第6章

对称密钥原语的实际构造

在前面的章节中，我们演示了如何利用加密原语（例如伪随机生成器、伪随机置换和哈希函数）来构造安全的加密方案和消息认证码。然而，我们尚未解决的一个问题是，这些加密原语是如何被构造出来的，或者它们是否真的存在！在下一章中，我们将从理论角度研究这个问题，并展示基于相当弱的假设构造伪随机生成器和伪随机置换。（事实证明，哈希函数更难构造，似乎需要更强的假设。我们将在第8.4.2节看到一个可证明安全的哈希函数构造。）在本章中，我们的重点将是相对启发式但效率更高、在实践中广泛使用的这些原语的构造。

正如刚才提到的，我们将在本章中探索的构造是**启发式**的，因为它们不能基于任何较弱的假设被证明是安全的。然而，这些构造是基于许多设计原则的，其中一些可以通过理论分析来证明。更重要的是，许多这些构造已经经受了多年的公众审查和密码分析尝试，鉴于此，假设这些构造是安全的，是相当合理的。

从某种意义上说，假设“因式分解是困难的”和假设“AES（一种我们将在本章稍后详细研究的分组密码）是伪随机置换”之间没有根本区别。然而，这些假设之间存在着重大的**定性**差异。主要区别在于前一个假设更可信，因为它似乎与一个较弱的要求有关：假设大整数难以分解，比假设AES使用均匀密钥与随机置换无法区分更自然。这些假设之间的其他相关区别是，对因式分解的研究时间比区分AES与随机置换的问题要长得多，而且早在基于它的加密方案出现之前，它就被认为是一个难题。

总而言之，假设本章中描述的推荐构造是安全的，并且人们在实践中可以放心地依赖这些假设是合理的。尽管如此，如果能将加密原语的安全性建立在更弱、更长期的假设之上，那就更好了。正如我们将在第7章中看到的，这（原则上）是可能的；不幸的是，我们将在那里看到的构造比这里描述的构造效率低了几个数量级，因此在实践中用处不大。

本章的目的

本章的主要目的是：(1)介绍现代加密原语构造中使用的一些设计原则，以及(2)向读者介绍一些在现实世界中广泛使用的流行构造。我们提醒注意：

- **本章的目的不是**教读者如何设计新的加密原语。相反，我们相信新原语的设计需要大量的专业知识和努力，不应该轻易尝试。对这个领域开发额外专业知识感兴趣的人，建议阅读本章末尾包含的更高级参考资料。
- **我们无意**在这里介绍我们讨论的各种原语的所有低级细节，并且我们的描述不应被用作实现

的依据。事实上，我们的描述有时是故意不准确的，因为我们省略了与我们试图强调的更广泛的概念性要点不相关的某些细节。

6.1 流密码

回想第3.3.1节，流密码由两个确定性算法（Init、GetBits）定义。Init算法将密钥k和（可选的）初始化向量IV作为输入，并返回一些初始状态st。GetBits算法可用于基于st生成无限比特流$y_1, y_2, \ldots$。流密码的主要要求是它应表现得像一个伪随机生成器，即当k均匀随机选择时，生成的序列$y_1, y_2, \ldots$应与任何计算有界攻击者无法区分的均匀独立比特序列。（在第3.6.1节中，我们注意到流密码有时必须满足更强的安全要求。我们在这里不明确讨论这一点。）

我们已经指出（参见第3.5.1节末尾），流密码可以很容易地从分组密码构造出来，分组密码是一种更强的原语。本节介绍的专用流密码构造的主要动机是效率，特别是在资源受限的环境中（例如，在硬件中，可能希望保持较少的门数）。然而，最近针对流密码的几个构造已经发现了攻击，它们的安全性似乎比分组密码更脆弱。因此，我们建议在可能的情况下使用分组密码（可能以流密码模式）。

6.1.1 线性反馈移位寄存器

我们首先讨论线性反馈移位寄存器（LFSR）。它们在历史上一直用于伪随机数生成，因为它们在硬件中实现效率极高，并且生成具有良好统计特性的输出。然而，仅凭它们本身，它们并**不**能提供密码学上强大的伪随机生成器，事实上我们将展示一个针对LFSR的简单密钥恢复攻击。尽管如此，LFSRs可以作为构建具有更好安全性的流密码的组件。

图6.1：线性反馈移位寄存器。

LFSR由n个寄存器$S_{n-1}, \ldots, S_0$的阵列以及由一组n个反馈系数$c_{n-1}, \ldots, c_0$指定的反馈回路组成。（参见图6.1。）阵列的大小称为LFSR的**度**。每个寄存器存储一个比特，LFSR在任何时间点t的状态st就是寄存器中包含的比特集合。LFSR的状态在一系列的“时钟周期”中更新，方法是将所有寄存器中的值向右移动，并将最左边寄存器的新值设置为当前寄存器中某些子集的XOR，该子集由反馈系数确定。也就是说，如果时间t的状态是$s_{n-1}^{(t)}, \ldots, s_0^{(t)}$，那么下一个时钟周期后的状态$s_{n-1}^{(t+1)}, \ldots, s_0^{(t+1)}$满足：

$$s_i^{(t+1)} := s_{i+1}^{(t)}, \quad i = 0, \ldots, n-2$$

$$s_{n-1}^{(t+1)} := \bigoplus_{i=0}^{n-1} c_i s_i^{(t)}$$

图6.1显示了一个度为4的LFSR，其中$c_0 = c_2 = 1$且$c_1 = c_3 = 0$。

在每个时钟周期，LFSR输出最右边寄存器s_0的值。如果LFSR的初始状态是$s_{n-1}^{(0)}, \ldots, s_0^{(0)}$，那么输出流的前$n$个比特恰好是$s_0^{(0)}, \ldots, s_{n-1}^{(0)}$。下一个输出比特$s_{n-1}^{(1)}$是$\bigoplus_{i=0}^{n-1} c_i s_i^{(0)}$。如果我们将输出比特表示为$y_1, y_2, \ldots$，其中$y_i = s_{i-1}^{(i-1)}$，那么：

$$y_i = s_{i-1}^{(0)}, \quad i = 1, \ldots, n$$

$$y_i = \bigoplus_{j=0}^{n-1} c_j y_{i-n+j-1}, \quad i > n$$

以图6.1中的LFSR为例，如果初始状态是$(0, 0, 1, 1)$，那么前几个时间段的状态是：

$$(0, 0, 1, 1)$$

$$(1, 0, 0, 1)$$

$$(1, 1, 0, 0)$$

$$(1, 1, 1, 0)$$

$$(1, 1, 1, 1)$$

输出（可以从上面的最右列读取）是比特流$1, 1, 0, