifras de Fluxo

No capítulo anterior, apresentamos uma definição formal para segurança contra ataques em que o adversário tem acesso apenas ao texto cifrado. Neste capítulo, vamos apresentar uma forma de construir um sistema que satisfaz essa definição.

A ideia geral da construção é a seguinte: partimos de uma sequência aleatória de bits chamada de semente e, a partir dela, geramos uma sequência maior de bits para encriptar a mensagem usando a operação de OU exclusivo (XOR), assim como no OTP. Embora essa sequência seja gerada de maneira determinística, a segurança do sistema depende do fato de que ela se pareça aleatória.

Informalmente, um *gerador de números pseudoaleatórios* (GNP) é uma função que recebe uma semente aleatória e a expande em uma sequência maior que aparenta ser aleatória. A qualidade de um GNP é medida pela sua capacidade de produzir sequências que são indistinguíveis de sequências verdadeiramente aleatórias para qualquer adversário eficiente.

Sistemas de cifra de fluxo foram estudados extensamente nos anos 80 [BM84, Yao82]. A abordagem para verificar se o gerador de números pseudoaleatórios é suficientemente forte consistia em aplicar uma série de testes estatísticos na sequência gerada para tentar distingui-la de uma sequência aleatória.

Por exemplo, um teste pode verificar se a probabilidade de o primeiro bit da sequência ser igual a 1 é 1/2. Isso significa que, em uma sequência verdadeiramente aleatória, a chance de o primeiro bit ser 1 deve ser igual à chance de ser 0, ou seja, 50%. O teste verifica se essa propriedade é mantida na sequência gerada pelo GNP. Se a sequência gerada tiver uma probabilidade

significativamente diferente de 1/2 para o primeiro bit ser 1, isso pode indicar que a sequência não é suficientemente aleatória.

Outro exemplo de teste é verificar se a probabilidade de ocorrência de pelo menos três 0s em qualquer subsequência de tamanho 4 é 5/16. Para entender por que essa é a probabilidade esperada, consideremos todas as possíveis subsequências de 4 bits:

- Existem exatamente 1 subsequência onde todos os bits são 0s (0000).
- Existem 4 subsequências onde três dos quatro bits são 0s (0001, 0010, 0100, 1000).

Portanto, há 5 subsequências que atendem ao critério (1 com quatro 0s e 4 com três 0s) em um total de $2^4 = 16$ possíveis subsequências de 4 bits. Assim, a probabilidade de ocorrência de pelo menos três 0s em uma subsequência de 4 bits é 5/16.

O teste então verifica se a sequência gerada pelo GNP possui essa distribuição específica de 0s em suas subsequências de 4 bits. Se a sequência gerada pelo GNP passar neste teste, isso sugere que a sequência é indistinguível de uma sequência verdadeiramente aleatória em relação a essa propriedade específica.

Devido ao seu papel em diferenciar sequências pseudoaleatórias de sequências verdadeiramente aleatórias, esses testes são chamados de *distinguidores*. Os distinguidores são essenciais para avaliar a robustez de um GNP, pois nos ajudam a garantir que as sequências geradas são adequadamente aleatórias para serem usadas em sistemas de criptografia seguros.

Uma bateria de distinguidores pode ser usada para verificar a qualidade de um GNP. Idealmente, nenhum teste eficiente deveria ser capaz de distinguir a sequência gerada por um GNP de uma sequência verdadeiramente aleatória, ou pelo menos, deveria ser incapaz de fazê-lo com uma probabilidade considerável[1].

Definimos um gerador de números pseudo-aleatórios (GNP) como um algoritmo G que recebe uma semente s de um determinado tamanho n e produz uma sequência maior de bits de tamanho $l(n)$. O fator de expansão do GNP, representado pela função l, indica quanto maior é a sequência produzida em comparação com a semente inicial.

[1] Usaremos o termo "considerável" para probabilidades não desprezíveis.

Para que um GNP seja considerado seguro, a sequência gerada a partir da semente deve ser indistinguível de uma sequência verdadeiramente aleatória. Em outras palavras, não deve ser possível para nenhum algoritmo eficiente distinguir entre a sequência gerada pelo GNP e uma sequência aleatória com uma probabilidade significativa.

Em outras palavras, um GNP é seguro se para todo algoritmo polinomial D (distinguidor), a diferença na probabilidade de D identificar corretamente a origem da sequência ($|Pr[D(r) = 1] - Pr[D(G(s)) = 1]|$) for desprezível.

A restrição de que o distinguidor seja eficiente é estritamente necessária. Isso se deve ao fato de que, em teoria, sempre é possível construir um distinguidor usando uma espécie de ataque de força bruta. Vamos explorar como isso funciona.

Imagine que temos uma sequência de bits w e queremos saber se ela foi gerada por um GNP G se é uma sequência verdadeiramente aleatória. Um algoritmo D, nosso distinguidor, pode realizar a seguinte estratégia:

1. Para cada possível semente s, o algoritmo D verifica se $G(s) = w$.
2. Se encontrar uma semente s tal que $G(s) = w$, D devolve 1 (indicando que a sequência foi gerada pelo GNP).
3. Se não encontrar nenhuma semente que gere w, D devolve 0 (indicando que a sequência é aleatória).

Para uma semente de n bits, existem 2^n possíveis combinações. Como apenas uma dessas combinações de s gera w corretamente, a chance de D encontrar essa combinação é $\frac{1}{2^n}$.

Além disso, esse teste é que ele não é eficiente. Para testar todas as possíveis sementes s, o tempo esperado é exponencial, mais precisamente 2^{n-1}. Esse tempo de execução torna o método impraticável para valores grandes de n. Portanto, embora seja teoricamente possível distinguir sequências geradas por um GNP de sequências aleatórias usando força bruta, na prática, a restrição de que o distinguidor seja eficiente é essencial para garantir que o GNP seja útil e seguro.

4.1 Segurança das Cifras de Fluxo

Como adiantamos no capítulo anterior, uma vez definida claramente a suposição que estamos fazendo, podemos tentar provar que, com essa suposição,

somos capazes de construir um sistema seguro. A abordagem para este tipo de prova é uma redução, similar às reduções que vimos em Teoria da Computação. Neste caso, vamos reduzir o problema de construir um sistema seguro contra ataques contra o texto cifrado apenas (*ciphertext-only*) ao problema de construir um gerador de números pseudo-aleatórios (GNP).

Reduções são ferramentas poderosas em teoria da computação e criptografia. Elas nos permitem transformar um problema desconhecido ou difícil em um problema conhecido ou mais simples. Se conseguirmos mostrar que resolver um problema implica resolver outro problema, estabelecemos uma relação de dependência entre eles. Neste contexto, se pudermos construir um GNP seguro, podemos utilizar esse GNP para construir um sistema de criptografia que seja seguro contra ataques ciphertext-only.

O processo de redução pode ser descrito da seguinte maneira:

Definir a Suposição: Primeiro, definimos claramente a suposição que estamos fazendo, no caso, a existência de um GNP seguro. Um GNP seguro é um gerador que produz sequências de bits que não podem ser distinguidas de sequências verdadeiramente aleatórias por qualquer algoritmo eficiente.

Transformar o Problema: Em seguida, transformamos o problema de construir um sistema seguro contra ataques ciphertext-only em um problema que envolve o GNP. Mostramos que, se temos um GNP seguro, podemos usar esse GNP como base para criar um sistema de criptografia que resista a ataques ciphertext-only.

Prova de Segurança: Finalmente, provamos que o sistema de criptografia resultante é seguro sob a suposição de que o GNP é seguro. Isso geralmente envolve demonstrar que qualquer ataque bem-sucedido ao sistema de criptografia implicaria em um ataque bem-sucedido ao GNP, o que contradiria a suposição de que o GNP é seguro.

Essa abordagem de redução nos permite construir sistemas de criptografia confiáveis baseando-nos em componentes cuja segurança já foi estabelecida ou assumida. Ela também facilita a compreensão das relações entre diferentes problemas de segurança e a reutilização de soluções existentes para resolver novos problemas.

Ao estabelecer essa conexão entre GNPs e sistemas de criptografia, criamos uma base sólida para desenvolver e analisar algoritmos de criptografia

modernos, garantindo que nossas construções sejam robustas e confiáveis sob as suposições corretas.

A *cifra de fluxo* é um método de criptografia que converte texto claro m em texto cifrado c, utilizando um fluxo de chaves pseudoaleatórias gerado por um gerador de números pseudoaleatórios (GNP). Este método é amplamente utilizado devido à sua eficiência e capacidade de operar em dados de tamanho variável, tornando-o ideal para aplicações como comunicação em tempo real e transmissão de dados.

O processo para criptografar uma mensagem m usando uma cifra de fluxo é o seguinte:

- Primeiro, uma semente aleatória k de tamanho fixo n é gerada. Esta semente deve ser segura e conhecida apenas pelas partes autorizadas. Ela será a chave do sistema.

- A semente é fornecida ao GNP G, que expande a semente em um fluxo de chaves $G(k)$ de comprimento adequado para criptografar a mensagem inteira.

- A mensagem original (texto claro) é combinada bit a bit (ou byte a byte) com o fluxo de chaves usando a operação XOR. O resultado é o texto cifrado.

$$c = m \oplus G(k)$$

Para descriptografar o processo é similar:

- A mesma semente k utilizada para criptografar a mensagem é novamente fornecida ao GNP G para gerar o mesmo fluxo de chaves $G(k)$.

- O texto cifrado é combinado bit a bit (ou byte a byte) com o fluxo de chaves usando a operação XOR.

$$c \oplus G(k)$$

  O resultado é a mensagem original.

Teorema 3. *Uma cifra de fluxo para uma mensagem de tamanho fixo n que utiliza um gerador de número pseudoaleatório para expandir a chave é seguro contra ataques* ciphertext only*.*

Demonstração. Para provar o teorema vamos mostrar que se existisse um adversário capaz de quebrar nosso sistema de criptografia, poderíamos usar esse adversário para criar um teste eficiente (distinguidor) que diferencie entre uma sequência gerada por um GNP e uma sequência aleatória.

Vamos supor que temos um adversário que pode quebrar nosso sistema de criptografia. Isso significa que, dado um texto cifrado, essa adversário consegue determinar informações sobre a mensagem original com uma precisão melhor do que a pura sorte.

Usamos o adversário para construir um distinguidor D. Este distinguidor funciona da seguinte forma:

1. D recebe uma sequência w e precisa determinar se w foi gerada por um GNP ou se é uma sequência verdadeiramente aleatória.

2. D roda o mesmo algoritmo do adversário para obter duas mensagens m_0 e m_1.

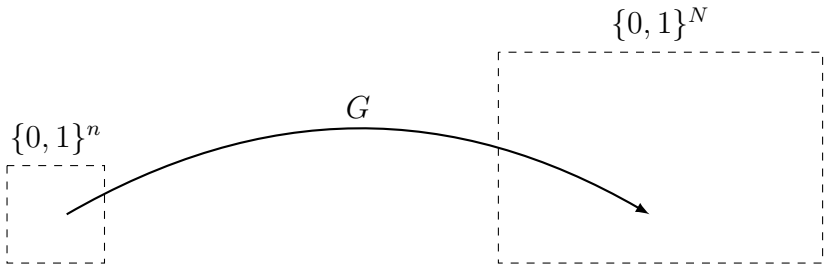
3. D escolhe aleatoriamente uma das duas mensagens m e calcula $c = w \oplus m$.

4. D entrega c ao adversário e devolve 1 se ele adivinhar corretamente qual das mensagens foi cifrada e 0 caso contrário.

O distinguidor D que acabamos de construit devolve 1 com a mesma probabilidade com que o adversário consegue quebrar o sistema. Se w é uma sequência verdadeiramente aleatória, o adversário não terá qualquer vantagem e só poderá adivinhar corretamente com uma probabilidade de 50%.

Se o adversário pode quebrar o sistema de criptografia, o distinguidor D pode usar essa capacidade para diferenciar entre uma sequência gerada por um GNP e uma sequência verdadeiramente aleatória. Isso mostraria que o GNP não é seguro, pois podemos construir um distinguidor eficiente que o quebra.

Ou seja, se o GNP é seguro, então não existe adversário eficiente capaz de derrotar o sistema. □



Algoritmos validados empiricamente como bons geradores de números pseudo-aleatórios (GNP) podem, portanto, ser utilizados para gerar cifras seguras, para ataques apenas contra o texto cifrado. Essa segurança se baseia na dificuldade que qualquer adversário eficiente teria em distinguir a saída do GNP de uma sequência verdadeiramente aleatória.

Em outras palavras, um gerador de números pseudoaleatórios (GNP) é uma função G que, a partir de um elemento aleatório escolhido no conjunto $\{0,1\}^n$, produz deterministicamente um elemento no conjunto maior $\{0,1\}^N$, onde N é muito maior que n. O gerador é considerado seguro se um adversário não for capaz de distinguir a saída de G de uma escolha verdadeiramente aleatória feita diretamente no conjunto maior $\{0,1\}^N$.

Nos próximos capítulos, vamos explorar definições mais robustas de segurança e as suposições associadas a essas definições. Analisaremos como essas definições mais fortes podem nos proteger contra adversários com capacidades adicionais e em diferentes cenários de ataque, como os ataques de texto claro escolhido (chosen-plaintext) e ataques de texto cifrado escolhido (chosen-ciphertext).

Antes de concluir este capítulo, é importante lembrar que as cifras de fluxo compartilham uma limitação significativa com a cifra de uso único (One-Time Pad, OTP): a repetição do uso da mesma chave. Assim como no OTP, se a mesma chave for reutilizada para criptografar diferentes mensagens, a segurança do sistema é comprometida. Isso ocorre porque a reutilização da chave permite que um adversário determine informações sobre as mensagens originais ao analisar as cifras resultantes.

Para mitigar esse problema, é crucial garantir que cada chave utilizada seja única e nunca repetida. Além disso, a gestão segura das chaves e a geração de chaves verdadeiramente aleatórias ou suficientemente pseudoaleatórias são aspectos fundamentais para manter a segurança das cifras de fluxo.

4.2 Modos de Operação

O Teorema 3 supõe que a mensagem a ser criptografada tem um tamanho fixo previamente conhecido. No entanto, em muitas aplicações práticas, precisamos criptografar mensagens de tamanho arbitrário. Para resolver esse problema, existem dois modos de operação para cifras de fluxo: o modo síncrono e o modo assíncrono.

No modo síncrono, o gerador de chaves (GNP) gera um fluxo contínuo de bits independente do texto claro. Já no modo assíncrono, o fluxo de chaves depende tanto da chave inicial quanto do texto cifrado anterior, permitindo uma ressincronização automática em caso de perda de dados.

Tipicamente, um gerador de números pseudo-aleatórios (GNP) é composto por dois algoritmos:

- O primeiro algoritmo, $Init$, recebe como entrada uma semente s e, opcionalmente, um *vetor inicial* VI. Ele devolve um estado inicial st_0.
- O segundo algoritmo, $GenBits$, recebe o estado atual st_i como entrada, gera um bit y e atualiza o estado para st_{i+1}.

Assim, para cada semente s, o GNP produz um fluxo contínuo de bits. Os primeiros $l(n)$ bits desse fluxo formam $G(s)$, a sequência pseudo-aleatória desejada.

No modo *síncrono*, tratamos uma sequência de mensagens $m_0, m_1, \ldots$ como uma única grande mensagem m, dividida em pedaços menores. O processo funciona da seguinte maneira:

1. O algoritmo $Init$ recebe a semente s gera o estado inicial st_0.
2. Cada vez que uma nova parte da mensagem (m_i) é enviada, o algoritmo $GenBits$ usa o estado atual para gerar os bits necessários para encriptar essa parte da mensagem.
3. O estado é então atualizado para st_{i+1} após cada comunicação.

A limitação desse modo é que ambas as partes, Alice e Bob, precisam manter seus estados sincronizados. Isso significa que, quando Alice atualiza o estado st, Bob também precisa atualizar seu estado correspondente de forma sincronizada.

O outro modo é *assíncrono*. Nesse modo, além da semente s, o algoritmo G recebe uma sequência de bits chamada *vetor inicial* (VI). O vetor inicial não é sigiloso, mas deve ser alterado a cada nova mensagem criptografada.

1. O algoritmo $Init$ recebe a semente s e o vetor inicial VI para gerar estado inicial st_0.

2. Cada vez que uma nova parte da mensagem (m_i) é enviada, o algoritmo $GenBits$ um novo vetor inicial VI é gerado. Esse vetor inicial é público e pode ser transmitido junto com o texto cifrado sem comprometer a segurança do sistema. O acrescimo do vetor inicial garante que o algoritmo não repita a mesma chave para criptografar pedaços diferentes da mensagem.

Ao garantir que o vetor inicial VI seja diferente para cada pedaço da mensagem, evitamos a repetição de chaves. Isso é crucial para impedir ataques onde um adversário poderia identificar padrões entre diferentes mensagens cifradas com a mesma chave.

Diferentemente do modo síncrono, onde a perda de sincronização pode comprometer toda a comunicação, o modo assíncrono permite que cada mensagem seja tratada de forma independente. Isso melhora a tolerância a falhas e erros de transmissão. Embora o vetor inicial não precise ser mantido em segredo, ele deve ser escolhido de maneira que não seja previsível.

Exemplo 9. *O Wired Equivalent Privacy (WEP) era o padrão para segurança em conexões WiFi desde 1997 e é essencialmente uma cifra de fluxo que opera de modo assíncrono. Para gerar um fluxo de bits pseudo-aleatórios, o WEP utiliza o RC4 [RS16] — um gerador de números pseudo-aleatórios (GNP) proposto por Ron Rivest em 1987. O RC4 recebe como entrada uma semente de 40 ou 104 bits e um vetor inicial (VI) não sigiloso de 24 bits.*

Para cada pacote transmitido, o WEP gera um novo VI. Este VI, combinado com a chave secreta, é utilizado para inicializar o GNP RC4.

O RC4 gera um fluxo de chaves a partir da combinação da chave secreta e do VI. Este fluxo de chaves é então combinado com o conteúdo do pacote usando a operação XOR, produzindo o texto cifrado.

Embora o VI de 24 bits não precise ser mantido em segredo, seu tamanho é consideravelmente pequeno. Devido ao espaço limitado de possíveis VI, há uma alta probabilidade de repetição. Especificamente, com 50% de chance, o VI se repete a cada 5000 pacotes transmitidos. Essa repetição torna o

WEP vulnerável a ataques de criptografia, como demonstrado por Fluhrer, Mantin e Shamir em 2001 [FMS01]. Eles mostraram que a repetição do VI permite que um adversário possa recuperar a chave secreta após coletar uma quantidade suficiente de pacotes.

Hoje em dia, scripts como o `aircrack-ng`[2] *são capazes de explorar essas vulnerabilidades. Utilizando um computador pessoal, é possível quebrar uma senha WEP em questão de minutos. Esta vulnerabilidade crítica levou à obsolescência do WEP.*

Devido às suas fraquezas, o WEP foi substituído pelo Wi-Fi Protected Access (WPA) e, posteriormente, pelo WPA2 entre 2004 e 2006. Esses novos padrões introduziram melhorias significativas na segurança, incluindo a utilização de protocolos de criptografia mais robustos e a implementação de vetores iniciais maiores e mais seguros, reduzindo drasticamente a probabilidade de repetição e aumentando a resistência contra ataques.

4.3 Construções Práticas

A existência de geradores de números pseudo-aleatórios (GNPs) não foi demonstrada matematicamente. No Apêndice 7, discutimos por que é tão desafiador encontrar um GNP que seja comprovadamente seguro. O ponto crucial, por enquanto, é que é extremamente difícil demonstrar matematicamente a segurança de um sistema de criptografia baseado em GNPs.

Devido a essa dificuldade, a abordagem predominante na validação de GNPs é empírica. Em vez de provar rigorosamente que um GNP é seguro, os candidatos a GNP são validados através de extensivos testes e tentativas de quebra. A segurança de um GNP é então inferida a partir da incapacidade de construir um distinguidor eficiente que possa diferenciá-lo de uma sequência verdadeiramente aleatória.

O processo de validação segue os seguintes passos:

Proposição de Candidatos: Novos algoritmos GNP são propostos por criptógrafos e pesquisadores. Estes algoritmos devem passar por um rigoroso processo de avaliação.

Testes de Distinção: Diversos testes estatísticos e distinções são aplicados aos candidatos. Esses testes tentam identificar padrões ou irre-

[2] http://www.aircrack-ng.org/

gularidades que possam diferenciar a saída do GNP de uma sequência aleatória.

Tentativas de Quebra: Comunidades de criptógrafos e pesquisadores tentam construir distinguidores que possam quebrar a segurança dos GNPs propostos. Cada tentativa frustrada de construir um distinguidor fortalece a confiança na segurança do GNP.

Validação Empírica: A segurança de um GNP é considerada válida empiricamente quando ele resiste a extensivos testes e tentativas de quebra ao longo do tempo. Isso não prova a segurança de forma absoluta, mas fornece uma base prática sólida para sua utilização.

A abordagem empírica para a validação de GNPs é fundamental para o avanço da criptografia moderna. Embora não possamos provar matematicamente que um GNP é seguro, a validação empírica oferece um método robusto para garantir a eficácia e a confiabilidade dos sistemas de criptografia na prática. Esse processo contínuo de avaliação e teste é essencial para identificar e utilizar GNPs que oferecem altos níveis de segurança.

4.3.1 RDRL

Os registradores de deslocamento com realimentação linear, conhecidos como *Registradores de Deslocamento com Retroalimentação Linear* (RDRLs), são uma classe de algoritmos frequentemente utilizados em criptografia para gerar sequências pseudo-aleatórias de bits.

Um RDRL é composto por duas partes principais:

Vetor de Registradores: Um vetor de registradores $s_0, s_1, \ldots, s_{n-1}$ que armazena exatamente um bit cada.

Coeficientes de Feedback: Uma sequência de bits $c_0, c_1, \ldots, c_{n-1}$, conhecida como coeficientes de feedback, que determina como o novo bit é calculado a cada passo.

O funcionamento de um RDRL pode ser descrito em etapas simples:

1. O RDRL começa com um *estado inicial* definido pelo vetor de registradores.

2. A cada passo, o estado do RDRL é atualizado da seguinte maneira:

- Todos os bits no vetor de registradores são deslocados uma posição para a direita.
- O bit mais à esquerda (s_0) é atualizado com um novo valor calculado usando uma operação XOR sobre os bits do vetor de registradores ponderados pelos coeficientes de feedback.

A fórmula para calcular o novo bit s_0 é:

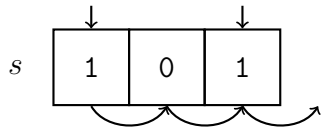
$$s_0 := \bigoplus_{i=0}^{n-1} c_i s_i$$

3. A cada atualização, o bit mais à direita (s_{n-1}) do vetor de registradores é produzido como parte da sequência pseudo-aleatória.

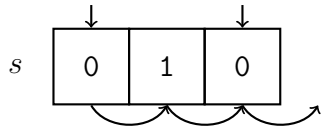
Exemplo 10. *Para ilustrar, considere um RDRL com três registradores e coeficientes de feedback $c_0 = 1$, $c_1 = 0$, $c_2 = 1$. Suponha que o valor inicial (semente) dos registradores seja $s_0 = 1$, $s_1 = 0$, $s_2 = 1$. A cada passo, o estado é atualizado da seguinte maneira:*

- *O novo valor de s_0 é calculado como $s_0 := c_0 s_0 \oplus c_1 s_1 \oplus c_2 s_2$. De maneira equivalente, o novo valor de s_0 é calculado como o XOR entre os bits indicados pelo coeficiente de feedback.*
- *O vetor de registradores é então deslocado para a direita, e o valor de s_{n-1} é emitido como parte da sequência de saída.*

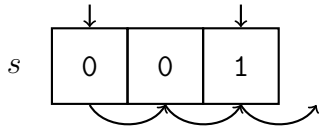
Vamos mostrar os primeiros passos desse RDRL:



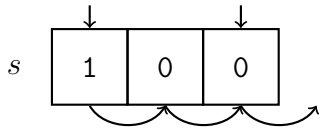
O novo bit a ser inserido na primeira posição é: $1 \oplus 1 = 0$
O primeiro bit produzido é `1` *que é o bit que estava na última posição.*



O novo bit a ser inserido na primeira posição é: $0 \oplus 0 = 0$
O segundo bit produzido é `0` *que é o bit que estava na última posição.*



O novo bit a ser inserido na primeira posição é: $0 \oplus 1 = 1$
O terceiro bit produzido é `1` *que é o bit que estava na última posição.*



O novo bit a ser inserido na primeira posição é: $1 \oplus 0 = 1$
O quarto bit produzido é `0` *que é o bit que estava na última posição.*
Portanto, os primeiros bits da sequência gerada seriam: `1, 0, 1, 0`.

Um RDRL com n registradores é capaz de gerar no máximo $2^n - 1$ bits antes de começar a repetir sua sequência. Isso ocorre porque o número de estados possíveis para um vetor de registradores com n bits é 2^n, e o estado em que todos os bits são 0 não é válido– se ele ocorresse, o RDRL continuaria gerando apenas 0s indefinidamente. Assim, depois de no máximo $2^n - 1$ passos, o RDRL retornará a um estado anterior, iniciando assim um ciclo.

O tamanho do ciclo de um RDRL refere-se ao número de bits gerados antes que a sequência comece a se repetir. Em outras palavras, é o período da sequência pseudo-aleatória produzida pelo RDRL. Um bom RDRL é aquele que tem um ciclo longo, idealmente próximo de $2^n - 1$, o que significa que ele pode gerar uma longa sequência de bits únicos antes de repetir.

Como discutido anteriormente, o comportamento de um RDRL é definido por um vetor de bits que determina o coeficiente de realimentação. Dado um RDRL com n registradores, certos vetores de feedback produzem ciclos ideais, enquanto outros não. Os vetores que geram um ciclo ideal representam polinômios irredutíveis de grau n. Um polinômio irredutível é aquele que não pode ser fatorado em polinômios de grau menor – eles funcionam como números primos no universo dos polinômios.

Por exemplo, o polinômio $x^3 + x^2 + x + 1$ (representado pelo vetor binário 1111_2) pode ser fatorado como $(x^2+1)(x+1)$ no caso de coeficientes binários,

que é o nosso contexto. Por outro lado, o polinômio $x^3 + x + 1$ (representado pelo vetor binário 1011_2) é irredutível. Portanto, para um registrador de 4 bits, o vetor binário 1011_2 gera um ciclo ideal de tamanho $2^4 - 1 = 15$, enquanto o vetor 1111_2 não produz um ciclo ideal. Contudo, não demonstraremos formalmente este resultado.

A linearidade dos RDRLs é um problema, pois torna previsível antecipar os próximos bits gerados com base nos anteriores. Essa previsibilidade compromete a segurança de sistemas de cifra de fluxo.

Na próxima seção, veremos como combinar múltiplos RDRLs de forma a quebrar essa linearidade, produzindo assim um GNP mais robusto e seguro, adequado para aplicações criptográficas.

O RDRL não é considerado seguro para aplicações criptográficas porque a operação que gera novos bits é linear. Essa linearidade permite que o RDRL seja previsível e, portanto, vulnerável a ataques. Um adversário que observa um número suficiente de bits gerados pelo RDRL pode resolver um sistema de equações lineares para determinar a semente inicial e, consequentemente, prever todos os bits subsequentes.

Embora os RDRLs não sejam seguros, muitos Geradores de Números Pseudo-Aleatórios (GNPs) usados na prática são versões modificadas deste esquema geral. A segurança dos GNPs modernos geralmente depende da introdução de não-linearidades no processo de geração de bits. Em vez de usar exclusivamente operações lineares como a XOR, os GNPs seguros utilizam funções não lineares. Essas funções podem incluir operações aritméticas complexas, substituições de bits baseadas em tabelas (substituições S-box) ou outras transformações matemáticas que dificultam a previsão dos bits gerados.

4.3.2 Trivium

Em 2008, um concurso científico focado na produção de cifras de fluxo seguras foi realizado com o objetivo de identificar algoritmos que fossem tanto seguros quanto eficientes. Este concurso, conhecido como projeto eSTREAM, reuniu diversas propostas de algoritmos de criptografia de fluxo, cada um competindo para se destacar como uma solução viável para a criptografia moderna.

Entre os algoritmos selecionados pelo projeto eSTREAM, o *Trivium* [DC06] se destacou como uma alternativa segura e eficiente. O Trivium foi desenvolvido por Christophe De Canniere e Bart Preneel em 2006. Este algoritmo

é projetado para ser simples e leve, mas ao mesmo tempo, capaz de oferecer alto grau de segurança.

O Trivium recebe dois valores como entrada, ambos com 80 bits:

1. *Semente (Key)*: Uma sequência de 80 bits que serve como chave secreta do algoritmo.

2. *Vetor Inicial (VI)*: Outra sequência de 80 bits, que é usada para garantir a unicidade da cifra gerada para diferentes mensagens, mesmo quando a mesma chave é utilizada.

O estado interno do Trivium é representado por um vetor de 288 bits. Esse estado é atualizado iterativamente para gerar a sequência de saída pseudo-aleatória. A atualização do estado é projetada de tal forma que o ciclo do Trivium tem um tamanho de 2^{64} bits, ou seja, o Trivium pode gerar uma sequência de até 2^{64} bits antes de começar a repetir.

Até o momento, o Trivium tem resistido a uma ampla gama de ataques criptoanalíticos, o que reforça sua reputação como um algoritmo seguro. Além disso, ele é eficiente em termos de uso de recursos computacionais, tornando-o adequado para aplicações com restrições de hardware, como dispositivos embarcados.

Estrutura e Funcionamento do Trivium

O Trivium possui três registradores denominados A, B e C, com tamanhos de 93, 84 e 111 bits, respectivamente, totalizando 288 bits.

A inicializaćão dos registradores é feita da seguinte forma:

1. *Chave (Key)*:

    - A chave de 80 bits é inserida nas primeiras posições do registrador A.
    - As demais posições do registrador A são preenchidas com zeros.

2. *Vetor Inicial (VI)*:

    - O vetor inicial de 80 bits é inserido nas últimas posições do registrador B.
    - As demais posições do registrador B são preenchidas com zeros.

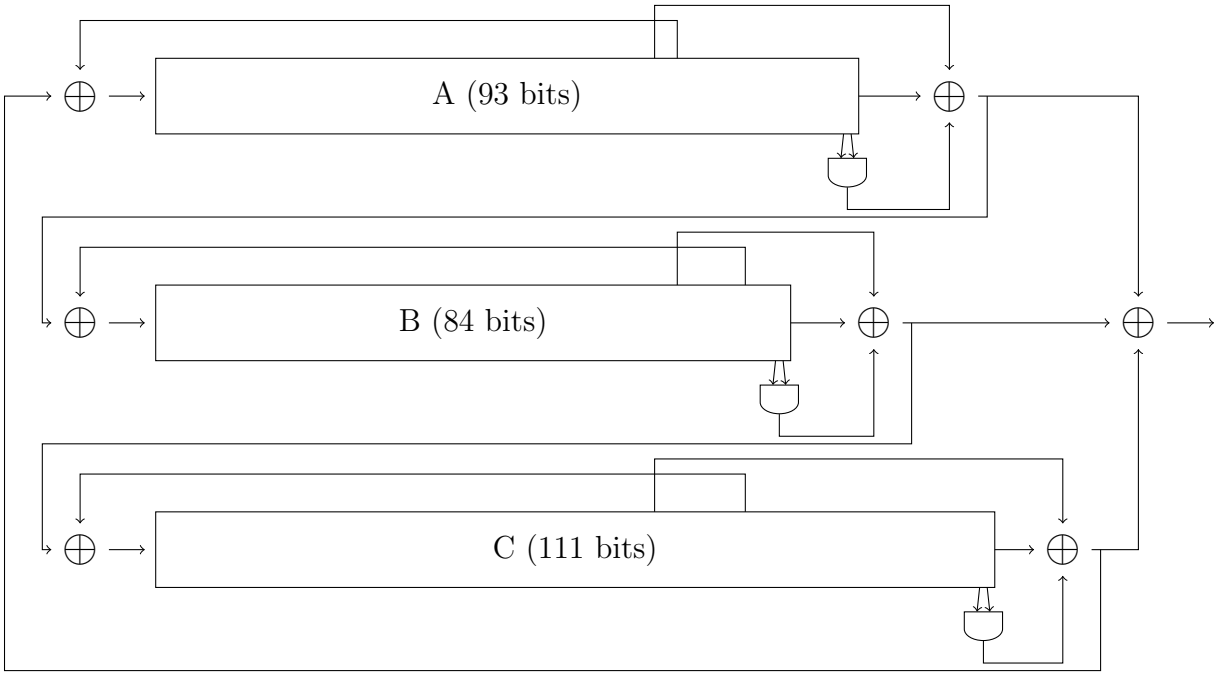
3. Todas as posições do registrador C são preenchidas com zeros, exceto as últimas três posições, que são preenchidas com uns.

Antes de começar a produzir o fluxo de bits, o Trivium passa por uma *fase de aquecimento*. Durante esta fase, o algoritmo roda $4 \times 288 = 1152$ ciclos para misturar completamente o estado interno. Esta fase de aquecimento ajuda a assegurar que o estado inicial não revele diretamente a chave ou o vetor inicial.

Os passos seguintes do Trivium são:

1. *Cálculo do Bit de Saída*:
   - Cada registrador calcula o AND entre o penúltimo e o antepenúltimo bit.
   - O resultado é então combinado usando XOR com o último bit e com um bit de uma posição fixa chamada de *feedforward bit*.
   - Este cálculo gera o *bit de saída*.
2. *Envio do Bit de Saída*:
   - O bit de saída de cada registrador é enviado para o registrador adjacente:
     - Registrador A envia seu bit de saída para o registrador B.
     - Registrador B envia seu bit de saída para o registrador C.
     - Registrador C envia seu bit de saída para o registrador A.
3. *Deslocamento dos Bits*:
   - Cada registrador desloca seus bits uma posição para a direita, semelhante ao RDRL.
4. *Inserção do Novo Bit*:
   - Cada registrador aplica XOR entre o bit recebido e um bit de uma posição fixa chamada de *feedbackward bit*.
   - O resultado é inserido na primeira posição do registrador.
5. *Geração do Bit de Fluxo*:

- O XOR dos três bits de saída dos registradores A, B e C é calculado.
- Este resultado final é o bit do fluxo gerado pelo Trivium.



Resumindo, o Trivium é um algoritmo de cifra de fluxo que utiliza três registradores interligados para gerar uma sequência pseudoaleatória de bits. Em cada passo, os registradores realizam operações lógicas e de deslocamento, trocando bits de saída entre si e inserindo novos bits com base em cálculos de feedback. A combinação dos bits de saída dos três registradores gera o fluxo de bits que é usado para criptografar ou descriptografar mensagens.

A Tabela 4.3.2 resume as especificações do Trivium, a posição do feedack-bit e do feed-foward bit de cada um dos três registradores.

Registrador	**Feedback Bit**	**Feedforward Bit**	**Tamanho**
A	69	66	93
B	78	69	84
C	87	66	111

Tabela 4.1: Especificações do Trivium

A escolha de um bom gerador de números pseudo-aleatórios (GNP) é

crucial para garantir a segurança de uma cifra de fluxo. Utilizar geradores inadequados pode comprometer seriamente a integridade do sistema criptográfico. Um erro comum é utilizar funções padrão de bibliotecas de programação, como a função `rand` da biblioteca padrão da linguagem `C`. Embora essas funções sejam adequadas para simulações e outras aplicações não críticas, elas não oferecem o nível de segurança necessário para aplicações criptográficas.

Para garantir a segurança de sistemas criptográficos, é essencial utilizar GNPs que foram rigorosamente avaliados e validados pela comunidade criptográfica. A orientação geral é utilizar geradores selecionados pelo projeto eSTREAM, que foram submetidos a testes extensivos e demonstraram ser seguros contra uma variedade de ataques criptoanalíticos. Dois exemplos notáveis de geradores aprovados pelo projeto eSTREAM são o Trivium, que foi detalhado nesta seção, e o SALSA20.

4.4 Exercicios

Exercício 19. *O que precisamos assumir para que um sistema de criptografia baseado em cifra de fluxo seja seguro?*

Em que sentido podemos considerá-lo seguro?

Exercício 20. *Mostre que o gerador G com fator de expansão $l(n) = n + 1$ que recebe $s \in \{0,1\}^n$ e devolve s concatenado com $\bigoplus_{i=0}^{n} s_i$ não é um GNP.*

Exercício 21. *Construa um dintinguidor eficiente D para o RDRL simples.*

Exercício 22. *Por que em uma cifra de fluxo não podemos criptografar duas mensagens distintas com a mesma chave?*

Exercício 23. *Sejam $y_0, y_1, y_2 \ldots$ os bits gerados pelo algoritmo RC4. É possível mostrar que para uma distribuição uniforme de sementes e vetores iniciais, a probabilidade dos bits $y_9, \ldots, y_{16}$ serem todos iguais a 0 é $\frac{2}{256}$. Mostre como construir um algoritmo eficiente D capaz de distinguir as sequências de bits produzidas pelo RC4 de uma sequência realmente aleatória.*

Exercício 24. *Considere a seguinte implementação de uma cifra de fluxo:*

1. *Utilizamos o número de segundos desde primeiro de janeiro de 1970 até o momento atual para gerar uma semente s que armazenamos em um local seguro.*

2. *Utilizamos, então, a implementação* `rand` *da biblioteca padrão do* `C` *para gerar uma sequência de* n *bits* $G(s)$.

3. *Produzimos a cifra* $c = G(s) \oplus m$ *supondo que* $|m| = n$.

4. *Para descriptografar recuperamos* s, *aplicamos* $G(s) \oplus c$.

Descreva duas vulnerabilidades deste protocolo.

Capítulo 5

Cifras de Bloco

No capítulo anterior, vimos que é possível construir um sistema criptográfico seguro contra ataques apenas contra o texto cifrado ("ciphertext only"). Para que isso seja viável, é necessário supor a existência de um gerador de números pseudo-aleatórios, ou seja, um algoritmo capaz de gerar uma sequência de bits que, para todos os efeitos práticos, não pode ser distinguida de uma sequência realmente aleatória.

Embora essa definição de segurança seja um pouco mais fraca do que a apresentada no Capítulo **??**, ela tem a vantagem de não exigir chaves extremamente longas. No entanto, assim como o "One-Time Pad" (OTP), as cifras de fluxo também apresentam dois problemas importantes: se uma chave for reutilizada, a segurança é completamente comprometida, e se o adversário souber partes da mensagem original, não há garantias de segurança.

O modelo das cifras de fluxo é ideal para descrever as Máquinas Enigma, utilizadas pelo exército nazista durante a década de 1940. A cifra empregada por essas máquinas foi quebrada não apenas pelo avanço tecnológico, que possibilitou a construção da máquina Bombe e do computador Colossus, mas também porque os aliados conseguiram obter trechos de mensagens já descriptografadas.

Com base na teoria da criptografia moderna, podemos afirmar que o tipo de ataque realizado pelos ingleses é classificado como um ataque com texto original conhecido ("known plaintext"), onde o atacante tem acesso tanto ao texto cifrado quanto a uma versão conhecida do texto original.

Neste capítulo, vamos explorar um modelo de ataque ainda mais poderoso: os ataques com escolha de texto plano (ETP). Nesse modelo, imaginamos que o adversário tem acesso a um "oráculo". Esse oráculo é como uma

caixa mágica que permite ao adversário enviar mensagens de sua escolha e receber de volta a versão criptografada dessas mensagens. O adversário pode usar essas informações para tentar descobrir a chave de criptografia ou outros segredos do sistema [BDJR97].

É importante notar que os ataques com texto original conhecido são um caso particular desse modelo mais geral, mas há situações em que devemos considerar que o adversário possui essa capacidade adicional, o que torna o cenário de segurança ainda mais desafiador.

Vamos explicar a segurança contra ataques com escolha de texto plano (ETP) de maneira semelhante à segurança contra ataques em que o adversário só tem acesso ao texto cifrado (*ciphertext only*). A garantia de segurança que buscamos continua sendo a mesma que discutimos anteriormente, mas agora vamos considerar um cenário de ameaças mais poderoso.

O modelo pode ser descrito da seguinte forma:

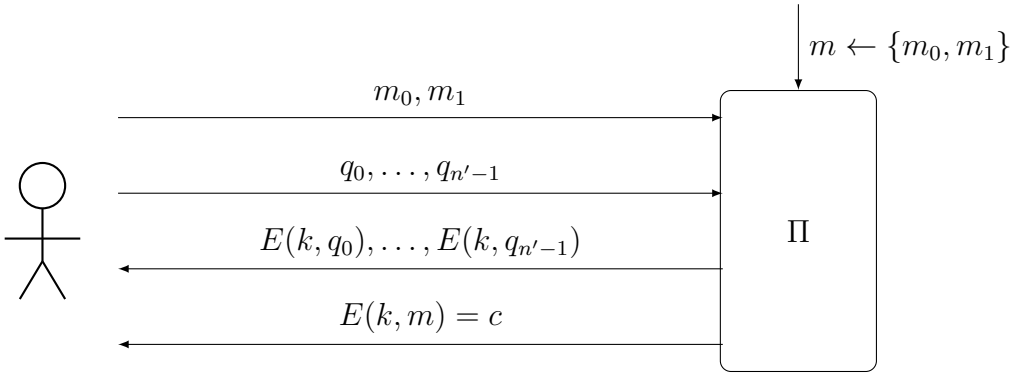
1. O adversário escolhe duas mensagens de mesmo tamanho, m_0 e m_1, e as envia para o sistema de criptografia.

2. O sistema gera uma chave secreta e escolhe aleatoriamente uma dessas mensagens para criptografar.

3. Durante o processo, e mesmo depois de receber o texto cifrado, o adversário pode perguntar ao sistema como outras mensagens escolhidas por ele seriam cifradas.

4. O sistema então retorna o texto cifrado para o adversário.

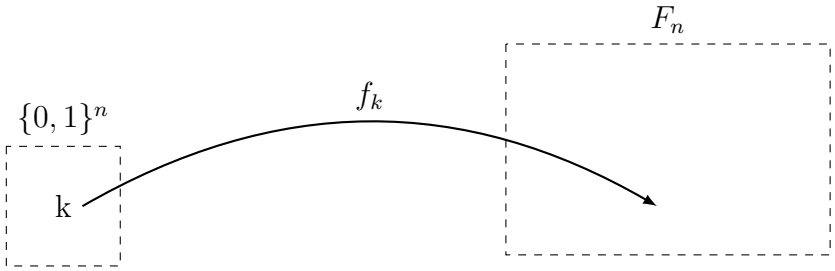
5. Com base nas informações disponíveis, o adversário tenta adivinhar qual das duas mensagens foi cifrada.



Um sistema é *seguro contra ETP* se nenhum adversário polinomial tiver uma vantagem significativa para vencer o jogo que acabamos de descrever. Ou seja, a probabilidade de que um adversário consiga vencer o jogo é apenas desprezivelmente maior do que 1/2

Para construir um sistema seguro contra ataques ETP, precisamos usar o conceito de funções pseudoaleatórias (FPA). Em termos simples, uma função pseudoaleatória é um processo que transforma uma entrada (um conjunto de bits) em uma saída que parece aleatória.

A diferença principal é que a função pseudoaleatória não apenas estende ou gera uma sequência longa a partir de um pequeno pedaço de informação (como um gerador de números pseudoaleatórios faria), mas sim utiliza uma chave para definir a própria função que realiza essa transformação. Em outras palavras, a chave é usada para escolher a função específica dentro de um grande conjunto de funções possíveis. Quando aplicamos essa função a uma entrada, ela "embaralha" os bits de maneira que a saída pareça aleatória, mesmo sendo gerada de forma determinística.



Na cifra de bloco, uma chave é escolhida aleatoriamente em $\{0,1\}^n$, e a partir dessa chave é gerada deterministicamente uma função dentro do conjunto de todas as funções F_n, que, dado um bloco de n bits, produz um novo bloco de n bits. Uma cifra de blocos é considerada segura se um adversário não for capaz de distinguir esse processo de uma escolha aleatória de uma função diretamente em F_n.

Em outras palavras, para qualquer adversário que tente diferenciar entre essas duas situações, a função pseudoaleatória será tão boa quanto uma função completamente aleatória.

Uma *cifra de bloco* é um tipo específico de função pseudoaleatória, que chamamos de *permutação pseudoaleatória* (PPA). Sua característica distintiva é a capacidade de reverter o processo de "embaralhamento" dos bits. Isso significa que, além de transformar uma mensagem de tamanho fixo (o

bloco) em uma saída que parece aleatória, ela também permite que essa transformação seja desfeita de forma eficiente, recuperando a mensagem original.

Assim como outras funções pseudoaleatórias, a cifra de bloco utiliza uma chave para definir como os bits serão embaralhados. A segurança dessa cifra vem do fato de que, para um adversário, ela é indistinguível de uma permutação completamente aleatória. Ou seja, não existe um método eficiente para diferenciar entre o comportamento da cifra de bloco e o de uma função que simplesmente embaralha os bits de maneira aleatória.

Na próxima seção, veremos como essas cifras de bloco podem ser combinadas para criptografar mensagens de qualquer tamanho, mantendo a segurança do sistema criptográfico.

5.1 Modos de Operação

Uma cifra de bloco é usada para criptografar informações em pedaços de tamanho fixo, chamados de *blocos*. No entanto, na prática, muitas vezes precisamos criptografar mensagens que são maiores ou menores que esse tamanho fixo. Para lidar com isso, precisamos de uma forma de combinar os blocos criptografados pela cifra de bloco para proteger a mensagem inteira.

Uma maneira simples e intuitiva de fazer isso é chamada de *Livro de Código Eletrônico* (LCE). Nesse modo de operação, dividimos a mensagem em vários blocos do mesmo tamanho e, em seguida, criptografamos cada um desses blocos separadamente com a cifra de bloco. Depois de criptografar todos os blocos, simplesmente juntamos os blocos cifrados para formar a mensagem final cifrada.

No entanto, esse método tem uma grande falha de segurança: blocos idênticos na mensagem original serão criptografados da mesma maneira, resultando em blocos idênticos no texto cifrado. Isso significa que, se um adversário observar blocos repetidos no texto cifrado, ele pode deduzir que partes da mensagem original são iguais, o que compromete a segurança.

Por exemplo, imagine que um adversário queira descobrir se duas partes de uma mensagem criptografada são iguais ou diferentes. Ele pode criar duas mensagens de teste: uma onde a mesma parte se repete, e outra onde as duas partes são diferentes.

Se ele criptografar essas mensagens e o texto cifrado resultar em dois blocos idênticos, o adversário saberá que a mensagem original tinha partes iguais. Se os blocos cifrados forem diferentes, ele saberá que as partes da

mensagem original também eram diferentes.

Em outras palavras, esse adversário consegue distinguir entre as duas cifras ao observar como cada bloco é criptografado. Isso significa que o modo LCE não é seguro nem mesmo contra *ataques apenas contra a cifra*. A Figura 5.1 ilustra essa vulnerabilidade.[1].

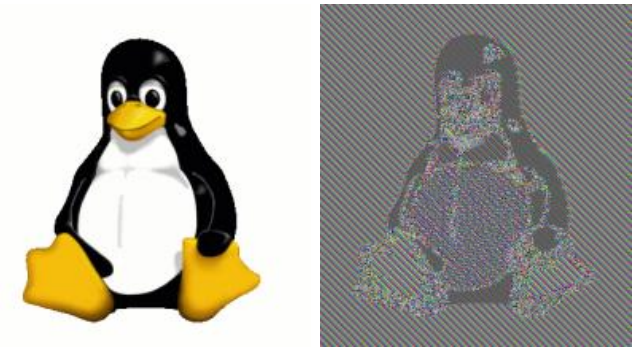


Figura 5.1: Imagem criptografada no modo LCE

Essa forma ingênua de combinar os blocos não garante a segurança contra os tipos de ataques mais básicos que discutimos anteriormente. O que realmente precisamos é de um sistema que seja seguro contra ataques ETP, onde o adversário pode escolher mensagens e ver como elas são criptografadas.

Na definição de segurança contra ETP, o adversário tem a capacidade de "consultar" o sistema para descobrir como uma mensagem específica seria criptografada. Isso significa que, para proteger o sistema, o processo de criptografia não pode ser determinístico, ou seja, não pode produzir o mesmo resultado toda vez que a mesma mensagem é criptografada. Se fosse determinístico, o adversário poderia facilmente enganar o sistema ao enviar mensagens que ele já consultou antes.

Para garantir isso, incluímos um bloco aleatório no início de cada mensagem, conhecido como *vetor inicial* (VI). Esse vetor inicial não precisa ser mantido em segredo, mas ele assegura que cada vez que uma mensagem é criptografada, o resultado será diferente, mesmo que a mensagem original seja a mesma. Assim como nas cifras de fluxo, o vetor inicial é fundamental para evitar que padrões previsíveis apareçam no texto cifrado, aumentando a segurança do sistema.

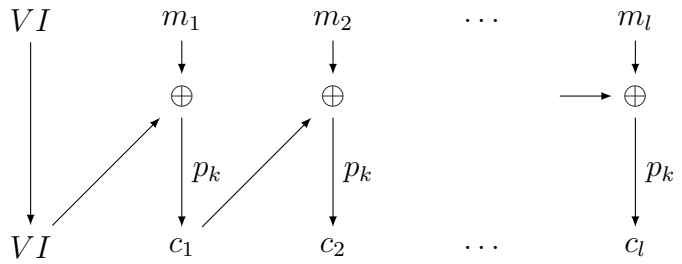
[1]Imagem tirada do verbete *Modo de Operação (criptografia)* da Wikipédia (`https://pt.wikipedia.org`)

No modo de operação *Encadeamento de Blocos Cifrados* (EBC), cada bloco de texto cifrado não depende apenas da cifra de bloco aplicada ao bloco de texto original, mas também do bloco cifrado que veio imediatamente antes dele. Isso cria uma "cadeia" onde cada bloco cifrado influencia o próximo, aumentando a segurança do processo.

O funcionamento é o seguinte:

- Primeiro, temos um bloco aleatório inicial, o vetor inicial (VI), que inicia o processo ($c_0 = IV$). Esse bloco serve para garantir que cada mensagem cifrada será única.
- Em seguida, cada bloco da mensagem original é combinado com o bloco cifrado anterior antes de ser criptografado usando uma PPA ($p_k(c_{i-1} \oplus m_i)$). Isso significa que, para criptografar um bloco da mensagem, usamos a cifra de bloco, mas também consideramos o resultado do bloco anterior.

Essa cadeia de dependência faz com que, mesmo que dois blocos da mensagem original sejam iguais, seus blocos cifrados finais sejam diferentes, porque cada um deles leva em conta o bloco anterior. Ao final, o resultado é uma sequência de blocos cifrados, onde cada um depende do anterior, tornando o sistema muito mais seguro.



Para descriptografar uma mensagem que foi cifrada usando o modo EBC, o processo é basicamente o inverso da criptografia.

Cada bloco da mensagem original é recuperado usando o bloco cifrado correspondente e o bloco cifrado anterior. Em outras palavras, para descriptografar um bloco específico, aplicamos o processo inverso da cifra de bloco ao bloco cifrado seguinte e, em seguida, combinamos esse resultado com o bloco cifrado anterior ($m_i := p_k^{-1}(c_{i+1}) \oplus c_i$).

Esse processo continua ao longo de toda a sequência de blocos cifrados, revertendo o "encadeamento" até que todos os blocos da mensagem original tenham sido restaurados.

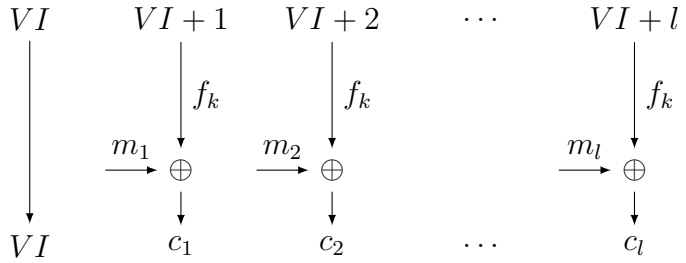
É possível provar que este sistema é seguro contra ETP.

Teorema 4. *Um sistema que utiliza uma permutação pseudoaleatória segura no modo EBC é seguro contra ataques do tipo ETP.*

A principal limitação do modo EBC é que os blocos de dados precisam ser processados em sequência, ou seja, cada bloco só pode ser criptografado depois que o bloco anterior foi processado. Isso significa que o algoritmo não pode ser facilmente paralelizado, o que pode tornar o processo mais lento.

O *modo contador* (Ctr) não tem essa limitação. Esse modo funciona de maneira semelhante a uma cifra de fluxo, permitindo que os blocos sejam processados simultaneamente, o que aumenta a eficiência.

No modo Ctr, o processo começa com uma sequência de bits aleatória, o vetor inicial (VI), que é usado como ponto de partida. Em vez de depender do bloco anterior, cada bloco da mensagem é combinado com um valor único, gerado ao somar o índice do bloco ao vetor inicial ($VI+i$). Esse valor é então processado por uma função pseudoaleatória, e o resultado é combinado com o bloco da mensagem para produzir o texto cifrado ($c_i = m_i \oplus f_k(VI + i)$).



Para descriptografar uma mensagem no modo contador (Ctr), o processo é simples e eficiente. Primeiro, somamos o vetor inicial (VI) ao número do bloco que estamos tentando recuperar. Em seguida, aplicamos a mesma função usada durante a criptografia para gerar o bloco de bits correspondente e, por fim, realizamos a operação XOR entre essa sequência e o bloco cifrado correspondente ($c_i \oplus f_k(VI + i)$).

Ao contrário do modo EBC, não precisamos reverter a função utilizada na criptografia. Em outras palavras, não é necessário usar uma permutação pseudoaleatória que permita desfazer o processo de "embaralhamento". Uma

função simples que embaralha os bits de maneira segura é suficiente para tanto a criptografia quanto a descriptografia.

O modo Ctr também é seguro contra ataques do tipo ETP.

Teorema 5. *Um sistema que utiliza uma função pseudoaleatória segura no modo Ctr é seguro contra ataques do tipo ETP.*

Para se convencer de que o teorema é válido, vamos considerar dois sistemas de criptografia: o sistema Π, que usa uma função pseudoaleatória no modo contador (Ctr), e o sistema Π', que é semelhante, mas usa uma função realmente aleatória.

Suponha que exista um adversário $\mathcal{A}$ que tenta atacar o sistema. Para analisar a segurança, podemos imaginar que estamos tentando construir um distinguidor que tenta identificar se o sistema está usando uma função pseudoaleatória ou uma função realmente aleatória.

O processo seria o seguinte:

1. Sempre que o adversário quiser saber como uma mensagem seria criptografada, geramos um vetor inicial (VI) e, em seguida, aplicamos a função (seja ela pseudoaleatória ou realmente aleatória) ao vetor inicial somado ao número do bloco correspondente. O resultado é então combinado com a mensagem original para gerar o texto cifrado que é devolvido ao adversário.

2. Quando o adversário tenta adivinhar entre duas mensagens enviadas ao sistema, o processo é o mesmo: geramos um vetor inicial, aplicamos a função escolhida e combinamos o resultado com as mensagens que o adversário deseja testar.

Se o sistema Π' usa uma função realmente aleatória, ele se comporta como uma "caixa preta" que não permite que o adversário aprenda nada útil sobre a mensagem original, além do que poderia descobrir por simples adivinhação. Neste caso, a chance de sucesso do adversário é equivalente a um palpite aleatório (50%).

Agora, precisamos verificar se o sistema Π, que usa uma função pseudoaleatória, se comporta de maneira suficientemente semelhante ao sistema Π' para que o adversário não possa distinguir entre eles. Se a função pseudoaleatória usada for segura, ela será praticamente indistinguível de uma função realmente aleatória, o que significa que a chance do adversário distinguir entre as duas será desprezivelmente superior a $1/2$.

A única maneira de o adversário ganhar uma vantagem é se o vetor inicial usado para criptografar coincider com o da mensagem que ele está testando. No entanto, como o número total de possibilidades para o vetor inicial é enorme (2^n para um bloco de tamanho n), a probabilidade de tal coincidência ocorrer é desprezível.

Portanto, a chance do adversário ter sucesso em um ataque ETP contra o sistema Π é apenas desprezivelmente superior a $1/2$. Isso nos permite concluir que o sistema Π é seguro contra ataques ETP.

A explicação indica que o tamanho dos blocos também influencia em sua segurança. Isso ocorre, pois quanto menores os blocos maior a chance de gerar vetores inciais idênticos, e portanto, de criptografar duas mensagens distintas com a mesma chave.

Optamos por apresentar os modos EBC e Ctr devido à sua ampla utilização e reconhecimento. O modo EBC é empregado no protocolo de segurança SSH (Secure Shell), que permite o acesso remoto seguro a sistemas, enquanto o Ctr é utilizado no protocolo de criptografia do WhatsApp e no TLS (Transport Layer Security), que protege comunicações na web, como em conexões via HTTPS. Além desses, existem outros modos, como o CFB (Cipher Feedback Mode), usado no protocolo PGP (Pretty Good Privacy), que pode ser usado para criptografia ponta a ponta garantindo a privacidade de e-mails.

5.2 Construções Práticas

Como vimos anteriormente, uma Permutação Pseudoaleatória (PPA) é um método que embaralha blocos de dados de forma que o resultado final seja imprevisível. Essa permutação é gerada de maneira determinística, ou seja, segue um processo definido a partir de uma chave específica.

Imaginemos que temos uma chave aleatória e uma função que, a partir dessa chave, cria uma permutação dos dados. O objetivo é que essa permutação seja indistinguível de uma permutação realmente aleatória, mesmo para quem está tentando analisar o sistema de fora. Embora não conheçamos um sistema demonstradamente seguro, na prática, sistemas como o AES (Advanced Encryption Standard) têm se mostrado eficazes e seguros.

O desafio aqui é desenvolver um algoritmo onde, ao mudar um único bit, seja na chave ou na mensagem, isso tenha um impacto significativo em toda a cifra resultante. Isso significa que uma pequena alteração deve se espalhar

por toda a saída, dificultando a previsibilidade do resultado.

Para alcançar esse efeito, utilizamos o conceito de confusão e difusão, proposto por Claude Shannon [Sha49]. A *confusão* é a técnica de tornar a relação entre a chave e o texto cifrado tão complexa quanto possível. Um exemplo simples é dividir o bloco de dados em pedaços menores, como grupos de 8 bits, e substituí-los por novos valores, seguindo uma *tabela de substituição*. Entretanto, apenas confundir os dados não é suficiente. Se alterarmos um bit no início do processo, isso pode afetar apenas uma pequena parte do resultado final.

É aí que entra a *difusão*, que tem como objetivo espalhar a influência dessa pequena mudança por todo o bloco de dados. Em cifras de bloco modernas, essas fases de confusão e difusão são repetidas várias vezes para garantir que qualquer pequena alteração na entrada cause uma mudança ampla e complexa na saída.

5.2.1 Data Encryption Standard (DES)

O *Data Encryption Standard* (DES) foi o padrão de criptografia para cifras de bloco que predominou do final dos anos 1970 até o final dos anos 1990. Esse algoritmo foi originalmente desenvolvido pela IBM, mas passou por modificações significativas feitas pela NSA (Agência de Segurança Nacional dos Estados Unidos) antes de se tornar um padrão internacional.

O DES utiliza uma estrutura conhecida como *rede de Feistel*, que permite a criação de uma função criptográfica que pode ser invertida de forma eficiente. Esse tipo de rede funciona aplicando uma série de funções sobre os dados de entrada, o que facilita a criptografia e a descriptografia.

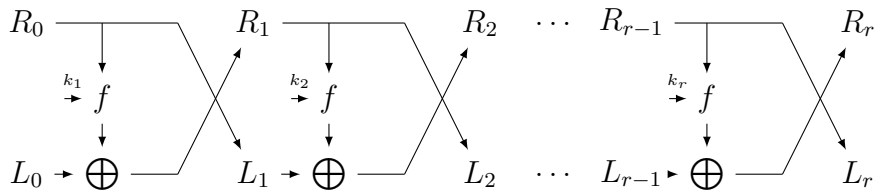
Quando o DES processa uma mensagem, essa mensagem é dividida em duas partes iguais. Vamos chamar essas partes de L (a primeira metade) e R (a segunda metade). Em cada etapa do processo de criptografia, essas partes são manipuladas da seguinte maneira:

L se torna igual a R da etapa anterior.

R se torna igual ao valor de L da etapa anterior, combinado com o resultado da função aplicada em R.

$$L_i := R_{i-1} \text{ e } R_i := L_{i-1} \oplus f_i(R_{i-1})$$

Esse processo é repetido várias vezes, o que garante que até mesmo pequenas alterações na entrada resultem em grandes mudanças na saída criptografada, tornando o sistema mais seguro e difícil de ser quebrado.



Quando trabalhamos com uma rede de Feistel, podemos facilmente reverter o processo de criptografia para recuperar os dados originais. Suponha que estamos na i-ésima rodada da rede e que temos as saídas L_i e R_i dessa rodada. Para recuperar os valores da rodada anterior, L_{i-1} e R_{i-1}, seguimos um processo simples:

- Primeiro, definimos R_{i-1} como igual a L_i. Isso porque, em uma rede de Feistel, L_i sempre vem diretamente de R da rodada anterior.
- Depois, calculamos L_{i-1} pegando o valor atual de R_i e combinando-o com o resultado da função aplicada em R_{i-1}.

$$L_{i-1} := R_i \oplus f_i(R_{i-1})$$

Ao repetir esse procedimento para cada rodada, conseguimos inverter todo o processo de criptografia, retornando à mensagem original.

O *Data Encryption Standard* (DES) utiliza uma rede de Feistel com 16 etapas (ou rodadas). Ele trabalha com blocos de dados de 64 bits e utiliza uma chave de 64 bits para a criptografia. No entanto, desses 64 bits da chave, apenas 56 são efetivamente utilizados para a criptografia, pois 8 bits são descartados logo no início do processo. Assim, o DES é descrito como uma função que combina uma chave de 56 bits com um bloco de 64 bits de dados para produzir um novo bloco criptografado de 64 bits.

A chave de 56 bits passa por um processo chamado *key schedule*, que gera 16 subchaves, cada uma com 48 bits, para serem usadas em cada uma das 16 rodadas de criptografia, similar a uma cifra de fluxo.

Antes de começar o processo da rede de Feistel, é aplicada uma permutação inicial dos 64 bits do bloco. o 58^o é colocado na primeira posição,

em seguida vem o 50^o e assim por diante conforme mostrado na Tabela 5.2.1, organizada em 8 linhas para facilitar a leitura. Ao final da Rede de Feistel uma permutação inversa a essa é aplicada.

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

Tabela 5.1: Sequência da permutação inicial

Em cada rodada do DES, a mesma função é aplicada. Essa função recebe uma subchave de 48 bits e metade do bloco de 64 bits (ou seja, 32 bits) como entrada, e produz uma nova sequência de 32 bits como saída. Aqui está como essa função funciona:

1. *Expansão*: Primeiro, os 32 bits do bloco são expandidos para 48 bits,

   Na fase de expansão, começamos copiando o último bit, o 32^o, para a primeira posição e depois copiamos os primeiros 5 bits da sequência. Em seguida, repetimos o 4^o e o 5^o bits. Continuamos com os próximos 4 bits e então repetimos o 8^o e o 9^o bits, e seguimos esse padrão até alcançar o 32^o bit. Por fim, repetimos o primeiro bit da sequência. Com esse processo, exatamente 16 bits são repetidos, transformando uma sequência de 32 bits em uma de 48 bits.

   A Tabela 1 indica a ordem dos bits resultantes, ela está organizada em 8 linhas para facilitar a leitura.

2. *XOR*: Esses 48 bits são combinados com a subchave utilizando a operação XOR (ou exclusivo),

3. *S-Boxes*: O resultado é então dividido em 8 partes, cada uma com 6 bits. Cada uma dessas partes de 6 bits passa por uma substituição através de uma tabela especial chamada *SBox*, que reduz os 6 bits para 4 bits.

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

Tabela 5.2: Sequência de expansão

A Tabela 3 apresenta uma das 8 SBoxes do DES, especificamente a primeira (S_1), usada aqui como exemplo. O processo de leitura das tabelas funciona da seguinte maneira: suponha que a entrada seja a sequência de 6 bits 010110_2. O primeiro e o último bit (0 e 0) determinam a linha, que nesse caso é a linha 00_2, ou seja, a primeira linha da tabela. Os quatro bits restantes (1011_2) formam a coluna, que corresponde à décima primeira coluna. Com essas coordenadas, a saída da tabela será o valor 12, que em binário é representado como 1100_2.

Essa é a fase de *confusão*.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

Tabela 5.3: Tabela SBox 1

4. *Difusão*: A combinação da saída dos oito SBoxes resulta em uma sequência de 32 bits. Esses 32 bits são embaralhados em um processo chamado *difusão*, que espalha as mudanças por todo o bloco, garantindo que até mesmo pequenas alterações na entrada resultem em grandes mudanças na saída.

   O bit da posição 16 vai para a primeira posição seguido do sétimo e assim por diante conforme indicado no diagram abaixo organizado em 4 linhas para facilitar a visualização:

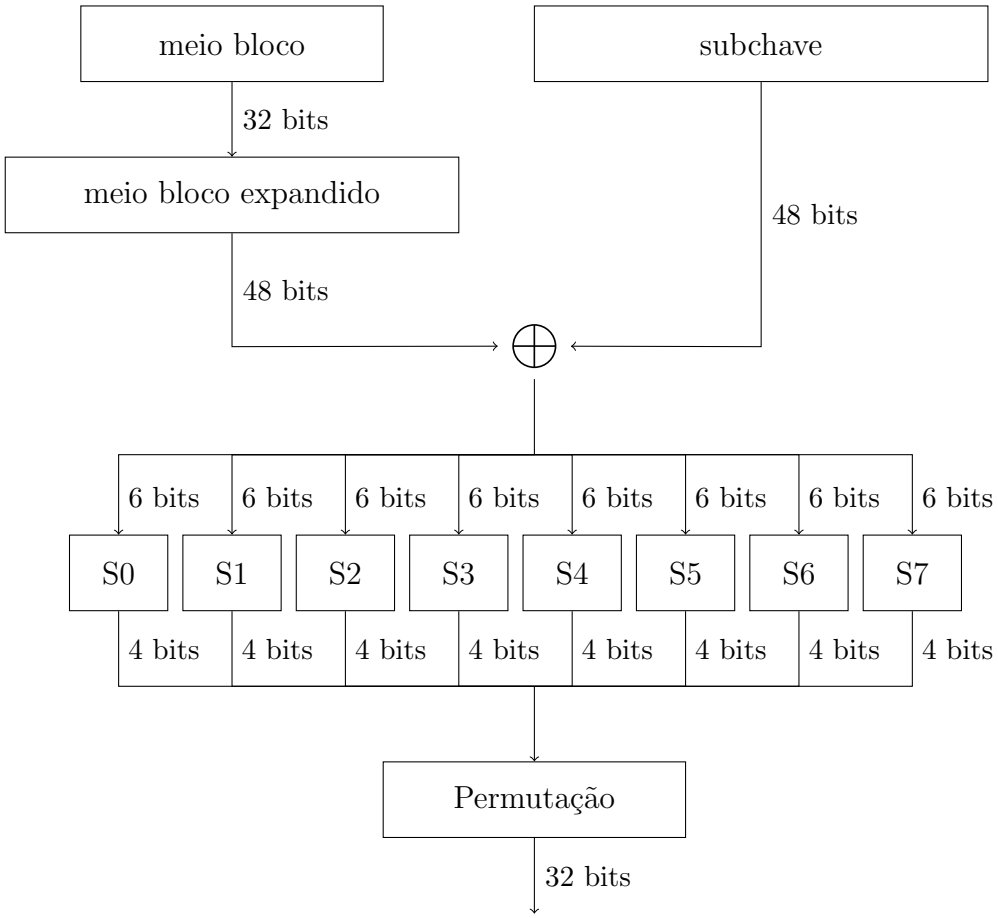


Figura 5.2: Função de Fiestel do DES

16	7	20	21	29	12	28	17
1	15	23	26	5	18	31	10
2	8	24	14	32	26	3	9
19	13	30	6	22	11	4	25

Tabela 5.4: Sequência da permutação

A adoção do DES como padrão de criptografia foi marcada por controvérsias. Um dos pontos mais discutidos foi o descarte de 8 bits da chave original de 64 bits, reduzindo-a para 56 bits sem uma justificativa clara. Na época, uma chave de 56 bits estava no limite da segurança, mas hoje sabemos que ela é vulnerável a ataques de força bruta.

Além do pequeno tamanho da chave, outro aspecto suspeito foi a modificação das S-Boxes pela NSA (Agência de Segurança Nacional dos Estados Unidos) sem grandes explicações antes que o algoritmo fosse adotado como padrão. Anos mais tarde, pesquisadores apresentaram uma técnica chamada *criptoanálise diferencial*, que tornou várias cifras inseguras. No entanto, surpreendentemente, o DES resistiu a essa técnica. Isso levou à suspeita de que os pesquisadores da NSA já conheciam a criptoanálise diferencial e ajustaram o DES de forma que ele se tornasse seguro contra esse tipo de ataque.

Em 1993, um artigo já alertava para a fragilidade dessa chave, propondo um projeto de hardware que, teoricamente, seria capaz de quebrar uma chave de 56 bits em apenas um dia e meio. Essa previsão se tornou realidade em 1998, quando a *Electronic Frontier Foundation* (EFF) – uma organização sem fins lucrativos que defende a privacidade, a liberdade de expressão e os direitos digitais na internet – construiu uma máquina chamada Deep Crack. Este dispositivo, que custou cerca de 250 mil dólares, conseguiu quebrar uma cifra DES em 56 horas, aproximadamente dois dias e meio.

A situação se agravou ainda mais em 2006, quando pesquisadores alemães desenvolveram um hardware de baixo custo — em torno de 10 mil dólares — capaz de realizar um ataque de força bruta ao DES em menos de uma semana. Essa evolução mostra que, enquanto nos anos 80 apenas grandes potências mundiais tinham os recursos para construir máquinas desse tipo, hoje em dia, qualquer entidade com recursos relativamente modestos pode quebrar o DES.

Com essas revelações e avanços, o DES, que antes era o padrão de segurança, é agora considerado totalmente inseguro para proteger informações

sensíveis.

Embora o DES não seja mais considerado seguro devido ao tamanho reduzido de sua chave (56 bits), uma variante conhecida como Triple DES (3DES) ainda é utilizada em algumas aplicações. O 3DES realiza três aplicações consecutivas do DES, sendo a segunda com a inversa da permutação, e utilizando três chaves independentes, resultando efetivamente em uma chave de 168 bits, o que aumenta significativamente a segurança. Vale ressaltar que a versão com apenas duas aplicações do DES não é segura, pois está sujeita ao ataque Meet in the Middle, cuja explicação não será abordada neste livro.

5.2.2 Advanced Encryption Standard (AES)

As crescentes preocupações em torno da segurança do DES, especialmente devido à possibilidade iminente de ataques de força bruta contra sua chave de 56 bits, levaram o NIST (Instituto Nacional de Padrões e Tecnologia dos Estados Unidos) a buscar uma solução mais segura. Em 1997, o NIST organizou um concurso acadêmico internacional para desenvolver um novo padrão de criptografia que pudesse substituir o DES.

Este concurso, chamado Advanced Encryption Standard Competition, incentivou pesquisadores e criptógrafos de todo o mundo a submeterem seus algoritmos de criptografia. Além de criar e apresentar um algoritmo, cada participante também tinha a tarefa de analisar e encontrar possíveis vulnerabilidades nos algoritmos dos outros concorrentes. Isso ajudou a garantir que o novo padrão fosse robusto e resistente a ataques.

Após uma rigorosa avaliação, cinco algoritmos finalistas foram considerados adequados para se tornarem o novo padrão de criptografia. Em abril de 2000, o algoritmo *Rijndael* foi anunciado como o vencedor do concurso. Este algoritmo, desenvolvido por dois criptógrafos belgas, foi então adotado como o novo padrão de criptografia e recebeu o nome de AES (*Advanced Encryption Standard*). Desde então, o AES tem sido amplamente utilizado e é considerado um dos algoritmos de criptografia mais seguros disponíveis atualmente.

O *AES* é um algoritmo de criptografia que opera em blocos de 128 bits. Ele possui três versões, cada uma utilizando um tamanho de chave diferente e um número correspondente de rodadas de criptografia:

AES-128: Usa chaves de 128 bits e realiza 10 rodadas de criptografia.

AES-192: Usa chaves de 192 bits e realiza 12 rodadas de criptografia.

AES-256: Usa chaves de 256 bits e realiza 14 rodadas de criptografia.

Ao contrário do *DES*, que utiliza uma estrutura chamada *rede de Feistel* para sua criptografia, o AES adota uma técnica diferente, conhecida como *rede de substituição e permutação*. Essa técnica envolve a substituição de dados em cada rodada seguida por uma permutação, ou rearranjo, dos bits, o que contribui para a robustez e a segurança do algoritmo.

No *AES*, o bloco de dados a ser criptografado é inicialmente dividido em 8 sequências de 16 bits. Essas sequências são organizadas em uma estrutura conhecida como *estado*, que é representada por uma matriz 4x4, onde cada célula da matriz corresponde a um bloco de 8 bits.

Durante o processo de criptografia, o AES realiza várias rodadas, onde em cada rodada o algoritmo aplica uma série de transformações ao estado. As transformações em cada rodada são as seguintes:

1. *AddRoundKey:* O primeiro passo de cada rodada é adicionar uma subchave ao estado. A subchave é gerada pelo processo de *key schedule* do AES, que produz uma subchave de 128 bits para cada rodada. Essa subchave é combinada com o estado usando a operação XOR, que mistura os bits da chave com os bits do estado de maneira que apenas quem possui a chave correta possa decifrar o bloco.

2. *SubBytes:* No segundo passo, cada byte do estado é substituído por um novo byte utilizando uma tabela de substituição chamada *SBox*, que é bijetora, garantindo que cada entrada tenha uma saída única e distinta. Diferente do DES, onde as SBoxes não possuem uma explicação teórica conhecida, a SBox do AES é baseada em operações sobre corpos finitos de Galois[2]. Isso assegura que a tabela seja inversível, ou seja, a transformação pode ser revertida. Embora essa tabela possa ser gerada dinamicamente, ela é tipicamente preprocessada para otimizar a eficiência durante as operações de criptografia.

   A Tabela 2 mostra o SBpox précomputado e deve ser lida da seguinte maneira: Digamos que a entrada é $3D_{16}$ substitumos esse valor pelo valor da linha 3 coluna D que neste caso é 80_{16}.

   Este passo corresponde a fase de confusão.

[2]Um exemplo de como computar o SBox pode ser encontrado no Apêndice A

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Tabela 5.5: SBox do AES em notação hexadecimal

3. *ShiftRows:* Em seguida, ocorre a etapa de *ShiftRows*, onde as linhas da matriz são rotacionadas. A primeira linha permanece inalterada, enquanto a segunda linha é rotacionada uma posição para a esquerda, a terceira linha duas posições, e a quarta linha três posições. Essa etapa contribui para espalhar os bytes ao longo do estado, aumentando a complexidade da criptografia.

4. *MixColumns:* Por fim, a etapa *MixColumns* trata as quatro colunas do estado como vetores e as multiplica por uma matriz fixa específica. Essa matriz também é construída a partir de operações sobre corpos de Galois, o que garante as propriedades matemáticas necessárias para a segurança e inversibilidade da operação[3]. Essa multiplicação assegura que qualquer alteração em um byte de entrada afete vários bytes de saída, promovendo o efeito de difusão, onde pequenas mudanças nos dados de entrada se propagam amplamente pelo bloco criptografado.

Esses quatro passos são repetidos em cada rodada do AES, contribuindo para a robustez e a segurança do algoritmo.

Na última rodada do *AES*, a etapa de *MixColumns* é substituída por uma segunda aplicação da fase *AddRoundKey*. Essa mudança finaliza o processo de criptografia de maneira segura e garante que a transformação aplicada ao estado seja adequadamente embaralhada antes de ser enviada como saída.

[3]Ver Apêndice A.

O AES foi cuidadosamente projetado para ser eficientemente reversível, mas apenas quando se tem a chave correta. Isso significa que, com a chave, é possível decifrar os dados criptografados de forma rápida e precisa, o que é essencial para a segurança e a utilidade prática do algoritmo.

Até a escrita destas notas, não há conhecimento de um ataque contra o AES que seja mais eficiente do que o ataque de força bruta. Em outras palavras, a única maneira conhecida de quebrar a criptografia do AES seria testar todas as chaves possíveis, o que, devido ao tamanho das chaves usadas pelo AES, é considerada computacionalmente inviável.

5.3 Exercícios

Exercício 25. *Considere a definição de segurança contra ataques ETP em que consideramos qualquer adversário – não apenas os eficientes – e exigimos que a probabilidade de que ele consiga distinguir duas mensagens em que uma delas foi cifrada com probabilidade exatamente $\frac{1}{2}$. Mostre que é impossível construir um sistema que satisfaça essa definição de segurança.*

Exercício 26. *Mostre que a operação $R_i \oplus f_i(R_{i-1})$ na rede de Feistel de fato recupera o valor de L_{i-1}.*

Exercício 27. *Mostre que os modos LCE e Ctr são corretos, ou seja, que em ambos os casos $D(k, E(k, m)) = m$;*

Exercício 28. *Suponha que f seja uma função pseudo-aleatória com chave e blocos ambos de 128 bits e considere o seguinte sistema:*

1. *Seleciona aleatoriamente duas sequência de 128 bits, a chave k e o vetor inicial VI*

2. *Divide a mensagem m em blocos de 128 bits: $m_0, m_1, \ldots, m_{n-1}$ (podemos supor que $|m|$ é múltiplo de 128).*

3. *A cifra $c = c_0||c_1||\ldots||c_{n-1}$ é tal que $c_i = m_i \oplus f_k(VI)$ para $i = 0, ..., n-1$*

4. *Para descriptografar fazemos $c_i \oplus f_k(VI)$ para $i = 0, \ldots, n-1$.*

Esse sistema é seguro? Por que?

Exercício 29. *Suponha que um bit em uma cifra tenha sido alterado por um erro. Qual o efeito disso na mensagem descriptografada caso a cifra tenha sido produzida usando o modo Ctr? E no caso de ter sido produzida usando o modo LCE?*

Capítulo 6

Integridade e Autenticidade

Até agora, exploramos sistemas de criptografia simétricos que têm como principal objetivo garantir a confidencialidade da comunicação entre as partes. No cenário mais poderoso de ataque que estudamos, o adversário tem a capacidade de escolher mensagens e verificar como elas seriam cifradas. Também discutimos sistemas que são seguros contra esse tipo de ataque.

Neste capítulo, vamos expandir nossa discussão para abordar outros dois problemas essenciais em criptografia:

> *Integridade:* Como garantir que a mensagem recebida não foi alterada durante a transmissão.
>
> *Autenticidade:* Como garantir que a mensagem foi realmente enviada por quem deveria enviá-la.

A importância dessas garantias é clara. Imagine, por exemplo, uma comunicação entre um cliente e seu banco. O banco envia uma mensagem cobrando uma dívida de um terceiro, e o cliente autoriza um pagamento de R$1000,00. É essencial que o cliente tenha a certeza de que a mensagem realmente veio do banco e não é um golpe (autenticidade). Além disso, tanto o cliente quanto o banco precisam ter certeza de que ninguém alterou o valor autorizado (integridade).

Um erro comum é pensar que os sistemas de criptografia que discutimos até agora também garantem a integridade de uma mensagem. Na realidade, esses sistemas não oferecem nenhuma proteção contra modificações no conteúdo da mensagem, e alguns deles são até *maleáveis*. Isso significa que a

cifra pode ser facilmente alterada por um adversário para produzir um efeito desejado.

Exemplo 11. *Vamos considerar um exemplo para ilustrar como um adversário pode injetar uma mensagem m' em uma cifra. Suponha que o adversário conheça a mensagem original m e queira enganar o destinatário, fazendo-o acreditar que a mensagem recebida é m'.*

O adversário pode interceptar a cifra c, que foi gerada pela operação $c = G(k) \oplus m$, e substituí-la por uma nova cifra c', calculada da seguinte forma:

$$c' = c \oplus m \oplus m' = G(k) \oplus m \oplus m \oplus m' = G(k) \oplus m'$$

Note que, quando o destinatário descriptografar c' usando a chave k, ele vai recuperar m', exatamente como o adversário planejou.

Embora a cifra c seja perfeitamente segura do ponto de vista da confidencialidade, ela não garantiu a integridade da mensagem. Isso permitiu ao adversário alterar o conteúdo da mensagem mesmo sem que ele possuisse a chave.

Exemplo 12. *Outro erro comum é tentar garantir a integridade de uma mensagem cifrada usando um checksum. Um* checksum[1] *é como uma "assinatura digital" da mensagem, gerada por uma função que produz uma sequência de bits quase única para cada mensagem. (Vamos explorar isso em mais detalhes no próximo capítulo.)*

A ideia seria a seguinte: você gera um checksum para a mensagem m, anexa esse checksum ao final da mensagem, e só então criptografa tudo. A lógica por trás disso é que, ao receber a cifra, o destinatário pode descriptografar, recuperar a mensagem e seu checksum, e verificar se o checksum corresponde à mensagem. Se alguém alterar a mensagem durante o envio, o checksum não corresponderá mais, e o destinatário poderá perceber que houve uma alteração e descartar a mensagem.

Embora isso pareça uma boa ideia, ela não oferece proteção contra um adversário mal-intencionado. Se um adversário consegue alterar a mensagem m, ele também pode alterar o checksum correspondente, enganando o destinatário. Checksums funcionam bem para detectar erros acidentais, como pequenos ruídos que possam corromper a mensagem durante a transmissão, mas não são eficazes contra ataques intencionais.

[1] Estudaremos isso no Capítulo **??**

6.1 Código de Autenticação de Mensagem

Para garantir que uma mensagem não foi alterada, usamos um *Código de Autenticação de Mensagem* (CAM). Um CAM consiste em três algoritmos:

> *Geração de chave Gen*: Este algoritmo cria uma chave secreta, que é como uma senha especial que será usada para criar e verificar o código de autenticação.
>
> *Criação de código Mac*: recebe uma chave k e uma mensagem m e devolve um código (*tag*) t.
>
> *Verificação de código Ver*: receve uma chave k, uma mensagem m e um código t e verifica se mensagem é váilda.

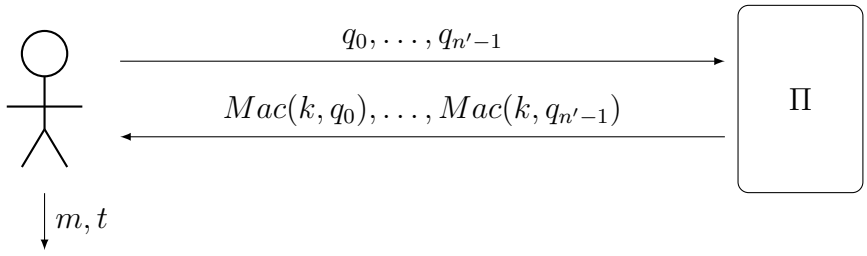
Um sistema de CAM é considerado *correto* quando, ao usar a mesma chave secreta tanto para gerar o código quanto para verificá-lo, a verificação confirma que o código corresponde à mensagem original. Em outras palavras, se você usa a mesma chave para criar o código e depois verificar esse código com a mensagem original, o algoritmo de verificação deve confirmar que a mensagem não foi alterada, garantindo assim a integridade da mensagem.

Vamos imaginar um cenário em que um adversário tem acesso a um sistema CAM e pode pedir a esse sistema para gerar códigos de autenticação para várias mensagens à sua escolha. O objetivo do adversário é criar uma nova mensagem, diferente das que ele já pediu, junto com um código que seja aceito como válido pelo sistema.

Um sistema CAM é considerado *seguro contra falsificação* se, mesmo com todo esse acesso e poder de cálculo, o adversário não conseguir criar um par de mensagem e código que seja aceito, a não ser com probabilidade desprezível.

Vamos descrever como isso funciona:

1. O sistema gera uma chave.
2. O adversário pode consultar o sistema pedindo que gere códigos de autenticação para várias mensagens.
3. O desafio do adversário é criar uma nova mensagem, que não foi consultada, e inventar um código de autenticação que o sistema aceite como válido para essa mensagem.



Um sistema de autenticação de mensagem, é considerado *seguro contra falsificação* se, para qualquer adversário eficiente, a chance de ele conseguir criar um código válido para uma nova mensagem é desprezível.

6.1.1 EBC-CAM

Agora que entendemos o sistema de autenticação de mensagem e definimos uma noção de segurança, o próximo passo é mostrar como construir concretamente tal sistema, detalhando as suposições que faremos.

Começaremos com uma construção básica, que é válida para mensagens de tamanho fixo, utilizando o modo EBC (*Encadeamento de Bloco Cifrado*).

Vamos supor que o tamanho da mensagem m seja um múltiplo de n, o tamanho do bloco de uma função pseudoaleatória (FPA) que chamaremos de f. Podemos dividir a mensagem m em blocos $m_1, m_2, \ldots, m_l$, onde cada bloco m_i tem tamanho n.

A construção funciona da seguinte maneira:

Mac: Este algoritmo gera um código de autenticação t para a mensagem m usando a chave k.

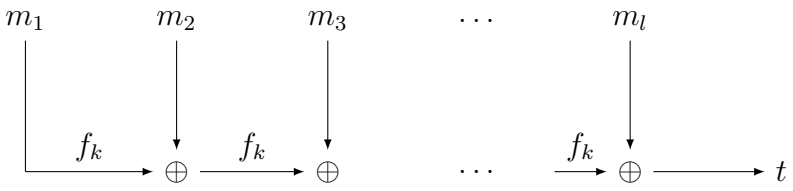
Primeiro, definimos t_0 como um bloco de zeros de tamanho n. Em seguida, para cada bloco m_i da mensagem, calculamos t_i usando a função f e a operação XOR, que combina o resultado do bloco anterior com o bloco atual:

$$t_i := f_k(t_{i-1} \oplus m_i)$$

O resultado final t_l é o código de autenticação.

Ver: Este algoritmo verifica se o código t corresponde à mensagem m quando gerado pela mesma chave k.

$$Ver(k, m, t) := \begin{cases} 1 & \text{se} \quad t = Mac(k, m) \\ 0 & \text{c.c.} \end{cases}$$



Teorema 6. *O sistema de autenticação EBC é seguro contra falsificação para mensagens de tamanho fixo.*

O EBC-CAM é uma técnica específica para gerar códigos de autenticação. Ele funciona de forma semelhante ao modo de operação EBC (Cipher Block Chaining), mas com duas diferenças importantes:

1. No EBC, usamos um vetor inicial (VI) aleatório para começar a cifrar os blocos da mensagem. Já no EBC-CAM, esse vetor inicial é substituído por um bloco cheio de zeros.

2. No EBC, cada bloco cifrado da mensagem é exposto, enquanto no EBC-CAM, apenas o último bloco cifrado (o CAM) é mostrado.

O EBC-CAM é originalmente projetado para funcionar com mensagens de tamanho fixo, mas muitas vezes precisamos lidar com mensagens de tamanhos variados. Para garantir a segurança nesses casos, existem duas estratégias eficazes:

1. Antes de calcular o CAM, adicionamos o tamanho da mensagem no começo do conteúdo. Isso impede que alguém consiga criar dois CAMs iguais para mensagens diferentes (por exemplo, uma mensagem curta poderia ser estendida para enganar o sistema).

2. Outra abordagem é usar duas chaves diferentes. A primeira chave é usada normalmente para gerar o CAM, e a segunda é usada para "ajustar" esse CAM final, reforçando a segurança.

Essas técnicas garantem que o EBC-CAM permaneça seguro, mesmo quando as mensagens variam em tamanho, protegendo contra tentativas de falsificação.

6.2 Criptografia Autenticada

No modelo ETP (Ataque com Escolha de Texto Plano), o adversário pode acessar um oráculo que criptografa mensagens escolhidas para tentar descobrir segredos do sistema. Agora, vamos considerar um modelo ainda mais forte de ameaça.

Além de poder enviar mensagens para serem criptografadas, o adversário também pode consultar como certas cifras seriam decifradas. Isso é muito mais poderoso, porque o adversário não está apenas vendo as mensagens antes de serem criptografadas, mas também pode ver como qualquer cifra seria transformada de volta em texto legível.

Esse novo cenário é raro na prática, porque na maioria das situações reais, os adversários não têm esse tipo de acesso. Entretanto, saber que é possível construir sistemas que permanecem seguros mesmo nesse cenário é fascinante. Isso mostra a robustez e a força das técnicas modernas de criptografia.

Mesmo que seja difícil imaginar uma situação prática onde um adversário tenha tanto poder, explorar esses modelos nos ajuda a desenvolver sistemas de criptografia que sejam ainda mais fortes e seguros. Além disso, entender esses cenários extremos nos dá mais confiança de que os sistemas que usamos todos os dias são realmente protegidos contra uma ampla gama de ameaças possíveis [NY90].

Para entender se um sistema de criptografia é realmente seguro, imaginamos um "jogo" entre o sistema e um adversário, que é alguém tentando quebrá-lo. O jogo funciona assim:

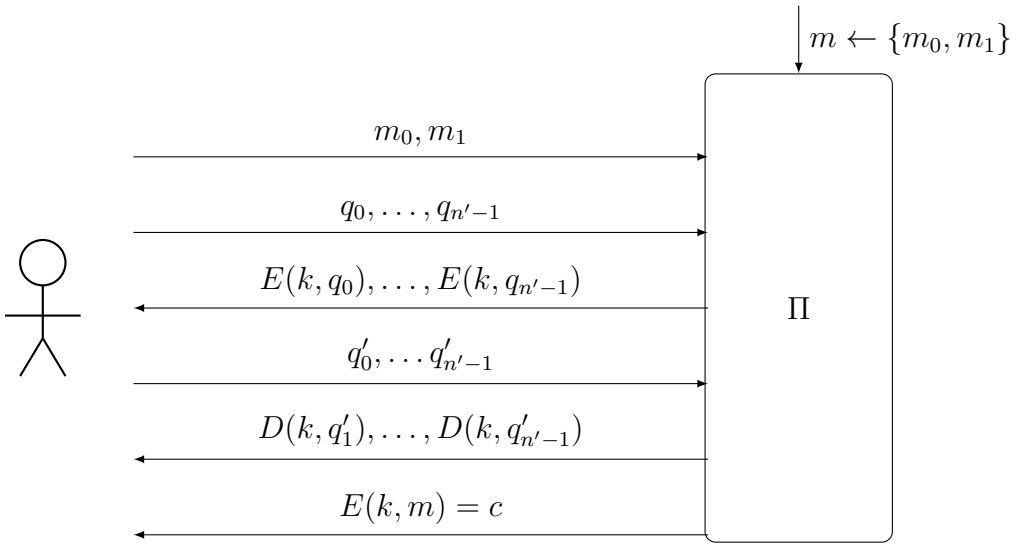
1. O adversário escolhe duas mensagens, que têm exatamente o mesmo tamanho, e as envia para o sistema.

2. O sistema gera uma chave e, em seguida, escolhe aleatoriamente uma das duas mensagens para criptografar. A mensagem escolhida é criptografada, e enviada para o adversário.

3. Durante o jogo, o adversário tem acesso a dois "oráculos":

    - Um oráculo que mostra como qualquer mensagem escolhida pelo adversário seria criptografada.
    - Outro oráculo que revela como qualquer cifra seria decifrada, exceto a cifra que foi devolvida pelo sistema.

4. Com base em todas as informações obtidas, o adversário tenta adivinhar qual das duas mensagens originais foi criptografada.



Dizemos que um sistema é seguro contra o *Ataque com Escolha de Texto Cifrado* (ETC) se para qualquer adversário eficiente existe a chance do adversário vencer o jogo é desprezivelmente maior que 1/2.

Sistemas de criptografia que não garantem a integridade das mensagens não são seguros contra ataques do tipo ETC. Um exemplo que ilustra bem essa vulnerabilidade é o modo Ctr de cifração de blocos.

Exemplo 13. *No modo Ctr, o adversário pode explorar a falta de integridade da seguinte maneira:*

1. *O adversário envia duas mensagens distintas e simples para o sistema: $m_0 = 0^n$ (um bloco de n bits todos iguais a 0) e $m_1 = 1^n$ (um bloco de n bits todos iguais a 1).*

2. *O sistema escolhe uma das mensagens, criptografa e envia a cifra correspondente c de volta ao adversário.*

3. *O adversário altera o primeiro bit da cifra c gerando uma nova cifra c' que é diferente da original.*

4. *O adversário então usa o oráculo de decifragem para verificar o resultado da decifragem de c'. Se o resultado for 10^{n-1}, ele deduz que o sistema criptografou $m_0 = 0^n$. Se o resultado for 01^{n-1}, ele deduz que o sistema criptografou $m_1 = 1^n$.*

Neste cenário, o adversário consegue descobrir qual mensagem foi criptografada apenas alterando a cifra e observando o comportamento do sistema. Isso evidencia a importância de proteger a integridade da mensagem para garantir a segurança contra ataques ETC.

Em um sistema de *criptografia autenticado*, a segurança vai além de simplesmente proteger o conteúdo das mensagens. Aqui, o sistema precisa ser capaz de detectar quando uma cifra foi alterada ou criada de forma maliciosa. Se uma cifra não for válida, o sistema deve acusar um erro e se recusar a tentar decifrá-la.

Uma característica importante de um sistema de criptografia autenticado é sua capacidade de impedir que um adversário produza uma cifra válida de maneira fraudulenta. Se um sistema consegue garantir que qualquer tentativa de falsificação de uma cifra resulte em uma mensagem de erro, dizemos que ele é *seguro contra falsificação*.

Para que um sistema de criptografia seja considerado verdadeiramente seguro e autenticado, ele deve satisfazer duas condições:

- Ser seguro contra ataques do tipo ETC.
- Ser seguro contra falsificação.

Quando um sistema atende a esses dois requisitos, chamamos ele de um *sistema autenticado de criptografia*.

Um *sistema autenticado de criptografia* é a junção de dois elementos essenciais: um esquema de criptografia para proteger o conteúdo das mensagens e um sistema de autenticação para garantir que as mensagens não foram adulteradas. Existem várias maneiras de combinar esses dois sistemas para alcançar segurança completa. Vamos explorar as principais:

- Autenticar e depois criptografar (*mac then encrypt*): Primeiro, geramos um código de autenticação para a mensagem. Em seguida, juntamos esse código com a mensagem original e criptografamos tudo de uma vez.

- Criptografar e autenticar separadamente (*encrypt and mac*): Neste método, primeiro criptografamos a mensagem para protegê-la. Em seguida, geramos um código de autenticação para a mensagem original. Ambos, a cifra e o código de autenticação, são enviados separadamente.

- Criptografar e depois autenticar (*encrypt then mac*): Aqui, começamos criptografando a mensagem. Depois de criptografada, geramos um código de autenticação baseado na cifra resultante.

Teorema 7. *O modelo que primeiro criptografa e depois autentica (*encrypt then mac*) que usa um sistema de criptografia seguro contra CPA e um sistema de autenticação seguro contra falsificação é um sistema de criptografia autenticada seguro contra ataques ETC e seguro contra falsificação.*

Para que o sistema acima seja seguro, é essencial que ele use uma chave para criptografar e outra distinta e independente para autenticar.

Exemplo 14. *Vamos explorar um exemplo onde o uso de uma única chave para criptografia e autenticação pode causar sérios problemas de segurança.*

Imagine que temos uma permutação pseudoaleatória p. Para criptografar uma mensagem m (que tem metade do tamanho de um bloco), fazemos o seguinte:

- *Escolhemos um valor aleatório r com metade dos bits de um bloco.*

- *Combinamos a mensagem m e o valor r e aplicamos a permutação p ($E(k, m) = p_k(m||r)$).*

Podemos provar que essa forma de criptografia é segura contra ataques CPA.

Agora, vamos usar a inversa da permutação p^{-1} para criar um código de autenticação (CAM). Ou seja, $t = p^{-1}(c)$.

Também podemos mostrar que esse CAM é seguro contra falsificação.

Agora, vamos ver o que acontece quando combinamos esses dois sistemas usando o paradigma Encrypt then Mac ***sem*** *usar chaves independentes:*

- *Primeiro, criptografamos a mensagem: $E(k, m) = p_k(m||r)$.*

- *Em seguida, aplicamos o CAM na cifra: $Mac(k, E(k, m)) = p_k^{-1}(p_k(m||r))$.*

Quando aplicamos o CAM na cifra, o resultado é a mensagem original m e o valor aleatório r: $p_k^{-1}(p_k(m||r)) = m||r$.

Isso significa que o código de autenticação revela exatamente o conteúdo da mensagem original, que deveria estar protegido! Em outras palavras, o adversário consegue ver a mensagem inteira simplesmente olhando para o código de autenticação, o que quebra completamente a segurança do sistema.

Esse exemplo mostra por que é importante usar chaves independentes para criptografia e autenticação. Quando não usamos chaves diferentes, corremos o risco de comprometer a segurança do sistema, permitindo que informações confidenciais vazem de maneiras inesperadas.

6.2.1 Comunicação Segura

Um sistema autenticado de criptografia é responsável por garantir três aspectos importantes na comunicação: *confidencialidade* (apenas o destinatário deve ser capaz de ler a mensagem), *integridade* (a mensagem não deve ser alterada) e *autenticidade* (o destinatário deve ter certeza de quem enviou a mensagem). No entanto, mesmo com essas proteções, ainda existem outros tipos de ameaças que precisam ser considerados. Vamos ver três dessas ameaças e como podemos lidar com elas.

Um sistema autenticado de criptografia garante a confidencialidade, a integridade e a autenticidade na comunicação. Existem, porém, outros tipos de ameaça que esse sistema não garante por si só:

Ataques de reordenação: Um adversário pode mudar a ordem das mensagens enviadas, o que pode causar problemas. Imagine um cenário onde as mensagens devem ser processadas em uma ordem específica; se a ordem for alterada, isso pode levar a ações incorretas ou até mesmo prejudiciais.

Para evitar isso, podemos usar um contador que mantém o registro da ordem das mensagens enviadas entre dois participantes. Por exemplo, um contador ctr_{AB} pode ser usado para as mensagens de A para B, e ctr_{BA} para as mensagens de B para A. Assim, se as mensagens chegarem fora de ordem, o sistema detectará e rejeitará a comunicação.

Ataque de repetição: Aqui, o adversário repete uma mensagem legítima. Por exemplo, se alguém envia uma ordem de pagamento, o adversário

pode repetir essa mensagem, tentando fazer com que o pagamento seja processado duas vezes.

Novamente, um contador pode ajudar. Como cada mensagem terá um número único, o sistema saberá se uma mensagem está sendo repetida e poderá rejeitar tentativas de repetição.

Ataque de reflexão: Neste ataque, um adversário intercepta uma mensagem e a envia de volta ao remetente, fazendo parecer que ela veio do destinatário. Isso pode confundir o remetente ou fazer com que ele realize uma ação indesejada.

Podemos adicionar um bit à mensagem que indique a direção da comunicação. Se o bit b_{AB} estiver definido como 1, isso significa que a mensagem foi enviada de A para B. Se estiver definido como 0, significa que a mensagem foi de B para A. Isso impede confusões sobre quem enviou a mensagem.

Outra solução é usar duas chaves independentes: uma k_{AB} para mensagens de A para B e outra k_{BA} para mensagens de B para A. Dessa forma, mesmo que uma mensagem seja enviada de volta, a chave errada será usada, tornando a mensagem inválida.

Ambas as soluções – tanto o uso de contadores quanto o bit de direção – devem ser incorporadas na mensagem antes de ela ser criptografada. Isso garante que a integridade e a autenticidade da mensagem sejam mantidas.

Com isso, concluímos nossa exploração dos sistemas de criptografia simétrica. Utilizando uma abordagem moderna, explicamos as suposições necessárias para garantir a segurança em diferentes sistemas de criptografia e autenticação.

No Capítulo 7, examinamos a suposição mínima necessária para construir os sistemas que discutimos: a existência de funções de mão única. Esse capítulo não só servirá como um resumo do que aprendemos até agora, mas também explicará por que precisamos dessas suposições básicas para provar a segurança em criptografia.

6.3 Exercícios

Exercício 30. *Considere que o adversário sabe que a mensagem* $m = 101010$ *foi cifrada por uma cifra de fluxo e produziu* $c = 110001$. *Que sequencia de bits* c' *ele precisa enviar para o destinatário para fazer com que ele ache que a mensagem original era* $m' = 001011$*?*

Exercício 31. *Mostre que a função encode que insere o tamanho da mensagem no começo e completa o último bloco com uma sequência de* $0s$ *não admite prefixo.*

Exercício 32. *Seja* f *uma função pseudo-aleatória e considere o sistema* Π *uma cifra de bloco que aplica* f *no modo contador. Suponha que Alice e Bob compartilham uma chave secreta* k. *Considere os seguintes cenários:*

1. *Alice enviar* $E(k, m)$ *para Bob que descriptografa usando a chave* k
2. *Alice gera um checksum* $H(m)$ *da mensagem e envia* $H(m)||m$ *para Bob que pode verificar o checksum antes de ler a mensagem*

Algum desses cenários garante que a mensagem lida por Bob é idêntica a mensagem que foi enviada por Alice? Por que? Caso nenhum dos cenários garanta isso, descreva como poderíamos fazê-lo.

Capítulo 7

Funções de Mão Única

Nos Capítulos 4, 5 e 6 mostramos que, sob certas condições, alguns sistemas de criptografia são seguros contra determinados tipos de ataque. Essas condições incluem a existência de geradores de números pseudoaleatórios, funções pseudoaleatórias e permutações pseudoaleatórias. Contudo, é importante destacar que, embora essas condições sejam amplamente aceitas, elas não foram provadas matematicamente, mas sim validadas por meio de experimentos práticos.

Neste capítulo, discutiremos por que não criamos sistemas que comprovadamente atendem a essas condições. Para isso, apresentaremos um conceito fundamental: a existência de funções de mão única, que são funções fáceis de calcular, mas difíceis de reverter. Essas funções são a base para a segurança de sistemas criptográficos. Além disso, explicaremos que provar a existência dessas funções seria o mesmo que provar que o problema P não é igual ao problema NP, um dos grandes desafios não resolvidos na matemática atual.

Este capítulo passará por vários dos principais resultados que já vimos e, assim, serve também como uma revisão do que foi apresentado até agora no livro.

Uma *Função de Mão Única* (FMU) $f : \{0,1\}^n \to \{0,1\}^n$ é uma função que tem duas características principais:

> *Fácil de computar*: Dado um valor x, calcular $f(x)$ é rápido (pode ser feito em tempo polinomial).
>
> *Difícil de inverter*: Dado um resultado y, é muito difícil achar um x que produza y (encontrar uma pré-imagem não é algo que possa ser feito em tempo polinomial).

Em termos mais rigorosos, uma FMU é tal que, se um adversário eficiente recebe y, ele só é capaz de encontrar um x que produza y (ou seja, $f(x) = y$) com uma probabilidade desprezível [DH76, Yao82].

Exemplo 15. *Uma candidata a função de mão é única é a seguinte função, onde x e y são números primos do mesmo tamanho:*

$$f(x, y) := x \cdot y$$

A dificuldade de inverter f está a associada ao problema da fatoração *que é considerado um problema difícil.*

Para certos grupos G com gerador g temos que a seguinte função f é de mão única:

$$f_g(x) := g^x$$

A dificuldade de inverter esta função está associada ao problema do logarítmo discreto.

7.1 Construindo um GNP

Chamamos de *predicado de núcleo duro* um pequeno pedaço de informação, como um único bit, sobre x que a função de mão única mantém escondido [BM84]. Mesmo que o adversário conheça $y = f(x)$ ele não obtem informação adicional sobre esse bit. Em outras palavras, a chance de o adversário adivinhar o bit é desprezivelmente maior do que meio.

Teorema 8 (Goldreich-Levin)**.** *Assuma que uma função de mão única existe f. Então a função $g(x, r) = \langle f(x), r\rangle$ também é uma função de mão única e o predicado $x \oplus r$ é um predicado de núcleo duro para g [GL89].*

Uma vez que conseguimos extrair um bit "seguro" de uma função de mão única, podemos usá-lo para criar um gerador de números pseudoaleatórios (GNP) que aumenta o tamanho da entrada em 1 bit.

Teorema 9. *Se f uma função de mão única com um predicado de núcleo duro nd, então a função $f(s)$ concatenada com $nd(s)$ ($G(s) := f(s)||nd(s)$) é um GNP que expande em 1 bit o tamanho original de s.*

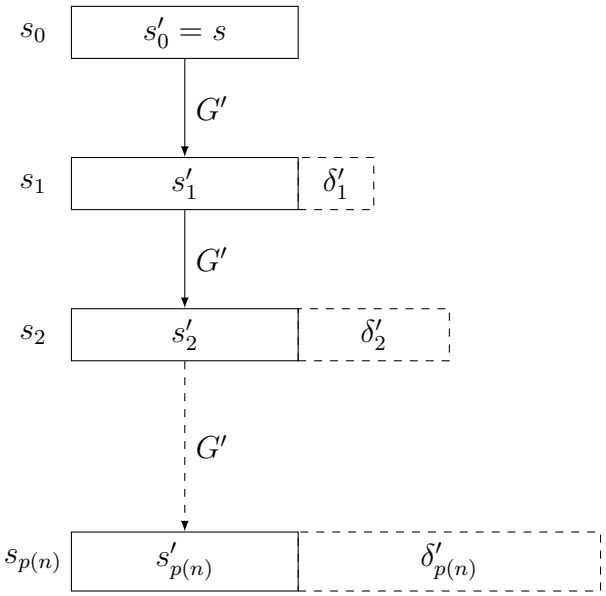
Agora que podemos gerar um GNP que expande a entrada em 1 bit, podemos usá-lo para criar um GNP que expande a entrada em $p(n)$ bits,

onde p é qualquer polinômio e n é o tamanho da entrada s. Seja G um GNP que expande a entrada em 1 bit, e construímos G' da seguinte maneira:

Usamos G para gerar $n + 1$ bits. O último bit gerado é usado como o primeiro bit da saída de G', e os outros bits são usados como uma nova semente para G. Repetimos esse processo até que G' tenha gerado $p(n)$ bits.

Essa construção nos leva ao seguinte teorema:

Teorema 10. *Se existe um GNP que expande a entrada de tamanho n em 1 bit, então, para qualquer polinômio p, existe um GNP que expande a entrada em $p(n)$ bits.*



7.2 Construindo uma FPA e um PPA

Podemos criar uma Função Pseudoaleatória (FPA) a partir de um Gerador de Números Pseudoaleatórios (GNP) que dobra o tamanho da entrada.

Primeiro, aplicamos G ao bloco x, o que gera uma sequência de bits com o dobro do tamanho do bloco. Se o primeiro bit da chave k for 0, pegamos a primeira metade dos bits gerados; se for 1, pegamos a segunda metade. Em seguida, aplicamos G ao bloco escolhido e repetimos o processo para o próximo bit de k, continuando assim até processar todos os bits de k.

Teorema 11 (Yao)**.** *Se G é um GNP que dobra o tamanho da entrada, então a construção acima gera uma FPA.*

Para completar, a partir de uma FPA é possível construir uma PRP usando uma rede de Feistel com três passos em que cada passo usa uma chave distinta.

$$f^{(3)}_{k_1,k_2,k_3}(x) := Feistel_{f_{k_1},f_{k_2},f_{k_3}}(x)$$

Lembrando que:

$$\begin{aligned} Feistel_{f_1,\ldots,f_n}(x) &:= L_n R_n \\ L_i &:= R_{i-1} \\ R_i &:= L_{i-1} \oplus f_i(R_{i-1}) \end{aligned}$$

Teorema 12. *Se f é uma PRF então $f^{(3)}$ é uma permutação pseudoaleatória.*

Resumindo, a existência de uma função de mão única é um ponto de partida na criptografia, pois ela nos permite construir Geradores de Números Pseudoaleatórios (GNP), Funções Pseudoaleatórias (FPA) e Permutações Pseudoaleatórias (PPA).

- No Capítulos 4 vimos que a existência de um GNP é condição suficiente para construir um sistema seguro contra ataques *ciphertext only*.
- Nos Capítulos 5 e 6 aprendemos que, se temos uma FPA, podemos construir sistemas de criptografia ainda mais robustos. Esses sistemas são seguros contra ataques mais avançados, como os ataques onde o atacante pode escolher textos para cifrar (ETP) ou até mesmo descriptografar (ETC). Além disso, a existência de uma FPA é suficiente para criar sistemas que garantem a autenticidade das mensagens.

O teorema a seguir aborda a condição reversa, mostrando que a existência de uma função de mão única não só permite a criação de GNP, FPA e PPA, mas também que essa função de mão única é necessária para que essas construções existam:

Teorema 13. *Se existe um sistema seguro contra ataques* apenas contra o texto cifrado *(ATC) que protega uma mensagem duas vezes maior do que sua chave então existe uma função de mão única.*

Para provar o teorema imagine que temos um sistema de criptografia que é seguro contra ataques cyphertext only que consegue cifrar uma mensagem duas vezes maior do que a chave.

Queremos provar que, se tal sistema seguro existe, então também deve existir uma função de mão única.

Primeiro vamos sortear uma sequencia de bits aleatórios r e então vamos construir uma função de mão única da seguinte forma:

$$f(k, m, r) := E(k, m||r)||m$$

Se um adversário polinomail for capaz de inverter f então ele recupera r e como m está exposto, ele recupera a mensagem $m||r$. Mas neste caso, o sistema de criptografia não seria seguro. Como assumimos que o sistema é seguro, não deve ser possível inverter f.

7.3 Conclusão

Acreditamos que certas funções são de mão única, ou seja, são fáceis de calcular, mas muito difíceis de inverter. Essa crença é apoiada por muitos experimentos, mas ainda não temos uma prova matemática definitiva.

Agora, imagine que conseguíssemos provar que uma função f é realmente de mão única. Por definição, isso significa que podemos calcular $f(x)$ em tempo polinomial. Além disso, se alguém nos der um valor y e disser que $f^{-1}(x) = y$, podemos verificar isso rapidamente, testando se $f(y) = x$.

Em termos técnicos, isso significa que f pode ser usada como um "oráculo polinomial" para verificar se uma solução está correta. Isso coloca o problema de inverter f na classe de problemas NP, que são aqueles problemas para os quais podemos verificar uma solução em tempo polinomial.

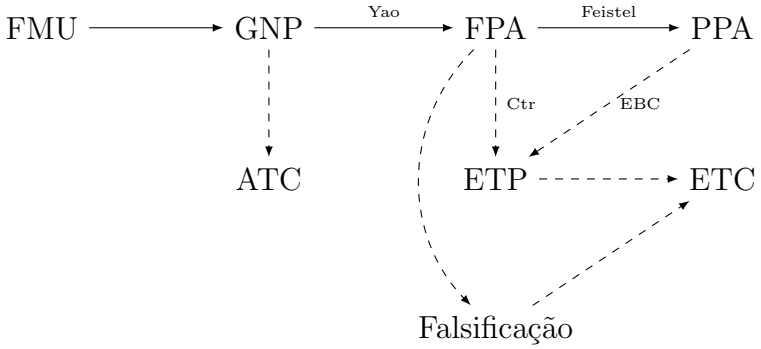
Por outro lado, a própria definição de função de mão única diz que inverter f não pode ser feito de maneira polinomial.

A partir disso, podemos concluir que, se uma função de mão única existe, então a classe de problemas que podem ser resolvidos em tempo polinomial (P) é diferente da classe de problemas onde as soluções podem ser verificadas em tempo polinimal (NP). Ou seja, provar que uma função é de mão única equivale a provar que P é diferente de NP.

Essa questão – a relação entre os problemas P e NP – é o maior problema em aberto na ciência da computação e um dos mais importantes em toda a matemática. Dado que ainda não conseguimos resolver essa questão, nos

contentamos em validar empiricamente que uma função é de mão única, que um gerador de números pseudoaleatórios é seguro, ou que uma permutação pseudoaleatória é segura. A partir dessas validações, podemos então provar a segurança de diversos sistemas de criptografia, como fizemos nos capítulos anteriores.

O diagrama abaixo resume as construções que estudamos neste capítulo. Se existir uma função de mão única, então é possível provar que existem GNPs (Geradores de Números Pseudoaleatórios), FPAs (Funções Pseudoaleatórias) e PPAs (Permutações Pseudoaleatórias) seguras. Com esses elementos, poderíamos provar que cifras de fluxo e cifras de bloco podem ser usadas para criar sistemas seguros contra falsificação e todos os tipos de ataques que discutimos, incluindo ataques do tipo ETC. No entanto, infelizmente, ainda não conseguimos provar a existência de funções de mão única.



7.4 Exercícios

Exercício 33. *Mostre que $f(x, y) = x \cdot y$ sem as restrições de que x e y sejam primos e do mesmo tamanho não é uma função de mão única.*

Parte II

Criptografia Assimétrica

Capítulo 8

Funções de Hash

Uma função de hash é uma ferramenta que pega uma sequência de dados de qualquer tamanho e a transforma em uma sequência curta e de tamanho fixo. Essa sequência curta que chamaremos de *resumo*, mas que em inglê é conhecida como *digest* ou *checksum*.

Quando estudamos funções de hash em Estrutura de Dados, o objetivo é geralmente otimizar o acesso a uma lista, permitindo que encontremos elementos de maneira muito rápida, em tempoconstante. Para conseguir isso, tentamos minimizar as colisões, ou seja, tentamos garantir que diferentes entradas não acabem no mesmo local da lista. Isso faz com que a busca seja mais eficiente.

No entanto, em aplicações de criptografia, evitar colisões é ainda mais crucial. Se duas entradas diferentes resultarem no mesmo digest, isso pode criar vulnerabilidades que adversários podem explorar para comprometer a segurança do sistema.

Uma função de hash criptográfica é uma função H que pega qualquer sequência de bits como entrada e a transforma em uma sequência de bits de tamanho fixo.

Um aspecto fundamental de uma função de hash criptográfica é sua *resistência a colisões*. Isso significa que deve ser extremamente difícil encontrar duas entradas diferentes que resultem na mesma saída. Para explicar esse conceito, podemos imaginar um jogo onde o objetivo do adversário é encontrar essas duas entradas diferentes.

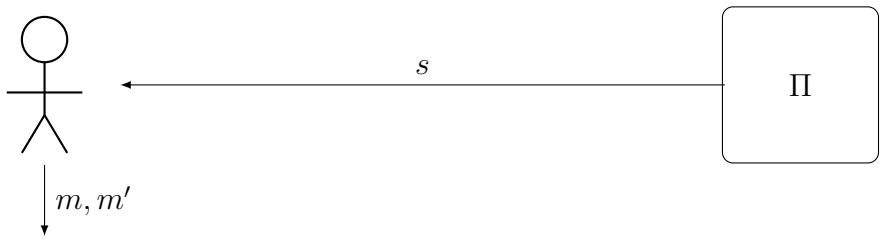
Aqui está como podemos imaginar esse jogo:

1. Primeiro, o sistema gera uma semente s usando um algoritmo. Essa semente não precisa ser secreta.

2. Em seguida, o adversário recebe essa semente s. A semente serve como um ponto de partida para tentar encontrar duas mensagens que, ao serem processadas pela função de hash, resultem no mesmo digest.

3. O adversário então tenta encontrar uma colisão, ou seja, um par de mensagens, m e m', que tenham o mesmo digest ($H_s(m) = H_s(m')$).

O sistema é considerado *seguro contra colisões* se nenhum adversário eficiente é capaz de encontrar uma colisão com probabilidade considerável.



A semente usada na definição da segurança para funções de hash tem um propósito estritamente técnico. Sem uma semente, um adversário poderia pré-calcular colisões.

Na prática, normalmente usamos funções de hash sem uma semente e as consideramos seguras contra colisões quando isso é validado empiricamente, ou seja, por meio de testes extensivos e análises práticas.

A resistência à colisão é uma das propriedades mais importantes de uma função de hash, e é uma das mais difíceis de alcançar. Ela garante que seja extremamente difícil encontrar duas entradas diferentes que resultem na mesma saída. Essa propriedade é mais forte do que outras propriedades que também desejamos em funções de hash, como:

> *Resistência contra colisões em alvos específicos*: Dado um valor s e uma entrada m, nenhum adversário eficiente deve ser capaz de encontrar outra entrada m' tal que $H_s(m) = H_s(m')$ com uma probabilidade significativa. Isso significa que, mesmo se alguém souber a saída $H_s(m)$, ele não deve ser capaz de encontrar outro valor que produza o mesmo resultado.
>
> *Resistência contra preimagem*: Dado um valor s e uma saída y, nenhum adversário eficiente deve ser capaz de encontrar uma entrada m tal que $H_s(m) = y$ com uma probabilidade considerável. Isso significa que,

> mesmo se alguém souber o resultado de uma função de hash, ele não deve ser capaz de descobrir qual entrada gerou aquele resultado.

Toda função de hash está vulnerável a ataques do tipo força bruta. Suponha que temos uma função H_s, que mapeia qualquer sequência de bits em uma saída de n bits. Se calcularmos $H_s(m)$ para uma sequência grande de diferentes entradas, eventualmente encontraremos uma colisão. Isso é garantido pelo princípio da casa dos pombos: se temos mais itens do que "caixas" para colocá-los, pelo menos duas entradas diferentes acabarão na mesma caixa (ou seja, produzirão o mesmo digest).

Na verdade, se assumirmos que a função de hash se comporta como uma função aleatória, podemos mostrar que a probabilidade de encontrar uma colisão é maior que 50% se testarmos cerca de $\sqrt{n}$ entradas diferentes. Esse resultado é conhecido como o *paradoxo do aniversário*. Ele é chamado assim porque, por exemplo, em um grupo de 23 pessoas (pense um time de futebol, com titulares, reservas e o técnico), a chance de duas delas terem o mesmo aniversário é maior que 50%. Embora 23 pessoas pareça um número pequeno para essa coincidência, a matemática por trás do paradoxo explica essa alta probabilidade de colisão.

Portanto, se quisermos um sistema de hash que ofereça segurança equivalente a uma função aleatória com uma chave de 128 bits, precisamos usar uma função de hash que produza uma saída de pelo menos 256 bits.

À primeira vista, encontrar uma colisão qualquer pode parecer inofensivo, já que não controlamos quais valores vão colidir. No entanto, esse ataque pode ser mais perigoso do que parece.

O ataque do aniversariante exige que sejam testadas cerca de $\sqrt{n}$ mensagens diferentes para que a probabilidade de encontrar uma colisão seja grande. Aqui, n é o número de possíveis valores de hash. O ponto crucial é que essas mensagens não precisam ser aleatórias. Podemos gerar mensagens de forma inteligente, utilizando variações controladas, para aumentar nossas chances de sucesso no ataque.

Vamos ilustrar isso com um exemplo. Suponha que um grupo de alunos estejam tentando forjar uma grande quantidade de variações de uma avaliação de um professor, de forma que todas pareçam legítimas, mas tenham pequenas diferenças, permitindo explorar o ataque do aniversariante.

Considere as seguintes frases:

> O projeto foi *executado/ realizado/ concluído* com *sucesso/ eficiência/ precisão*, e os *resultados/ objetivos* foram *atingidos/ superados*

> dentro do *prazo/ tempo/ cronograma* previsto. O grupo demonstrou *dedicação/ comprometimento/ esforço* durante todo as aulas.

As palavras em itálico podem ser trocadas entre si sem alterar o sentido geral da mensagem. Com essa abordagem, podemos gerar 324 variações diferentes da frase (3 opções para a primeira substituição, 3 para a segunda, 2 para a terceira, e assim por diante).

Se precisássemos gerar um conjunto de 2^{32} mensagens, poderíamos elaborar um texto com pelo menos 32 palavras onde cada uma tem pelo menos um sinônimo, o que nos permitiria criar um número imenso de versões da mesma mensagem, todas com significado equivalente.

8.1 Construções

A maioria das funções de hash que usamos hoje seguem uma estrutura conhecida como *Merkle-Damgård*. Essa construção é uma maneira de estender a segurança de uma função de compressão (que processa mensagens de tamanho fixo) para trabalhar com mensagens de qualquer tamanho, mantendo a resistência a colisões [Mer90, Dam90].

Vamos explorar como essa construção funciona, passo a passo.

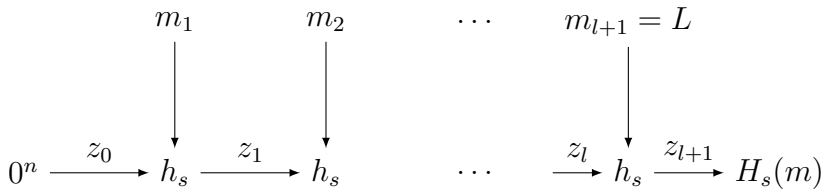
Imagine que temos uma função de hash h que pode processar apenas blocos de dados de tamanho fixo – digamos, ela pega uma entrada de $2n$ bits e a comprime para um resumo de n bits. Esse tipo de função é chamada de *função de compressão*.

Vamos agora entender como construímos uma função de hash completa usando essa ideia:

- Suponha que temos uma mensagem m de tamanho arbitrário. Primeiro, dividimos essa mensagem em blocos menores de tamanho n bits. Se o tamanho total da mensagem não for um múltiplo exato de n, completamos o último bloco com zeros (isso é chamado de *padding*).

- Definimos um valor inicial como um bloco de 0s que será usado para começar o processo de hashing.

- Agora, processamos cada bloco da mensagem, um de cada vez. Aplicamos a função de compressão h usando o bloco resultado anterior e o bloco atual da mensagem ($z_i = h(z_{i-1}||m_i)$).



Teorema 14 (Merkle-Damgård)**.** *Se a função de compressão h para uma mensagem de tamanho fixo é resistente a colisão então a função de hash H construída com o modelo* Merkle-Damgård *para mensagens de tamanho arbitrário é resistente a colisão.*

8.1.1 SHA-1

O *Secure Hash Algorithm 1* (SHA-1) é um algoritmo de hash que pertence à mesma família do MD4. Ele é projetado para pegar uma entrada de tamanho arbitrário (desde que seja menor que 2^{64} bits) e produzir um digest de 160 bits.

Antes que o SHA-1 possa processar a mensagem, ele precisa prepará-la de uma forma específica. Esse processo envolve algumas etapas importantes:

Preenchimento ("padding"): Primeiro, a mensagem original é "preenchida" para que seu tamanho final seja um múltiplo de 512 bits. Esse preenchimento é feito adicionando uma sequência especial de bits ao final da mensagem.

A sequência de padding começa com um 1 seguido por uma série de 0s e, no final, o tamanho original da mensagem em bits.

O objetivo é garantir que, após o padding, o tamanho total da mensagem seja um múltiplo de 512 bits. Isso é importante porque o SHA-1 processa a mensagem em blocos de 512 bits.

- *Divisão da Mensagem em Blocos*: Após o padding, a mensagem completa é dividida em blocos de 512 bits cada.

Com esses blocos prontos, o SHA-1 pode então aplicar seu algoritmo para processar cada bloco e gerar o digest final de 160 bits. Esse digest é uma representação compacta e segura da mensagem original, útil para garantir a integridade e a autenticidade dos dados em diversas aplicações de segurança.

Na construção Merkle-Damgåevrd, a função de hash h desempenha um papel crucial, processando os blocos de dados para gerar o digest final. Essa função h consiste em 80 rodadas de cálculos, onde a mensagem é processada em partes para garantir que o resultado final seja seguro e resistente a ataques.

O processo começa com o *message schedule*, que gera 80 strings de 32 bits cada, chamadas $W_0, \ldots, W_{79}$, a partir do bloco x_i da mensagem que está sendo processado. Essas strings são usadas em cada uma das 80 rodadas de processamento da função de hash.

Antes de começar as rodadas, são definidos valores iniciais fixos para cinco variáveis:

$A = \texttt{67452301}$

$B = \texttt{EFCDAB89}$

$C = \texttt{98BADCFE}$

$D = \texttt{10325476}$

$E = \texttt{C3D2E1F0}$

Esses valores serão alterados em cada uma das 80 rodadas para ajudar a processar a mensagem.

Durante as 80 rodadas, os valores A, B, C, D, E são atualizados usando a seguinte fórmula:

Atualiza A: O novo valor de A é calculado somando o valor de E, uma função específica $f_t(B, C, D)$ que depende dos valores de B, C, D, uma versão rotacionada de A, a string W_j correspondente à rodada, e uma constante K_t.

Desloca os Valores: Os valores de B, C, D, E são então ajustados para a próxima rodada, com B sendo rotacionado 30 bits para a esquerda.

Para adicionar uma camada extra de complexidade e segurança, as constantes K_t e as funções $f_t(B, C, D)$ mudam a cada 20 rodadas:

1. *Rodadas 0-19:*

   $K_t := \texttt{5A827999}$

$$f_t(B, C, D) := (B \wedge C) \vee (B \wedge D)$$

2. *Rodadas 20-39:*

$$K_t := \texttt{6ED9EBA1}$$
$$f_t(B, C, D) := B \oplus C \oplus D$$

3. *Rodadas 40-59:*

$$K_t := \texttt{8F1BBCDC}$$
$$f_t(B, C, D) := (B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$$

4. *Rodadas 60-79:*

$$K_t := \texttt{CA62C1D6}$$
$$f_t(B, C, D) := B \oplus C \oplus D$$

Essas mudanças ajudam a garantir que a função de hash seja resistente a colisões e outras formas de ataque, tornando-a segura para uso em criptografia.

Em fevereiro de 2017, um grupo de pesquisadores da Google, em parceria com uma universidade holandesa, anunciou ter conseguido realizar um ataque prático ao algoritmo de hash SHA-1, demonstrando que é possível encontrar colisões – ou seja, duas entradas diferentes que produzem o mesmo hash.

Os pesquisadores estimaram que, em um ataque de força bruta, levaria cerca de um ano utilizando aproximadamente 12 milhões de processadores gráficos (GPUs) para encontrar uma colisão no SHA-1. No entanto, ao explorar vulnerabilidades específicas do SHA-1, eles conseguiram reduzir drasticamente esse número, realizando o ataque com apenas 110 processadores[1].

Para provar a eficácia do ataque, eles criaram dois documentos PDF diferentes. Cada documento continha um infográfico explicando o ataque, mas com cores diferentes. O impressionante é que, apesar de serem documentos distintos, ambos produziam exatamente o mesmo hash quando processados pelo SHA-1.

Este ataque prático ao SHA-1 foi um sinal claro de que o algoritmo não é mais seguro para uso em aplicações que exigem alta segurança. A recomendação atual é utilizar algoritmos de hash mais robustos e confiáveis, como o SHA-256 ou o SHA-3, que oferecem uma resistência muito maior contra colisões e outros tipos de ataques.

[1] https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html

8.2 Aplicações

As funções de hash são ferramentas essenciais em protocolos de segurança, com diversas aplicações que garantem a integridade e a segurança dos dados. Nesta seção, vamos explorar quatro aplicações distintas dessas funções:

Código de Autenticação de Mensagem baseado em Hash (HMAC): Um sistema de autenticação amplamente utilizado, que usa funções de hash para assegurar que uma mensagem não foi alterada e que sua origem é autêntica. HMAC é especialmente popular em protocolos de comunicação segura, como SSL/TLS.

Derivação de Subchaves a Partir de Outras Chaves: Funções de hash podem ser usadas para gerar subchaves seguras a partir de uma chave principal. Isso é útil em sistemas onde uma única chave mestre precisa ser subdividida em várias chaves menores para diferentes finalidades, mantendo a segurança geral do sistema.

Derivação de Chaves a Partir de Senhas: Em muitos sistemas, é necessário converter uma senha em uma chave criptográfica. As funções de hash desempenham um papel crucial nesse processo, transformando a senha em uma chave segura que pode ser usada para criptografia, garantindo proteção contra ataques como força bruta.

Essas aplicações demonstram a versatilidade e a importância das funções de hash em várias áreas da segurança da informação. Vamos examinar cada uma delas em mais detalhes.

8.2.1 HMAC

No capítulo anterior, discutimos como criar um sistema de autenticação para mensagens de tamanho arbitrário usando um esquema semelhante ao modo CBC. Agora, vamos explorar uma abordagem alternativa chamada *Hash-and-MAC*.

Em vez de autenticar diretamente a mensagem de tamanho arbitrário, a ideia por trás do Hash-and-MAC é primeiro comprimir a mensagem em um valor de tamanho fixo, o resumo, usando uma função de hash. Em seguida, aplicamos um sistema de autenticação para esse digest, como se fosse uma

mensagem de tamanho fixo. Esse método é muito eficaz e é amplamente utilizado em protocolos de segurança.

Considere que temos um sistema de autenticação para mensagens de tamanho fixo Π_M e uma função de hash H resistente contra colisão que produz um digest de tamanho n.

O sistema *Hash-and-MAC* Π para mensagens de tamanho arbitrário é construído da seguinte maneira:

> *Geração de Chaves*: Gera uma chave composta por duas partes: k_M, a chave usada pelo sistema de autenticação, e s, a chave usada pela função de hash.
>
> *Geração do CAM*: Para autenticar uma mensagem m de tamanho arbitrário, primeiro calculamos o *digest* $H_s(m)$ usando a função de hash. Em seguida, aplicamos o sistema de autenticação de tamanho fixo Mac_M ao *resumo* para gerar o código de autenticação ($Mac(k, m) := Mac_M(k_M, H_s(m))$).
>
> *Verificação*: Verificamos a mensagem exatamente da mesma forma como nos modos que já apresentamos.

Teorema 15. *Se Π_M é um sistema de autenticação seguro contra falsificação e H é um hash resistente a colisão então a construção acima é segura contra falsificação.*

Demonstração. A ideia central da prova é a seguinte: Imagine um adversário que tenta quebrar a segurança do sistema, consultando os códigos de autenticação (tags) de uma série de mensagens Q. Agora, vamos supor que, de alguma forma, o adversário consiga gerar um código válido para uma nova mensagem m que não estava na lista Q.

Se isso fosse possível, teríamos dois cenários:

1. Suponha que existe uma mensagem m' em Q tal $H_s(m') = H_s(m)$. Isso contraria a segurança contra colisões de H.

2. Se não há colisão, então o adversário teria conseguido gerar um código de autenticação válido para uma nova mensagem, a partir do digest $H_s(m)$, que tem um tamanho fixo. Isso contradiria a hipótese de que o sistema de autenticação Π_M é seguro contra falsificação.

□

O HMAC é um dos sistemas de autenticação mais amplamente utilizados em protocolos de segurança. Ele funciona aplicando uma função de hash duas vezes sobre a combinação de uma chave secreta com os dados, utilizando duas constantes diferentes, chamadas `ipad` e `opad`. Exemplos de sua aplicação incluem os protocolos TLS/SSL, que garantem a integridade e autenticidade de dados transmitidos em conexões seguras, como o HTTPS, o IPsec, que autentica e protege pacotes de dados em redes IP, como em VPNs, e o SSH, que assegura a integridade dos dados durante transferências seguras.

Vamos entender como o HMAC funciona:

- Primeiro, combinamos a chave k com a constante `ipad` usando a operação XOR.
- Em seguida, aplicamos a função de hash H_s ao resultado dessa combinação junto com a mensagem m.
- Depois disso, combinamos a chave k com a constante `opad` (também usando XOR).
- Finalmente, aplicamos a função de hash H_s a essa nova combinação junto com o resultado do primeiro hash.

$$Mac(k, m) := H_s((k \oplus \texttt{opad})||H_s(k \oplus \texttt{ipad})||m)$$

O processo de verificação é identico aos demais que vimos anteriormente.

8.2.2 Funções de Derivação de Chaves

Uma *função de derivação de chaves* (FDC) é uma ferramenta essencial em criptografia, projetada para gerar uma chave segura a partir de um material que contém uma quantidade significativa de entropia, chamado de *key material*.

Existem duas situações comuns em que uma FDC é útil:

Preparação de Material Criptográfico: Às vezes, o material que você tem (como uma senha) não está suficientemente seguro ou preparado para ser usado diretamente como uma chave criptográfica. Nesse caso, a FDC ajuda a transformar esse material em uma chave segura.

Derivação de Subchaves: Quando você tem uma chave inicial e precisa gerar várias subchaves diferentes a partir dela, a FDC pode ser usada para criar essas subchaves de forma segura.

Formalmente, uma FDC recebe como entrada:

- O material da chave δ, que é o material base para derivar a chave.
- O tamanho l da chave que queremos produzir.
- Opcionalmente, um valor chamado salt (r), que é uma chave pública adicional usada para garantir que a saída seja única, mesmo se o mesmo key material for usado novamente.
- Um valor contextual c, que pode ser utilizado para fornecer informações adicionais sobre o contexto em que a chave será usada.

Uma FDC segura deve usar esses valores para produzir uma sequência *pseudoaleatória* de bits do tamanho desejado l.

Uma FDC típica funciona em duas etapas:

Extração: Nesta etapa, a FDC recebe o δ e, opcionalmente, um valor público conhecido como salt. O objetivo é processar esses valores para produzir uma sequência de bits pseudoaleatória de tamanho fixo.

Expansão: Na segunda etapa, a sequência de bits produzida na fase de extração é expandida para o tamanho final desejado l. Isso garante que a chave gerada seja adequada para o uso pretendido.

Essas duas etapas juntas garantem que a chave derivada seja segura e adequada para criptografia, seja para substituir uma senha insegura por uma chave robusta ou para gerar subchaves seguras a partir de uma chave inicial.

Uma implementação popular do modelo de função de derivação de chaves (FDC) é o *FDC baseada no HMAC* (HKDF) [Kra10]. Vamos ver como ele funciona:

Passo 1: Extração

Na fase de extração, o HKDF usa o HMAC para processar o material de entrada (key material δ) junto com um valor opcional chamado *salt* (r). O resultado dessa operação é uma chave pseudoaleatória (CPA):

$$\text{CPA} := \text{HMAC}(r, \delta)$$

Passo 2: Expansão

Na fase de expansão, o HKDF usa a CPA gerada na fase de extração para produzir a chave final de tamanho l desejado. Isso é feito em várias etapas, concatenando blocos de saída $K(1), K(2), \ldots$ até que o tamanho total l seja atingido:

$$\text{HKDF}(r, \delta, c, l) := K(1)||K(2)\ldots||K(t)$$

Aqui está como cada bloco $K(i)$ é gerado:

- *Primeiro Bloco*:

$$K(1) := \text{HMAC}(\text{PRK}, c||0)$$

  Onde c é um valor contextual opcional e 0 é um byte adicional para garantir que o primeiro bloco seja único.

- *Blocos Subsequentes*:

$$K(i+1) := \text{HMAC}(\text{PRK}, K(i)||c||i)$$

  Onde $K(i)$ é o bloco anterior, c é o valor contextual, e i é um contador que garante que cada bloco gerado seja diferente.

A concatenação de todos os blocos $K(1), K(2), \ldots$ forma a chave final gerada pelo HKDF.

O HKDF garante que a chave derivada seja segura, mesmo quando o *material da chave* original não é ideal, e é altamente flexível, permitindo a derivação de chaves de qualquer tamanho desejado. Além disso, o uso do HMAC como base garante que o HKDF herde as propriedades de segurança bem estabelecidas dessa construção.

Quando o *material da chave* δ é uma senha fornecida por um usuário, enfrentamos um desafio adicional. Normalmente, as senhas têm uma *entropia baixa*, ou seja, elas não são suficientemente aleatórias. Isso as torna vulneráveis a ataques conhecidos como *ataques de dicionário*, onde um adversário tenta adivinhar a senha testando várias combinações comuns.

Para mitigar esse problema, podemos usar uma *função de derivação de chaves lenta*. Isso significa que a função é projetada para ser intencionalmente demorada, dificultando o trabalho de um adversário que tenta executar um grande número de tentativas para adivinhar a senha.

Um tipo popular de função de derivação de chaves baseada em senha é o *Password-Based Key Derivation Function 2* (PBKDF2).

O PBKDF2 recebe como parâmetros:

Salt r: Um valor aleatório que é adicionado à senha para garantir que senhas iguais resultem em chaves diferentes.

Tamanho l: O tamanho da chave final que queremos gerar.

Senha δ: A senha fornecida pelo usuário.

Número de ciclos n: Um valor que especifica quantas vezes a função de hash será aplicada, tornando o processo mais lento e seguro.

O PBKDF2 gera a chave final em várias etapas:

$$\text{PBKDF2}(r, \delta, c, n, l) := K(1)||K(2)||\ldots||K(t)$$

Aqui está como cada bloco $K(i)$ é calculado:

Cálculo do Bloco $K(i)$:

$$K(i) := U_0^i \oplus U_1^i \oplus \cdots \oplus U_n^i$$

Onde cada U_j^i é uma sequência de bits gerada em várias iterações.

Primeira Iteração:

$$U_0^i := H(\delta||r||i)$$

Nesta etapa, a função de hash H é aplicada à combinação da senha δ, do salt r, e do número do bloco i.

Iterações Subsequentes:

$$U_j^i := H(\delta||U_{j-1}^i)$$

Cada iteração subsequente aplica a função de hash H à combinação da senha δ e do resultado da iteração anterior U_{j-1}^i.

Cada bloco $K(i)$ é gerado após n iterações da função de hash, tornando o processo computacionalmente caro e, portanto, mais seguro contra ataques.

O uso do *salt* impede que um adversário prepare um dicionário de senhas e hashes comuns antes do ataque (preprocessamento). Isso significa que, mesmo que duas pessoas usem a mesma senha, elas terão hashes diferentes.

O valor n deve ser suficientemente grande para garantir que a derivação da chave seja lenta, tornando inviável a construção de um dicionário mesmo após a obtenção do hash.

O PBKDF2, ao incorporar essas características, oferece uma proteção robusta contra ataques de dicionário e é amplamente utilizado em sistemas de autenticação e armazenamento de senhas.

8.3 Exercícios

Exercício 34. *Mostre que resistência contra colisões garante a resistência contra preimagem.*

Exercício 35. *Mostre que o seguinte sistema não é seguro contra falsificação considerando que o sistema de hash foi construído a partir do paradigma de Merkle-Damgard:*

- $Gen(1^n) := k \leftarrow \{0,1\}^n$
- $Mac(k,m) := H(k||m)$
- $Ver(k,m,t) := \begin{cases} 1 & se \quad Mac(k,m) = t \\ 0 & c.c. \end{cases}$

Exercício 36. Prova de trabalho *é uma medida para garantir que um determinado usuário tenha que executar uma certa quantidade de processamento durante a excecução de um protocolo. Essa ideia é usada na mineração de bitcoins e para evitar spams. No segundo caso, brevemente, a ideia é exigir uma quantidade mínima de processamento para um cliente que envie um email. Essa quantidade é desprezível para quem manda algumas dezenas de emails por dia, mas é muito cara para quem deseja mandar milhões de spams.*

Uma forma de prova de trabalho é entregar para o cliente a saída de um hash e pedir para que ele compute uma entrada que produza aquela saída. Agumente que, se a função de hash escolhida é segura contra colisão, o melhor que o cliente pode fazer é gerar valores aleatórios de entrada até encontrar um cuja a saída coincida com o resultado esperado.

Exercício 37. *Explique com suas palavras a definição de segurança de hashes. Por que o SHA-1 não é mais considera segura?*

Capítulo 9

Distribuição de Chaves

Até agora, todos os sistemas criptográficos que discutimos são *simétricos*. Isso significa que as partes envolvidas na comunicação compartilham uma mesma informação secreta, que chamamos de *chave*. Já falamos sobre como gerar essas chaves, mas ainda não abordamos uma questão fundamental em criptografia: *como distribuir essas chaves* de forma segura entre as partes.

Nesta seção, vamos explorar dois métodos distintos para resolver o problema da distribuição de chaves. O primeiro método envolve a existência de uma *autoridade central* responsável por distribuir as chaves para as partes interessadas. Essa abordagem é direta, mas depende fortemente da confiança na autoridade central.

O segundo método nos introduz a um novo paradigma chamado *criptografia assimétrica*. Ao contrário da criptografia simétrica, onde a mesma chave é usada para criptografar e descriptografar, na criptografia assimétrica cada parte tem um par de chaves: uma pública (usada para criptografar mensagens) e outra privada (usada para descriptografar mensagens). Esse conceito será explorado mais detalhadamente nos próximos dois capítulos, onde veremos como a criptografia assimétrica resolve o problema de distribuição de chaves de forma elegante e segura.

9.1 Centro de Distribuição de Chaves

No modelo de *Centro de Distribuição de Chaves* (CDC), a segurança das comunicações entre os usuários é centralizada em uma autoridade confiável, chamada de administrador. Para garantir a segurança, cada usuário possui

uma chave permanente única, que é mantida em segredo e gerenciada pelo administrador.

O processo funciona da seguinte forma:

1. *Distribuição de chaves permanentes*: O administrador cria uma chave permanente para cada usuário e a entrega pessoalmente, após verificar a identidade do usuário. Isso garante que cada chave permaneça segura e que apenas o usuário legítimo tenha acesso a ela.

2. *Requisição de comunicação*: Quando Alice deseja se comunicar com Bob, ela primeiro envia uma solicitação ao administrador, informando que deseja estabelecer uma conexão segura com Bob.

3. *Geração e distribuição da chave efêmera*: O administrador, ao receber o pedido de Alice, gera uma chave efêmera k, que será usada apenas durante aquela comunicação específica. O administrador então criptografa essa chave efêmera duas vezes:

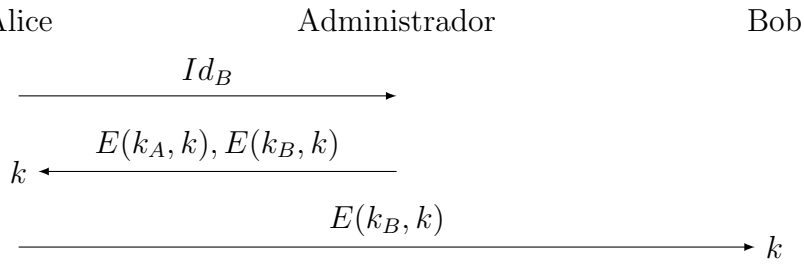
   - Uma cópia da chave é criptografada usando a chave permanente da Alice $E(k_A, k)$.
   - Outra cópia é criptografada usando a chave permanente de Bob $E(k_B, k)$

4. *Envio das chaves*: O administrador envia para Alice ambas as versões da chave efêmera. Alice recebe:

   - $E(k_A, k)$, que ela pode decifrar usando sua chabe permante.
   - $E(k_B, k)$, que ela envia para o Bob.

5. *Transmissão para o destinatário*: Alice então encaminha $E(k_B, k)$ para Bob que pode decifrá-la com sua chave permanente.

Agora, tanto Alice quanto Bob têm a chave k, que podem usar para se comunicar de forma segura durante aquela sessão. Este processo garante que a chave efêmera é conhecida apenas por Alice e Bob, além do administrador que a gerou, mantendo a comunicação confidencial.



Depois que a comunicação entre Alice e Bob é concluída, ambos devem *descartar a chave efêmera* que foi utilizada. Isso é importante porque essa chave é válida apenas para aquela sessão específica de comunicação. Se Alice precisar se comunicar novamente com Bob no futuro, ela deverá solicitar ao administrador uma nova chave efêmera.

Uma implementação bastante conhecida desse modelo de Centro de Distribuição de Chaves é o *protocolo Kerberos*. O Kerberos é amplamente utilizado em ambientes corporativos para gerenciar a autenticação e a segurança de comunicações, seguindo os princípios do CDC. No processo de autenticação, o servidor recebe as credenciais do usuário e, após verificá-las, emite um Ticket de Concessão de Tickets (TGT). O TGT, criptografado com a chave do CDC, contém informações como o ID do usuário, endereço de rede e uma chave de sessão. O usuário então utiliza esse TGT para solicitar um ticket de serviço ao CDC, enviando junto um authenticator, que inclui o ID do cliente e uma marca temporal para prevenir ataques de repetição. Após verificar essas informações, o CDC emite o ticket de serviço, que é criptografado com a chave secreta do servidor e inclui uma nova chave efêmera.

O modelo CDC, no entanto, tem limitações importantes. Ele pressupõe a existência de um administrador confiável que gerencia todas as chaves permanentes dos usuários. Além disso, esse modelo possui um problema significativo conhecido como ponto único de falha. Isso significa que se o servidor do administrador falhar, todas as comunicações seguras entre os usuários ficarão interrompidas.

Portanto, embora o modelo CDC possa funcionar bem em ambientes fechados com uma hierarquia bem definida, ele não é ideal. Em ambientes abertos e distribuídos, onde não há uma autoridade central confiável ou onde a disponibilidade contínua do servidor não pode ser garantida, esse modelo é inadequado e pode levar a falhas significativas na segurança e na comunicação.

9.2 Protocolo de Diffie-Hellman

Em um ambiente aberto, onde as partes não têm a possibilidade de encontrar um administrador previamente, confiar em uma autoridade central para a distribuição de chaves não é realista. No entanto, existe uma solução surpreendente: é possível que as partes troquem chaves diretamente entre si, mesmo utilizando um canal de comunicação inseguro.

O método que apresentaremos nesta seção é conhecido como *protocolo de Diffie-Hellman*, que marca o início do paradigma revolucionário da *criptografia assimétrica* [DH76]. Nos próximos capítulos, exploraremos esse conceito em mais detalhes.

Informalmente, o protocolo de Diffie-Hellman funciona da seguinte maneira: Alice e Bob trocam entre si alguns dados públicos. A partir desses dados, ambos conseguem calcular um valor comum, que pode ser usado como entrada em uma Função de Derivação de Chaves para gerar uma chave secreta compartilhada. O interessante é que, para um observador externo (a Eva), que só teve acesso aos dados públicos trocados, é impossível calcular esse valor comum e, consequentemente, a chave secreta.

Para explicar o protocolo precisamos definir o que são grupos, geradores e grupos cíclicos e precisamos apresentar o *problema do logaritmo discreto* cuja dificuldade é condição necessária – apesar de insuficiente – para a segurança do protocolo.

Um *grupo* é uma estrutura matemática que possui duas partes:

- Um *conjunto* que podemos chamar de G.
- Uma *operação* que combina dois elementos desse conjunto e resulta em um terceiro elemento do mesmo conjunto. Essa operação pode ser algo como adição, multiplicação, ou outra forma de combinação.

Para que essa estrutura seja considerada um grupo, a operação deve seguir algumas regras específicas:

(fecho) Se você pegar dois elementos de G e usar a operação entre eles, o resultado também deve ser um elemento de G. Por exemplo, se G for o conjunto dos números inteiros e a operação for a adição, então somar dois números inteiros sempre dará outro número inteiro.

(identidade) Dentro do conjunto G, deve existir um elemento especial que não muda o valor de outros elementos quando usado na operação.

Esse elemento é chamado de *identidade* ou *elemento neutro*. No caso da adição dos números inteiros, esse elemento é o zero, porque somar zero a qualquer número não altera o número. No caso da multiplicação o elemento neutro é o um.

(inverso) Para cada elemento em G, deve existir outro elemento em G que, quando combinado com ele usando a operação, resulta na identidade. Por exemplo, para a adição, o inverso de um número é o seu oposto (como 3 e -3), porque somá-los dá zero.

(associatividade) A forma como os elementos são agrupados na operação não deve alterar o resultado. Isso significa que, ao combinar três ou mais elementos, não importa se você combina os dois primeiros ou os dois últimos primeiro – o resultado será o mesmo. Ou seja, $(a \circ b) \circ c = a \circ (b \circ c)$.

Um grupo pode ser finito ou infinito dependendo se o conjunto G tiver um número limitado de elementos ou não. Quando ele é finito, o número total de elementos em G é chamado de *ordem* do grupo.

Além disso, se a operação der o mesmo resultado independentemente da ordem em que você combina os elementos, o grupo é chamado de *abeliano* ou *comutativo*. Ou seja, $a \circ b = b \circ a$.

O conjunto dos número inteiros com a operação de adição é um grupo abeliano. Já o conjunto dos números naturais não é porque nenhum número natural tem inverso aditivo.

O conjunto dos números inteiros com a operação de multiplicação não é um grupo porque, com exceção do 1, nenhum outro elemento tem inverso multiplicativo. Já o conjunto dos número racionais sem o 0 é um grupo.

Vamos apenas apresentar um grupo finito:

Exemplo 16. *Se p um número primo, o conjunto $\mathbb{Z}_p^\star = \{1, \ldots, p-1\}$ que possui p elementos com a operação de multiplicação módulo p é um grupo (Lembre que o módulo o resto da operação por p).*

Não é difícil se convencer de que essa estrutura é fechada, associativa e que o 1 é seu elemento neutro. O difícil é motrar que todos elementos possuem inverso.

Mais para frente vamos tentar provar isso, por hora veja o exemplo do conjunto $\mathbb{Z}_7^\star$. Todos seus elementos tem um inverso multiplicativo:

$1^{-1} \equiv 1 \ mod \ 7$ $\quad$ $4^{-1} \equiv 2 \ mod \ 7$

$2^{-1} \equiv 4 \ mod \ 7$ $\quad$ $5^{-1} \equiv 3 \ mod \ 7$

$3^{-1} \equiv 5 \ mod \ 7$ $\quad$ $6^{-1} \equiv 6 \ mod \ 7$

Para nossos propósitos, será útil definir como funciona a operação de exponenciação dentro de um grupo. Se temos um elemento g em um grupo G, e um número inteiro positivo m, podemos definir a exponenciação de g pelo número m como a operação de combinar g consigo mesmo m vezes (vamos usar o símbolo $\circ$ para representar uma operação genérica):

$$g^m := \underbrace{g \circ \cdots \circ g}_{m}$$

Teorema 16 (Euller)**.** *Seja G um grupo abeliano finito de tamanho m, então para qualquer elemnto g temos que:*

$$g^m = 1$$

Demonstração. Note que se $g \circ g_i = g \circ g_j$ então $g_i = g_j$. Assim temos que $\{(g \circ g_1), \ldots, (g \circ g_m)\}$ tem todos os elementos de G porque todos eles são distintos e há m elementos. Assim temos que:

$$\begin{aligned} g_1 \circ \cdots \circ g_m &= (g \circ g_1) \circ \cdots \circ (g \circ g_m) \\ &= g^m \circ (g_1 \circ \cdots \circ g_m) \end{aligned}$$

Multiplicando ambos os lados pelo inverso de $g_1 \circ \cdots \circ g_m$ temos que $g^m = 1$. □

Se pegarmos um elemento específico g de um grupo e começarmos a combiná-lo repetidamente consigo mesmo, vamos gerar uma sequência de novos elementos. Por exemplo, começamos com o próprio elemento, depois combinamos ele com ele mesmo, depois combinamos o resultado com ele novamente, e assim por diante. O conjunto de todos os elementos que conseguimos gerar dessa forma é o que chamamos de conjunto dos elementos gerados por aquele elemento específico.

Escrevemos assim o conjunto gerado pelo elemento g:

$$\langle g \rangle := \{g^0, g^1, \ldots\}$$

A *ordem de um elemento* é o número de vezes que precisamos combiná-lo consigo mesmo até que ele "volte" ao ponto de partida, ou seja, até que o resultado seja o elemento especial do grupo que não muda nada quando combinado com outros (como o 0 na adição ou 1 na multiplicação).

O teorema anterior nos diz que esse número (a ordem) é sempre menor ou igual ao número total de elementos do grupo. Isso significa que, em um grupo finito, o conjunto dos elementos gerados por qualquer elemento específico também será finito.

Finalmente, um grupo é chamado de *cíclico* se existe um elemento especial dentro dele, chamado de *gerador*, que, quando combinado repetidamente consigo mesmo, é capaz de gerar todos os elementos do grupo. Em outras palavras, o conjunto dos elementos gerados por esse gerador é exatamente o grupo inteiro ($\langle g \rangle = G$).

Em um grupo cíclico com gerador g, qualquer elemento h dentro desse grupo pode ser obtido ao combinar g consigo mesmo um certo número de vezes x.

Esse número x é chamado de logaritmo de h na base g. Ou seja, $log_g(h)$ é o número de vezes precisamos combinar o gerador g consigo mesmo para chegar ao elemento h no grupo.

Exemplo 17. *Considere o grupo $\mathbb{Z}_7^\star$ com a operação de multiplicação:*

> $\langle 2 \rangle = \{1, 2, 4\}$. *A ordem de* 2 *é* 3 *e, portanto,* 2 *não é um gerador de* $\mathbb{Z}_7^\star$. *Além disso, neste grupo* $log_2(1) = 0$, $log_2(2) = 1$ *e* $log_2(4) = 2$.
>
> $\langle 3 \rangle = \{1, 3, 2, 6, 4, 5\}$. *A ordem de* 3 *é* 6, *portanto* 3 *é um gerador de* $\mathbb{Z}_7^\star$ *e o grupo é cíclico. Além disso, neste grupo* $log_3(1) = 0$, $log_3(2) = 2$ *e* $log_3(3) = 1$, $log_3(4) = 4$, $log_3(5) = 5$ *e* $log_3(6) = 3$.

O *protocolo de Diffie-Hellman* permite que duas pessoas, Alice e Bob, compartilhem uma chave secreta para comunicação, mesmo que estejam usando um canal público. Funciona assim:

1. Alice começa criando um grupo cíclico e escolhendo um gerador g desse grupo. Esse gerador é usado para criar outros elementos no grupo.

2. Alice então escolhe um número secreto, que chamamos de x, e usa esse número para calcular um valor h_A, combinando o gerador consigo mesmo x vezes. Ou seja, $h_A = g^x$.

3. Alice envia para Bob o grupo que ela criou, o gerador, e o valor h_A que ela calculou. (As vezes h_A é chamado de chave pública de Alice e x sua chave privada).

4. Quando Bob recebe essas informações, ele também escolhe um número secreto, y, e usa esse número para calcular um valor h_B, combinando o gerador g consigo mesmo y vezes. Ou seja, $h_b = g^y$.

5. Agora Bob envia h_B para Alice. (As vezes h_B é chamado de chave pública de bob e y sua chave privada).

6. Agora, Alice e Bob podem calcular uma chave secreta que ambos compartilham. Alice usa o valor h_B que recebeu de Bob, e eleva-o a x que ela escolheu, para obter a chave $h_B^x = (g^y)^x = g^{xy}$. Bob faz a mesma coisa, pega h_A, que recebeu de Alice, e elevá-o ao seu número secreto y para obter $h_A^y = (g^x)^y = g^{xy}$

No final do processo, tanto Alice quanto Bob conseguem calcular o valor g^{xy} que pode ser usado para gerar uma chave (usado como material para a chave). Alice consegue fazer isso porque ela conhece seu próprio número secreto x e o valor g^y que recebeu de Bob. Da mesma forma, Bob consegue calcular g^{xy} porque ele conhece seu número secreto y e o valor g^x que recebeu de Alice.

Agora, se alguém estiver espionando a comunicação entre Alice e Bob, essa pessoa (vamos chamá-la de Eva) só terá acesso aos valores g^x e g^y. Para descobrir a chave secreta, Eva precisaria descobrir x a partir de g^x (ou y a partir de g^y, que é equivalente). Isso significa que Eva precisaria resolver o logaritmo discreto — ou seja, encontrar o número x que satisfaz a equação $g^x = h_A$.

Existe uma forma simples de Eva calcular o logaritmo discreto, que é o que chamamos de força bruta. Para isso ela teria que testar todos os valores possíveis de x até encontrar o correto. Esse tipo de ataque é linear no número de possíveis valores de x, mas se x for grande um tipo de ataque como esse é inviável na prática.

Em resumo, o protocolo de Diffie-Hellman é seguro se o problema do logaritmo discreto para o grupo escolhido for difícil.

Você pode estar se perguntando: "Mas Bob também não teria que tentar todas as possibilidades para calcular g^{xy}?" A resposta é *não*, e isso é porque Bob pode usar um método muito mais eficiente para fazer essa exponenciação.

Primeiro, vamos entender o método "ingênuo" de calcular g^x. A ideia básica é que você pode calcular g^x multiplicando g por ele mesmo x vezes, como mostrado abaixo:

$$g^x = g \circ g^{x-1}$$

Nesse método, começamos com $g^0 = 1$, já que qualquer número elevado a zero é 1. Depois, continuamos multiplicando g até chegar a g^m. Esse procedimento pode ser implementado de maneira simples, mas não é muito eficiente, pois leva m passos para completar o cálculo. O tempo necessário para esse cálculo cresce linearmente com m e exponencialmente com o tamanho de m (lembre que podemos escrever o número m em notação binária usando $log_2(m)$ dígitos), o que significa que o método fica muito lento para números grandes.

No entanto, podemos fazer isso de maneira muito mais rápida usando um truque inteligente chamado *exponenciação rápida*. Em vez de calcular g^m multiplicando g repetidamente, podemos dividir o problema em partes menores e resolver cada parte de forma mais eficiente:

$$g^m = \begin{cases} g^{m/2} \circ g^{m/2} & \text{se } m \text{ é par} \\ g^{\lfloor m/2 \rfloor} \circ g^{\lfloor m/2 \rfloor} \circ g & \text{se } m \text{ é ímpar} \end{cases}$$

Em vez de precisar de x passos, agora precisamos de apenas $\log_2(x)$ passos, o que é muito menor. Assim, o tempo necessário para calcular g^m cresce linearmente com o tamanho do número m, o que torna esse método extremamente eficiente, mesmo para números grandes.

A dificuldade em resolver o problema do logaritmo discreto é uma condição necessária, mas não suficiente por si só, para garantir a segurança do protocolo de Diffie-Hellman. Existem certas estruturas matemáticas que são bem adequadas para criar grupos cíclicos onde o problema do logaritmo discreto é considerado difícil de resolver. Duas das mais utilizadas na prática são os *resíduos quadráticos em grupos de tamanho primo* e as *curvas elípticas*.

Se o problema do logaritmo discreto fosse fácil, uma espiã como Eva poderia encontrar os valores secretos x e y a partir de g^x e g^y, descobrindo assim o valor de g^{xy}. No entanto, em um sistema seguro, o melhor que Eva poderia fazer seria tentar todos os possíveis valores de x ou y até encontrar o correto, o que é impraticável se o grupo for grande o suficiente. Portanto, ao escolher grupos de ordem suficientemente grande, essa tarefa se torna computacionalmente inviável.

Uma vez que Alice e Bob possuem o mesmo valor secreto g^{xy}, eles podem usá-lo para alimentar uma Função de Derivação de Chave, gerando assim as chaves necessárias para um sistema de criptografia simétrico.

Na teoria, assumimos que o grupo e o gerador são criados cada vez que Alice e Bob decidem se comunicar. Na prática, porém, é mais comum que esses parâmetros sejam definidos pelo protocolo e reutilizados em todas as comunicações. Isso não afeta a segurança do sistema.

Caso o problema do logaritmo discreto não seja difícil, Eva seria capaz de produzir x e y a partir de g^x e g^y e descobrir o valor de g^{xy}. Se o sistema for seguro, porém, o melhor que Eva pode fazer é tentar todos os valores de $\mathbb{Z}_n$ até encontrar x ou y. Escolhendo grupos de ordem suficientemente grande, essa tarefa é computacionalmente inviável. Uma vez que as partes possuem o mesmo valor secreto, elas podem utilizá-lo para alimentar um KDF e gerar chaves para um sistema de criptografia simétrico.

Embora o protocolo de Diffie-Hellman permita a criação de uma chave secreta sem a necessidade de uma reunião física ou uma autoridade central confiável, ele não garante a autenticidade das partes envolvidas na comunicação. Isso significa que o protocolo é vulnerável a um tipo de ataque conhecido como "Man in the Middle". Voltaremos a isso no Capítlo 11.

9.3 Exercícios

Exercício 38. *Mostre que se $n|a$ e $n|b$ então para quaisquer r e s temos que $n|(a \cdot r + b \cdot s)$.*

Exercício 39. *Considere a estrutura $\langle \mathbb{Z}_{12}, \cdot \rangle$ em que a multiplicação é calculada módulo 12. Mostre que 6 não possui inverso nesta estrutura e, portanto, ela não é um grupo.*

Exercício 40. *Considere as estruturas $\langle \mathbb{Z}_n, + \rangle$ formadas pelo conjunto $\mathbb{Z}_n := \{0, \ldots, n-1\}$ e a operação de soma (+) módulo n. Mostre essa estrutura é um grupo cíclico para qualquer valor de $n \geq 1$ e que o número 1 é sempre um gerador nesses grupos. (Dica: Você precisa mostrar que a operação satisfaz fecho, associatividade, possui elemento neutro e inverso. Depois você deve mostrar que o elemento 1 gera todos os elementos do grupo.)*

Por que o grupo $\langle \mathbb{Z}_n, + \rangle$ não é um bom candidato para ser usado no protocolo de Diffie-Hellmann.

Exercício 41. *Descreva um ataque força-bruta contra o protocolo de Diffie-Hellman. Em termos assintóticos em relação ao tamanho da entrada, qual é o consumo de tempo deste ataque?*

Capítulo 10

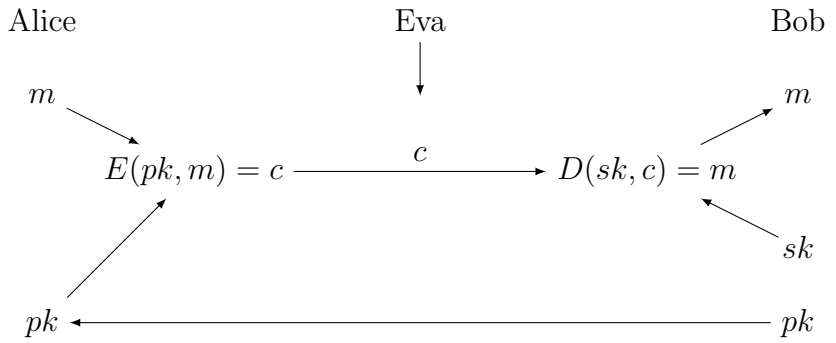
Criptografia Assimétrica

No capítulo anterior, vimos um protocolo que permite a troca de chaves de criptografia sem que as partes precisem se encontrar pessoalmente ou confiar em um terceiro para garantir a segurança. Essa ideia inovadora foi apresentada pela primeira vez em um artigo importante escrito por Diffie e Hellman [DH76].

Nesse mesmo artigo, os autores sugeriram um conceito revolucionário: a *criptografia assimétrica*. Para entender como isso funciona, imagine um cadeado. Qualquer pessoa pode fechar o cadeado, mas só o dono da chave consegue abri-lo. A criptografia assimétrica é semelhante. Cada pessoa tem duas chaves: uma chave pública, que é como o cadeado, e uma chave secreta, que é a única capaz de abri-lo.

Vamos dar um exemplo: se Alice quer enviar uma mensagem segura para Bob, ela primeiro precisa da chave pública de Bob. Essa chave pública pode estar disponível online ou Bob pode enviá-la para Alice por qualquer meio, mesmo que não seja seguro. Com essa chave, Alice pode "trancar" a mensagem, ou seja, criptografá-la, e só Bob, com sua chave secreta, conseguirá "abrir" a mensagem e lê-la.

O processo é ilustrado no diagrama a seguir:



Um *sistema de criptografia assimétrico* é composto por três algoritmos: Gen, E e D.

> Gen é o algoritmo responsável por gerar as chaves. Ele cria um par de chaves: uma chave secreta, que chamamos de sk, e uma chave pública, que chamamos de pk.
>
> E é o algoritmo de criptografia, que usa a chave pública pk para transformar uma mensagem, m, em uma mensagem cifrada.
>
> D é o algoritmo de descriptografia, que usa a chave secreta sk para transformar a mensagem cifrada de volta na mensagem original.

Para garantir que o sistema seja *correto*, é preciso que, ao descriptografar com a chave secreta uma mensagem que foi criptografada com a chave pública correspondente, o resultado seja exatamente a mensagem original:

$$D(sk, E(pk, m)) = m$$

Isso significa que, se você criptografar uma mensagem com a chave pública de alguém e depois descriptografá-la com a chave secreta dessa pessoa, você recupera a mensagem original.

Um sistema de criptografia assimétrico é considerado *seguro* se for inviável para um adversário, que tenha acesso à chave pública, descobrir qualquer informação útil que ajude a distinguir qual entre duas mensagens diferentes foi criptografada. Isso deve ser verdade mesmo que a chave pública esteja disponível para todos.

Ou seja, possuir a chave pública é semelhante a ter acesso a um oráculo, um recurso que ajuda a criptografar mensagens. Em outras palavras, a posse da chave pública dá ao adversário uma capacidade equivalente ao que assumimos em um ataque do tipo ETP.

10.1 El Gammal

O primeiro sistema de criptografia assimétrica foi descoberto por Rivest, Shamir e Adleman, um ano após o trabalho de Diffie e Hellman. Antes de introduzirmos essa solução, vamos apresentar um esquema que possui muitas semelhanças com o protocolo discutido no capítulo anterior. Apesar dessas semelhanças, esse sistema só foi formalizado em 1985 por Taher ElGamal, durante seu doutorado, sob a orientação do próprio Martin Hellman [EG85].

Esse sistema de criptografia segue o modelo de criptografia assimétrica, que utiliza dois tipos de chaves: uma pública e uma privada. Ele é composto por três algoritmos:

> *Gen* cria um grupo cíclico com um gerador g. A chave secreta sk é um número x. A chave pública é $h = g^x$.
>
> E sorteia um número y e multiplica a mensagem m por h^y e junta o valor g^y.
>
> $$E(pk, m) = h^y \cdot m, g^y$$
>
> D pega a primeira parte da cifra $(h^y \cdot m)$ e divide pela segunda parte (g^y) elevado a x.
>
> $$(h^y \cdot m) \cdot (g^{xy})^{-1}$$

Não é difícil verficar que este sistema é correto. Basta lembrar que $h = g^x$, substituir e desenvolver:

$$\begin{aligned}(h^y \cdot m) \cdot (g^{xy})^{-1} &= (g^{xy} \cdot m) \cdot (g^{xy})^{-1} \\ &= m\end{aligned}$$

Assim como no protocolo de Diffie-Hellman, a segurança do sistema de El Gammal depende da dificuldade do problema do logaritmo discreto para o grupo escolhido.

10.2 RSA

Em 1978, Rivest, Shamir e Adleman apresentaram o primeiro sistema de criptografia assimétrica, conforme idealizado por Diffie e Hellman um ano

antes [RSA78]. Diferente do protocolo de Diffie-Hellman e do sistema de El Gamal, que se baseiam na dificuldade do problema do logaritmo discreto, o sistema RSA depende da dificuldade de outro problema matemático: o problema da fatoração.

Antes de explicar como o sistema funciona, precisamos fazer uma breve pausa para discutir alguns conceitos matemáticos e mostrar um exemplo de grupo finito.

Vamos considerar o conjunto $\mathbb{Z}_n$ composto pelos números que vão de 0 até $n-1$. As operações de soma e multiplicação dentro deste conjunto serão obtidas somando os números e depois tomando o módulo por n. Isso garante que toda operação permanece dentro de $\mathbb{Z}_n$.

Proposição 1. *Um número a possui inverso multiplicativo em $\mathbb{Z}_n$ se e somente se a e n são primos entre si, ou seja $mdc(a, n) = 1$.*

Demonstração. Se a e n são primos entre si, então é possível mostrar que existem números inteiros x e y tais que $ax + ny = 1$ (isso é conhecido como a identidade de Bezout). Nesse caso, $ax = 1$ no sistema módulo n, ou seja, x é o inverso de a.

Por outro lado, se a e n não são primos entre si, então existe um número $b > 1$ que divide tanto a quanto n. Agora, imagine que a tivesse um inverso, ou seja, um valor x tal que ax é igual a 1 no sistema módulo n. Isso implicaria que poderíamos escrever 1 como $ax + ny$ para algum inteiro y. No entanto, como b divide tanto a quanto n, b também teria que dividir 1. Mas isso é impossível, porque o único divisor de 1 é ele mesmo, e $b > 1$. Concluímos, então, que esse x não pode existir e, portanto, a não tem inverso em $\mathbb{Z}_n$. □

Proposição 2 (Identidade de Bezout)**.** *Para quaisquer dois números inteiros a e b sempre existem outros dois números inteiros x e y tais que $xa + yb = mdc(a, b)$.*

Demonstração. Considere o conjunto formado por todos os números que podem ser escritos na forma $x'a + y'b$ para x' e y' inteiros. Seja $d = xa + yb$ o menor desses números que podem ser escritos dessa forma.

Agora, pegue um número qualquer c que também possa ser escrito como $c = x'a + y'b$. Se dividirmos c por d, teremos um quociente q e um resto r, ou seja, $c = qd + r$.

Podemos então reescrever r como

$$r = c - qd = x'a + y'b - q(xa + yb) = a(x' - qx) + b(y' - qy).$$

Isso mostra que o resto r também pode ser escrito na forma $ax' + by'$.

Agora, o resto r teria que ser menor do que d, mas isso contradiz o fato de que d é o menor número que pode ser escrito na forma $xa + yb$. Concluímos, então, que $r = 0$, ou seja, não há resto.

Como o resto é 0, isso significa que d divide c. Mas c foi escolhido como qualquer número que pode ser escrito como $x'a + y'b$. Em particular, isso vale para a e para b. Portanto, d divide tanto a quanto b.

Agora, resta mostrar que d é o maior dos divisores de a e b. Pegue qualquer divisor d' de a e de b. Nesse caso, d' também dividiria $ax + by = d$. Portanto, d' teria que ser menor ou igual a d.

Concluímos, então, que d é o maior dos divisores de a e b, ou seja, $d = ax + by = \text{mdc}(a, b)$. □

Para encontrar os coeficientes x e y que satisfazem a Identidade de Bezout, ou seja, $xa + yb = \text{mdc}(a, b)$, podemos utilizar uma versão estendida do algoritmo de Euclides. O Algoritmo de Euclides é um dos algoritmos mais antigos e conhecidos da matemática, descrito pelo matemático grego Euclides por volta de 300 a.C. no seu livro "Elementos". Sua principal função é calcular o máximo divisor comum (mdc) entre dois números inteiros a e b. Euclides notou que o mdc de dois números pode ser encontrado repetindo divisões sucessivas, substituindo o número maior pelo menor e o menor pelo resto da divisão, até que o resto seja zero. Quando isso ocorre, o divisor no passo anterior será o mdc. A versão extendida deste algoritmo não apenas calcula o mdc de dois números a e b, mas também determina os valores de x e y tais que $xa + yb = mdc(a, b)$.

```
AEE(a, b)
1  ▷ Recebe a, b inteiros com a > b
2  ▷ Devolve t, x e y tais que t = mdc(a, b) = xa + yb
3  if b|a
4     then return b, 0, 1
5          ▷ a = bq + r
6     else t, x, y ← AEE(b, r)
7          return t, y, x − y · q
```

Vamos definir um conjunto $\mathbb{Z}_n^\star$ composto por todos os números inteiros positivos menores do que n que possuem inverso multiplicativos:

$$\mathbb{Z}_n^\star := \{a \in \mathbb{Z}_n : mdc(a, n) = 1\}$$

Para todo a em $\mathbb{Z}_n^\star$ temos que a e n são primos entre si. Então podemos usar Algoritmo Extendido de Euclides para calcular x e y tais que $ax+ny = 1$. O valor x assim calculado é o inverso de a em $\mathbb{Z}_n$.

Todos os elementos de $\mathbb{Z}_n^\star$ possuem inverso multiplicativo. Não é difícil mostrar que as outras propriedades dos grupos também são satisfeitas nesse conjunto quando aplicamos a multiplicação módulo n (Exercício 42). Chamamos de *função de Euler* a função que nos dá o tamanho de $\mathbb{Z}_n^\star$ para cada n:

$$\phi(n) := |\mathbb{Z}_n^\star|$$

Quando n é o produto de dois número primos, calcular a função de Euler é bastante simples:

Proposição 3. *Sejam p e q números primos:*

$$\phi(pq) = (p-1)(q-1)$$

Demonstração. Para entender por que essa fórmula funciona, vamos considerar o seguinte:

Se pegarmos um número a dentro do conjunto $\mathbb{Z}_n$, e a não tem p ou q como divisor, então a será "primo" em relação a n. Mas, se a for divisível por p ou q, então a não será "primo" em relação a n.

Os números em $\mathbb{Z}_n$ que são divisíveis por p são: $p, 2p, \ldots, (q-1) \cdot p$. Os números que são divisíveis por q são: $q, 2q, \ldots, (p-1) \cdot q$.

Para encontrar $\phi(n)$, subtraímos do total dos elementos de $\mathbb{Z}_n$ o 0, e os números divisíveis por p e os divisíveis por q. O resultado nos dá a quantidade de números em $\mathbb{Z}_n$ que não são divisíveis por p ou q, que é exatamente $(p-1)(q-1)$.

Portanto, a fórmula $\phi(pq) = (p-1)(q-1)$ nos dá o número de elementos em $\mathbb{Z}_n$ que possuem um inverso multiplicativo, ou seja, que são "primos" em relação a n. □

A geração de chaves *Gen* no sistema RSA segue os seguintes passos:

1. Escolhemos aleatoriamente dois números primos, p e q, ambos com o número de bits predeterminado e suficientemente grande.

2. Em seguida, calculamos N, que é o produto dos dois primos $p \times q$.

3. Calculamos $\phi(N) = (p-1)(q-1)$.

4. Agora, escolhemos um número e que seja maior que 1 e que não tenha divisores comuns com $\phi(N)$, exceto o número 1. Isso garante que e e $\phi(N)$ sejam "primos entre si".

5. Depois, calculamos d, que é o inverso de e em relação a $\phi(N)$ usando o AEE.

6. N é a chave pública e d é a chave privada.

Na prática, podemos escolher e previamente e calcular d com base nisso. Para tornar a criptografia mais eficiente, é útil escolher um valor para e que tenha poucos bits `1`, o que facilita o processo de cálculo durante a criptografia. Dois valores comumente usados para e são 3 e $2^{16} + 1$.

Como vimos, o cálculo de d é feito usando o Algoritmo Estendido de Euclides.

Em sua versão crua, o sistema de RSA é definido pela seguinte tripla de algoritmos:

> *Gen* gera um par de chaves, pública (e, N) e privada (d).
>
> E criptografa uma mensagem m elevando-a a e e calculando o móulo por N.
>
> $$m^e \ mod \ N$$
>
> D descriptografa a cifra c elevando-a a d e calculando o módulo por N.
>
> $$c^d \ mod \ N$$

Como $\mathbb{Z}_n^\star$ com a operação de multiplicação forma um grupo, segue do Teorema 16 que para qualquer a temos que:

$$a^{\phi(n)} \equiv 1 \ (mod \ n)$$

Esse resultado é chamado de *Teorema de Euler*.

Corolário 1. *Para todo $a \in \mathbb{Z}_n^\star$ temos que $a^x \equiv a^{[x \ mod \ \phi(n)]} \ (mod \ n)$.*

Demonstração. Seja $x = q \cdot \phi(n) + r$ em que $r = [x \ mod \ \phi(n)]$:

$$a^x \equiv a^{q \cdot \phi(n) + r} \equiv a^{q \cdot \phi(n)} \cdot a^r \equiv (a^{\phi(n)})^q \cdot a^r \equiv 1^q \cdot a^r \equiv a^r \ (mod \ n)$$

□

Corolário 2. *Seja $x \in \mathbb{Z}_n^\star$, $e > 0$ tal que $mdc(e, \phi(n)) = 1$ e $d \equiv e^{-1} \ (mod \ \phi(n))$ então $(x^e)^d \equiv x \ (mod \ n)$*

Demonstração.

$$x^{ed} \equiv x^{[ed \ mod \ \phi(n)]} \equiv x \ (mod \ n)$$

□

Esses são os resultados necessários para mostrar a corretude do sistema RSA:

$$D(sk, E(pk, m)) = [(m^e)^d \ mod \ N] = [m^{ed} \ mod \ N] = [m \ mod \ N] = m$$

Uma condição necessária para a segurança dos sistemas RSA é a dificuldade do *problema da fatoração*.

No problema da fatoração o adversário recebe $N = pq$ em que p e q são primos e deve achar os valores p e q.

Esse problema é considerado difícil porque não conhecemos algoritmo eficiente que o resolva com probabilidade considerável.

Caso um adversário eficiente fosse capaz de fatorar N, ele produziria p e q e seria então capaz de gerar $\phi(N) = (p-1)(q-1)$ sem nenhuma dificuldade. Com isso, o sistema se torna completamente inseguro.

Ou seja, a dificuldade do problema da fatoração é necessária para garantir a segurança do sistema. Não sabemos, porém, se essa condição é suficiente.

A construção apresentada acima é chamada de RSA simples (*plain RSA*) e é insegura por uma série de motivos. Um mecanismo para torná-la mais segura é sortear uma sequência de bits r aleatoriamente e usar $m' = r||m$ como mensagem. O mecanismo segue de maneira idêntica, a única diferença é que, ao decifrar, se recupera $r||m$ e então é preciso descartar os primeiros bits.

Essa versão é chamada *padded RSA* e uma adaptação dela é usada no padrão `RSA PKCS #1 v1.5`, que é usada na prática.

10.3 Sistemas Híbridos

Os sistemas de criptografia assimétricos que vimos até agora são, pelo menos, uma ordem de grandeza menos eficientes do que os sistemas de criptografia simétrica que estudamos anteriormente. Além disso, é difícil garantir a segurança desses sistemas para mensagens com qualquer distribuição de probabilidades.

Por esses motivos, o modelo mais popular de criptografia assimétrica segue uma abordagem híbrida, dividida em duas fases: uma fase de encapsulamento de chave e, em seguida, uma fase de encapsulamento dos dados.

Um *mencanismo de encapsulamento de chave* (MEC) é um sistema formado por três algoritmos:

> *Gen* gera um par de chaves pk é uma chave pública e sk uma chave secreta
>
> *Encaps* utiliza chave pública pk para produzir uma cifra c e uma chave k
>
> *Decaps* utiliza chave secreta sk e a cifra c para recupera a chave k.

Um sistema MEC é correto se o processo de decapsulamento sempre recuperar a mesma chave que foi criada na fase de encapsulamento.

Um sistema MEC é *seguro contra ETP* se um adversário polinomial não é capaz de distinguir com probabilidade considerável a chave produzida por *Encaps* de uma chave de mesmo tamanho escolhida aleatoriamente.

Se $\Pi_K = \langle Gen_K, Encaps, Decaps\rangle$ é um MEC seguro contra ETP, e $\Pi' = \langle Gen', E', D'\rangle$ é um sistema de criptografia simétrica seguro contra ataques "ciphertext only", então o seguinte sistema, chamado de *híbrido*, é seguro contra ETP:

> *Gen* gera um par de chaves sk e pk
>
> E utiliza pk para gerar duas cifras c_k e c da seguinte forma:
>
> - *Encaps* utiliza pk para gerar a cifra c_k e a chave k
> - E' utiliza k para criptografar a mensagem m e gerar a cifra c
>
> D utliza sk para recuperar a mensagem da seguinte forma:

- *Decaps* utiliza a chave secreta sk para descriptografar c_k e recuperar a chave k
- D' utiliza a chave k para descriptografar a cifra c e recuperar a mensagem m

Uma forma natural de produzir um MEC é produzir uma chave k e usar um sistema de criptografia assimétrica para criptografar a chave. Esse esquema é seguro desde que o sistema de ciptografia assimestrica seja seguro, mas existem sistemas mais eficientes em determinados casos. Não trataremos desses sistemas neste livro, porém.

10.4 Exercícios

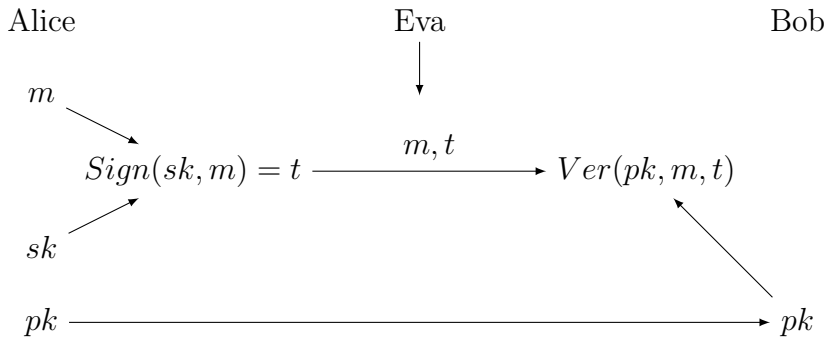
Exercício 42. *Mostre que para qualquer $n \in \mathbb{Z}$ a estrutra $\langle \mathbb{Z}_n^\star, \cdot \rangle$ é um grupo.*

Capítulo 11

Assinaturas Digitais

No capítulo anterior, vimos que é possível construir um sistema de criptografia onde as partes não precisam trocar um segredo antecipadamente. Os esquemas de criptografia assimétrica dependem da geração de um par de chaves: uma pública e outra secreta.

As assinaturas digitais, que veremos neste capítulo, são o equivalente àos códigos de autenticação de mensagem que vimos no Capítulo 6, mas no contexto da criptografia assimétrica.



Um *sistema de assinatura digital* consiste de três algoritmos:

- *Gen* gera um par de chaves sendo um pública pk e outra secreta sk.
- *Sign* recebe a chave secreta sk e uma mensagem m e produz uma assinatura t.

- Ver recebe a chave pública pk, uma mensagem m e uma assinatura t e verifica se essa assinatura é válida para mensagem m e de fato da pessoa que possui a chave secreta.

A ideia é garantir que qualquer pessoa possa verificar se uma determinada mensagem foi assinada pela pessoa que detém a chave secreta correspondente à chave pública. A assinatura digital serve como uma espécie de "carimbo" que autentica a mensagem, assegurando que ela não foi alterada (integridade) e que realmente veio de quem a assinou (autenticidade).

Em uma assinatura em papel, a mesma pessoa produz sempre o mesmo garrancho. Já no meio digital, cada mensagem diferente exige uma assinatura diferente. Isso é essencial porque no meio digital é trivial copiar um arquivo.

Um sistema de assinatura digital é *correto* se, dado um par de chaves pk, sk, uma assinatura t para uma mensagem produzida pela chave sk pode ser verificada por pk.

Um sistema de assinatura é *seguro contra falsificação* se qualquer adversário eficiente, mesmo com acesso a um oráculo que lhe entregue assinaturas para mensagens diferentes de m, não for capaz de produzir uma assinatura válida para uma nova mensagem m com probabilidade considerável.

Da mesma forma que os sistemas de criptografia assimétrica, produzir uma assinatura digital para uma mensagem muito grande pode ser um processo lento. Uma técnica usada na prática para mitigar este problema é assinar não a mensagem m em si, mas um hash da mensagem $H(m)$. Neste caso, para verificar a integridade e autenticidade da mensagem, basta gerar $H(m)$ e verificar a assinatura. Essa construção é chamada de *paradigma Hash-and-Sign*. É possível provar a segurança contra falsificação em uma construção como essa no caso em que o esquema de assinatura usado é seguro contra falsificação e o hash é resistente à colisão.

11.1 Assinatura RSA

Um sistema de assinatura digital pode ser construído invertendo o esquema de criptografia assimétrica que vimos no capítulo anterior. Ou seja, criptografamos a mensagem com a chave secreta para gerar a assinatura e descriptografamos com a chave pública para verificar. Esse esquema não é seguro até aplicarmos o paradigma *hash-and-sign*. O esquema todo é formado por três algoritmos:

- *Gen*: gera um par de chaves $sk = d$ e $pk = e, N$ como vimos no capítulo anterior.

- *Sign*: calcula o hash de m e eleva esse valor por d módulo N ($t = H(m)^d \ mod \ N$).

- Ver: eleva t a e e verifica se o resultado é equivalente a $H(m) \ mod \ N$

A correção deste sistema segue os mesmos passos da correção do sistema de criptografia RSA que vimos no capítulo anterior. Além disso, podemos provar que se o sistema RSA é seguro e se o hash H usado é resistente à colisão, então o sistema acima é *seguro contra falsificação*. Esse sistema é a base do sistema de assinatura do protocolo `RSA PKCS #1 v2.1`.

11.2 Esquemas de Indentificação e DSA

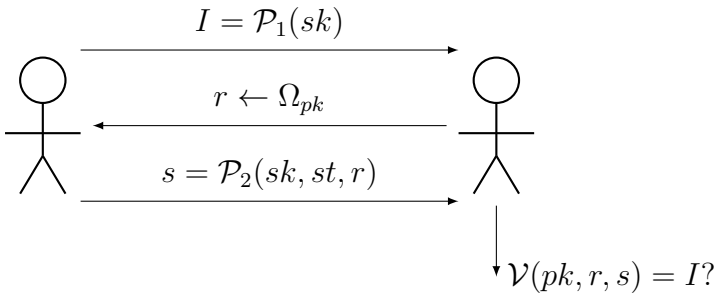
Um *esquema de identificação* é um protocolo de comunicação entre duas partes: o *provador*, que deseja provar sua identidade, e o *verificador*, que deseja confirmá-la. Esse processo é interativo e envolve uma sequência passos:

1. O provador usa um algoritmo $\mathcal{P}_1$ com sua chave secreta sk para gerar uma *mensagem inicial* I e um estado st e envia I para o verificador.

2. O verificador escolhe aleatoriamente um *desafio* r de um conjunto gerado a partida da chave pública do provador e envia r de volta para o provador.

3. O provador usa outro algoritmo $\mathcal{P}_2$ com sua chave secreta sk, o estado st e o desafio r para gerar uma resposta s que ele envia para o verificador.

4. Por fim, o verificador testa se uma função de verificação $\mathcal{V}$ com as entradas da chave públcia pk, o desafio r e o estado s calcula corretamente a mensagem inicial I.



A segurança desse esquema depende de o provador ser o único que possui a chave secreta, garantindo que apenas ele possa responder corretamente ao desafio.

A partir de esquemas de identificação seguros, podemos gerar assinaturas digitais usando a *transformação de Fiat-Shamir*. Nessa transformação, um esquema de identificação é modificado para eliminar a interatividade, tornando-o adequado para assinaturas digitais. Em vez de o verificador enviar o desafio, esse desafio é gerado por meio de uma função de hash aplicada à mensagem e à informação do provador. Essa técnica permite transformar um protocolo interativo em uma assinatura digital não interativa, que pode ser verificada por qualquer pessoa a partir da chave pública do provador.

O DSA (Digital Signature Algorithm) é um dos algoritmos mais usados para criar assinaturas digitais e baseia sua segurança no problema matemático do logaritmo discreto. Assim como no esquema de identificação, o remetente, usando sua chave privada, gera uma assinatura baseada na mensagem, e o receptor, com a chave pública do remetente, verifica se a assinatura é válida. O DSA também utiliza funções de hash para garantir que a assinatura seja única para cada mensagem, aumentando a segurança.

Uma variação do DSA, chamada ECDSA (Elliptic Curve Digital Signature Algorithm), segue o mesmo princípio, mas em vez de usar grupos cíclicos clássicos, utiliza curvas elípticas. A principal vantagem do ECDSA é que ele oferece a mesma segurança do DSA com chaves de tamanho muito menor, o que torna o processo de assinatura e verificação mais eficiente.

11.3 Infraestrutura de Chaves Públicas

O protocolo de Diffie-Hellman e a criptografia assimétrica garantem a segurança da comunicação contra ataques passivos sem precisarmos supor que

as partes compartilhem um segredo *a priori*. Um problema que ainda não tratamos até aqui é como garantir a segurança contra um ataque ativo, ou seja, contra um modelo de ameaça em que o adversário não só é capaz de observar a comunicação, como também é capaz de interferir nela.

Um ataque ao qual os sistemas que vimos até aqui estão particularmente sujeitos é o seguinte: Digamos que Alice pretende se comunicar com Bob usando um sistema de criptografia assimétrico. O primeiro passo para Alice é obter a chave pública de Bob. Neste momento, Eva pode enviar a sua chave pública como se fosse a de Bob, receber as mensagens que Alice enviaria para Bob, copiá-las e reencaminhá-las para Bob. Este tipo de ataque é chamado de *Man In The Middle* e nada do que vimos até aqui o previne.

Não há como evitar este tipo de ataque sem algum encontro físico em que algum segredo seja compartilhado de maneira segura, mas os sistemas de assinatura digitais podem facilitar esse processo.

A ideia é confiar a alguma autoridade a identificação das chaves públicas. A autoridade A teria então a responsabilidade de verificar se a entidade portadora da identidade Id_B é de fato a pessoa que possui a chave secreta correspondente à chave pública pk_B.

Neste caso, a autoridade A pode emitir um *certificado* $cert_{A\to B}$, que nada mais é do que um arquivo assinado por A atestando que Id_B é o titular da chave pk_B:

$$cert_{A\to B} := Sign(sk_A, Id_B||pk_B)$$

Existem diversos modelos de *infraestrutura de chaves públicas* (PKI):

- *Autoridade Certificadora Única:*
  - Neste cenário, assume-se que todos possuem a chave pública da autoridade certificadora A, que foi obtida de maneira segura.
  - Sempre que algum novo ator B precisar publicar sua chave pública, ele deve se apresentar a essa autoridade e comprovar sua identidade para obter o certificado $cert_{A\to B}$.
- *Múltiplas Autoridades Certificadoras:*
  - Neste cenário, existem várias autoridades certificadoras, e assume-se que todos possuem chaves públicas de algumas ou todas elas.

  - Quando um novo ator precisar publicar sua chave, ele pode procurar uma ou mais autoridades para gerar o certificado, que será válido para aqueles que confiam na autoridade certificadora que o emitiu.
  - Neste cenário, a comunicação é tão segura quanto a menos confiável das autoridades incluídas na lista das partes.

- *Delegação de Autoridade:*
  - Neste cenário, as autoridades certificadoras não apenas podem certificar a autenticidade de uma chave pública, mas também podem produzir um certificado especial $cert^*$ que dá o poder a outro de produzir certificados.
  - Por exemplo, se C conseguir um certificado de B sobre a autenticidade de sua chave $cert_{B\to C}$, e B possuir um certificado especial de A que o autorize a produzir certificados em seu nome $cert^*_{A\to B}$, alguém que possui a chave pública de A pode verificar o certificado $cert^*_{A\to B}$ e usar a chave pública assinada de B para verificar $cert_{B\to C}$.

- *Rede de Confiança:*
  - Neste cenário, todas as partes atuam como autoridades certificadoras e podem assinar certificados para qualquer chave que verificaram.
  - Cabe a cada um avaliar a confiança que tem em seus colegas quanto à idoneidade em produzir certificados.

Um certificado pode e deve conter uma *data de expiração* que limite seu tempo de validade. Depois dessa data, o certificado não deve ser mais considerado válido e precisa ser renovado.

Em alguns modelos, o portador pode comprovar sua identidade com a autoridade que emitiu o certificado, e esta pode assinar e publicar um *certificado de revogação* anunciando que a chave não deve mais ser considerada válida — por exemplo, no caso de ela ser perdida ou roubada.

11.4 Protocolos

Para fechar este capítulo, apresentaremos brevemente uma série de protocolos que utilizam as primitivas criptográficas que vimos até aqui.

Um *protocolo*, propriamente dito, deveria descrever os passos de comunicação em um nível de detalhes suficiente para ser implementado sem ambiguidades. Este não é nosso propósito.

Pretendemos aqui apenas apresentar aplicações práticas de uso de criptografia forte e como as primitivas são usadas para resolver problemas de comunicação digital. Assim, esta seção serve mais como um resumo e uma aplicação prática do que estudamos até aqui.

11.4.1 Transport Layer Security (TLS/SSL)

O *Transport Layer Security* (TLS) é uma evolução do protocolo *Secure Socket Layer* (SSL), amplamente utilizado para garantir a segurança nas conexões web, especialmente em sites com o prefixo `https`. Introduzido em 1999, o protocolo TLS é dividido em duas fases principais: a fase de *handshake*, responsável por estabelecer as chaves de criptografia simétrica entre o cliente e o servidor, e a fase *record-layer*, que assegura a confidencialidade, integridade e autenticidade da comunicação. Durante o handshake, o servidor apresenta um certificado digital, que foi emitido por uma autoridade certificadora utilizando o modelo de delegação. Esse certificado vincula a identidade do servidor à sua chave pública. O cliente verifica o certificado usando as chaves públicas de autoridades certificadoras que considera confiáveis e, se for válido, utiliza a chave pública do servidor para encapsular uma chave secreta que será usada na comunicação. Ao final desse processo, cliente e servidor compartilham chaves simétricas autênticas, que são então usadas para criptografar e autenticar as mensagens trocadas.

11.4.2 Secure Shell (SSH)

O *Secure Shell* (SSH) é um protocolo utilizado para comunicação segura, especialmente para login e transferência de arquivos em servidores remotos. Antes da fase de troca de chaves, as partes passam por uma fase de negociação de algoritmos, onde definem o grupo criptográfico, a função de hash e os algoritmos de criptografia a serem usados. Durante a troca de chaves,

o cliente e o servidor geram e compartilham chaves temporárias, que são autenticadas pelo servidor através de uma assinatura digital. O SSH permite o uso de certificados para verificar a autenticidade da chave pública do servidor, mas tipicamente adota o modelo de segurança conhecido como *Trust On First Use* (TOFU). Nesse modelo, a primeira vez que o cliente se conecta ao servidor, a identidade do servidor é salva no cliente, e futuras conexões utilizam essa informação para verificar a autenticidade da chave. Além disso, o SSH apresenta um "fingerprint" da chave pública, que nada mais é do que um hash da chave, permitindo a verificação manual de forma segura. Após a troca de chaves, o servidor e o cliente compartilham as chaves de criptografia e autenticação necessárias para a comunicação segura, além dos vetores iniciais para a encriptação. Antes de iniciar a troca de mensagens, o servidor autentica o cliente, o que pode ser feito por meio de chave pública, senha ou autenticação baseada no host.

11.4.3 Pretty Good Privacy (PGP)

O PGP (*Pretty Good Privacy*) foi desenvolvido no início dos anos 1990 por Phil Zimmermann, inicialmente como uma ferramenta para proteger a comunicação dos ativistas contra o uso de armas nucleares. O PGP combina criptografia híbrida, compactação de arquivos, assinaturas digitais e o modelo de *rede de confiança* para a certificação de chaves públicas, contrastando com o modelo de delegação de autoridade utilizado pelo protocolo TLS/SSL. Em vez de confiar em uma autoridade central para a validação das chaves, o PGP permite que os usuários verifiquem manualmente a autenticidade das chaves públicas e publiquem certificados em servidores de chaves. Originalmente, as *cryptoparties* eram eventos onde as pessoas se reuniam para verificar esses certificados, fortalecendo a rede de confiança do PGP

O protocolo permite que os usuários criptografem mensagens, assinem digitalmente, ou realizem ambas as operações, utilizando algoritmos como RSA, DSA ou ElGamal. Para criptografar uma mensagem, o PGP gera uma chave efêmera que é criptografada de forma assimétrica usando a chave pública do destinatário. Esta chave efêmera é então utilizada para criptografar a mensagem compactada utilizando criptografia simétrica no modo *Cipher Feedback* (CFB) que é menos comum do que os modeos ECB e Ctr apresentados no livro.

11.4.4 Off The Record (OTR)

Enquanto o PGP foi projetado para comunicação assíncrona, como e-mails, o protocolo Off-the-Record (OTR) foi desenvolvido para conversas privadas em comunicação síncrona, como em chats [BGB04]. O PGP simula uma carta assinada e carimbada, garantindo autenticidade e integridade, enquanto o OTR simula uma conversa ao pé do ouvido, oferecendo, além da confidencialidade, negação plausível – o destinatário não pode provar quem enviou a mensagem.

O OTR também oferece sigilo perfeito para o futuro, impedindo que mensagens antigas sejam decifradas mesmo que as chaves sejam comprometidas. Ele utiliza o algoritmo AES em modo contador (Ctr), um modo maleável, e adota um processo de revezamento e esquecimento de chaves para garantir esse sigilo. A negação plausível é reforçada pela exposição das chaves de autenticação após o uso, tornando impossível provar a origem das mensagens após sua leitura.

Para mitigar ataques de Man-in-the-Middle, o protocolo permite a verificação manual de fingerprints ou o uso de métodos mais avançados, como o protocolo do "milionário socialista", que depende da comunicação síncrona.

11.4.5 Signal

O *protocolo Signal*, inicialmente chamado de Axolotl, foi desenvolvido para solucionar os desafios de segurança em mensageiros modernos, que operam de maneira assíncrona, ao contrário do OTR, que foi projetado para comunicação síncrona. Assim como o OTR, o Signal opera em duas fases: handshake e troca de mensagens. No entanto, o Signal adota uma abordagem mais robusta e elegante, dispensando o uso de assinaturas digitais durante o handshake e fortalecendo a negação plausível, evitando ter que expor as chaves de autenticação após a verificação.

Durante o handshake, as partes geram uma *chave raiz* derivada de suas chaves permanentes e efêmeras, que é usada para gerar uma *chave de cadeia* e uma *chave mestra* para criptografia e autenticação. O protocolo introduz o conceito de *ratchet*, que permite o revezamento e esquecimento de chaves de forma mais frequente. Como no protocolo OTR, a chave mestra é trocada sempre que um usuário responde a uma mensagem, mas, além disso, a chave da cadeia é trocada por uma subchave a cada nova mensagem independente se for uma resposta ou uma nova mensagem do mesmo usuário.

O Signal utiliza o modelo de certificação *Trust On First Use (TOFU)*, semelhante ao SSH, e oferece a opção de verificação manual de fingerprints. Originalmente implementado no aplicativo de código aberto Textsecure, o protocolo Signal foi incorporado ao WhatsApp em 2016. O Textsecure passou a ser chamado de Signal e, além de manter seu código aberto, possui uma política de privacidade que minimiza o armazenamento de metadados dos usuários.

11.5 Exercícios

Exercício 43. *Explique com suas palavras o que é uma autoridade certificadora e qual sua importância para garantir a segurança na comunicação.*

Apêndice A

Corpos Finitos

Um *corpo* é uma estrutura matemática formada por um conjunto F e duas operações ($+$ e $\cdot$) tal que:

- Os elementos de F formam um grupo[1] com a operação $+$ e o elemento neutro é o 0.
- Os elementos de F, com exceção do 0, formam um grupo com a operação $\cdot$ e o elemento neutro é o 1.
- Para todo $a, b, c \in F$ temos que $a \cdot (b+c) = a \cdot b + a \cdot c$ (distributividade).

O conjunto dos números reais com as operações convencionais de soma e multiplicação é um corpo.

Estamos interessados, porém, em corpos finitos, também conhecidos como *corpos de Galois*. Um teorema interessante, mas cuja demonstração foge ao escopo dessas notas, estabelece que a cardinalidade de um corpo finito é necessariamente uma potência de um primo. Em símbolos, se $\langle F, +, \cdot \rangle$ é um corpo então $|F| = p^m$ em que p é um número primo e m um inteiro.

Vamos representar por $GF(p)$ o corpo definido pelo conjunto $\mathbb{Z}_p$ e as operações de soma e multiplicação modulo o número primo p. Não é difícil mostrar que $GF(p)$ é um corpo, a parte mais complicada é mostrar que todos os elementos diferentes de 0 possuem inverso em relação a operação de multiplicação. Essa parte, porém, segue diretamente da Proposição 1 e pelo fato de que, com exceção do 0, para qualquer elemento $a \in \mathbb{Z}_p$ temos que $mdc(a, p) = 1$ uma vez que p é primo.

[1] Ver capítulo 10

Exemplo 18. *Vamos considerar o corpo $GF(5)$. Neste corpo temos que:*

$$\begin{aligned} 4+4 &= 3 \\ 2+3 &= 0 \\ -2 &= 3 \\ 3\cdot 3 &= 4 \\ 2\cdot 3 &= 1 \\ 2^{-1} &= 3 \end{aligned}$$

Com um valor pequeno assim, podemos pré-computar sem grandes problemas toda as tabelas de soma e multiplicação:

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

·	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

O corpo de ordem p^m para $m > 1$ será denominado $GF(p^m)$ e é formado pelo conjunto dos polinômios de de grau $m-1$ e cujos coeficientes são elementos em $\mathbb{Z}_p$. A soma de dois elementos em $GF(p^m)$ é dada pela soma dos polinômios.

Exemplo 19. *Considere o corpo $GF(2^4)$ e os elementos x^3+x+1 (que podemos representar como 1011) e x^2+x (que podemos representar como 0110). A soma desses elementos é:*

$$\begin{array}{cccc} x^3 & & x & 1 \\ & x^2 & x & \\ \hline x^3 & x^2 & & 1 \end{array}$$

Um *elemento irredutível* em GFp^m é um polinômio que não pode ser quebrado como a multiplicação de dois polinômios de grau estritamente menor[2].

Exemplo 20. *O polinômio x^4+x^3+x+1 não é irredutível em $GF(2^4)$ pois:*

$$x^4 + x^3 + x + 1 = (x^2 + x + 1)(x^2 + 1)$$

Por outro lado o polinômio $x^4 + x + 1$ é irredutível.

Para multiplicar elementos em $GF(p^m)$ primeiro é preciso fixar um polinômio irredutível de grau m. A multiplicação em $GF(p^m)$ é dada então pela multiplicação dos dois polinômios módulo o polinômio irredutível fixado.

Exemplo 21. *Vamos fixar o polinômio irredutível $P(x) = x^4+x+1$ e vamos calcular a multiplicação entre $x^2 + x$ e $x^3 + x^2 + 1$.*

$$(x^2 + x)(x^3 + x^2 + 1) = x^5 + x^3 + x^2 + x$$

Note que o resultado da operação não é um elemento de $GF(2^4)$, por isso dividimos o resultado por $P(x)$ e ficamos com o resto.

Para isso note que $x^4 = P(x) + x + 1$ e, portanto, $x^5 = xP(x) + x^2 + x$. Concluímos que $x^5 \equiv x^2 + x \ mod(P(x))$. Agora podemos reescrever o resultado da multiplicação como:

$$\begin{aligned}(x^2 + x)(x^3 + x^2 + 1) &\equiv (x^2 + x) + x^3 + x^2 + x \ mod(P(x)) \\ &\equiv x^3 \ mod(P(x))\end{aligned}$$

Como vimos na Seção a cifra de bloco padrão desde 2000, o AES, funciona em turnos que repetem as operações *AddRoundKey*, *SubBytes*, *ShiftRow* e *MixColumns*. A operação *SubBytes* é onde ocorre a fase de confusão. Seja A_i um vetor de um byte (8 bits) que representa um dos 16 elementos do estado que alimenta o turno do algoritmo AES. A operação *SubBytes* tem duas etapas:

1. calcula A_i^{-1}, o inverso de A_i em $GF(2^8)$ cujo polinômio irredutível é $x^8 + x^4 + x^3 + x + 1$

2. multiplica A_i^{-1} por uma matriz e somado por uma constante (*afince mapping*)

[2]Pense como uma espécie de número primo no contexto dos polinômios.

Exemplo 22. *Seja* $A_i = 11000010_2 = C2_{16}$ *(usamos o subscrito para indicar a notação binária ou hexadecimal). Note que* $A_i^{-1} = 00101111_2 = 2F_{16}$*. Para verificar isso vamos multiplicar os dois:*

$$\begin{aligned}
(x^7+x^6+x)(x^5+x^3+x^2+x+1) &= (x^{12}+x^{10}+x^9+x^8+x^7)+ \\
&\quad (x^{11}+x^9+x^8+x^7+x^6)+ \\
&\quad (x^6+x^4+x^3+x^2+x) \\
&= x^{12}+x^{11}+x^{10}+x^4+x^3+x^2+x \\
&= (x^8+x^7+x^5+x^4)+ \\
&\quad (x^7+x^6+x^4+x^3)+ \\
&\quad (x^6+x^5+x^3+x^2)+ \\
&\quad x^4+x^3+x^2+x \\
&= x^8+x^4+x^3+x \\
&= (x^4+x^3+x+1)+x^4+x^3+x \\
&= 1
\end{aligned}$$

O resultado deve passar agora pela fase affine mapping*:*

$$\begin{pmatrix} 1&0&0&0&1&1&1&1 \\ 1&1&0&0&0&1&1&1 \\ 1&1&1&0&0&0&1&1 \\ 1&1&1&1&0&0&0&1 \\ 1&1&1&1&1&0&0&0 \\ 0&1&1&1&1&1&0&0 \\ 0&0&1&1&1&1&1&0 \\ 0&0&0&1&1&1&1&1 \end{pmatrix} \begin{pmatrix} 0\\0\\1\\0\\1\\1\\1\\1 \end{pmatrix} + \begin{pmatrix} 1\\1\\0\\0\\0\\1\\1\\0 \end{pmatrix} = \begin{pmatrix} 0\\0\\1\\0\\0\\1\\0\\1 \end{pmatrix}$$

Na prática, porém, é possível e mais simples précomputar os resultados de SubBytes em uma tabela (a Tabela 2 na Seção A apresenta essa tabela usando representação hexadecimal dos polinômios). No caso a entrada é $C2_{16}$ *e a saída* 25_{16} *exatamente conforme computamos.*

Na fase de descriptografia do AES outra matriz e outra constante são aplicadas e em seguida é calculado o inverso em GF(2^8) de forma a reverter o processo.

Bibliografia

[BDJR97] Mihir Bellare, Anand Desai, Eron Jokipii, and Phillip Rogaway. A concrete security treatment of symmetric encryption. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, FOCS '97, pages 394–, Washington, DC, USA, 1997. IEEE Computer Society.

[BGB04] Nikita Borisov, Ian Goldberg, and Eric Brewer. Off-the-record communication, or, why not to use pgp. In *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society*, WPES '04, pages 77–84, New York, NY, USA, 2004. ACM.

[BM84] Manuel Blum and Silvio Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM J. Comput.*, 13(4):850–864, November 1984.

[Dam90] Ivan Damgård. A design principle for hash functions. In *Proceedings of the 9th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '89, pages 416–427, London, UK, UK, 1990. Springer-Verlag.

[DC06] Christophe De Cannière. Trivium: A stream cipher construction inspired by block cipher design principles. In *Proceedings of the 9th International Conference on Information Security*, ISC'06, pages 171–186, Berlin, Heidelberg, 2006. Springer-Verlag.

[DH76] Withfield Diffie and Martin Hellman. New directions in cryptography. *IEEE Trans. Inf. Theor.*, 22(6):644–654, September 1976.

[EG85] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proceedings of CRYPTO*

84 on Advances in Cryptology, pages 10–18, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[FMS01] Scott Fluhrer, Itsik Mantin, and Adi Shamir. *Weaknesses in the Key Scheduling Algorithm of RC4*, pages 1–24. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[GL89] Oded Goldreich and Leonid A. Levin. A hard-core predicate for all one-way functions. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*, STOC '89, pages 25–32, New York, NY, USA, 1989. ACM.

[GM84] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270 – 299, 1984.

[Kah96] David Kahn. *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet*. Scribner, 1996.

[Kra10] Hugo Krawczyk. *Cryptographic Extraction and Key Derivation: The HKDF Scheme*, pages 631–648. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[Mer90] Ralph C. Merkle. One way hash functions and DES. In *Proceedings of the 9th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '89, pages 428–446, London, UK, UK, 1990. Springer-Verlag.

[NY90] Mori Naor and Moti Yung. Public-key cryptosystems provably secure against chosen ciphertext attacks. In *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*, STOC '90, pages 427–437, New York, NY, USA, 1990. ACM.

[RS16] Ronald L. Rivest and Jacob C. N. Schuldt. Spritz - a spongy RC4-like stream cipher and hash function. *IACR Cryptology ePrint Archive*, 2016:856, 2016.

[RSA78] Ron L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[Sha49] Claude E. Shannon. Communication Theory of Secrecy Systems. *Bell Systems Technical Journal*, 28:656–715, 1949.

[Sin04] Simon Singh. *O livro dos códigos*. RECORD, 2004.

[Yao82] Andrew C. Yao. Theory and application of trapdoor functions. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, pages 80–91, Washington, DC, USA, 1982. IEEE Computer Society.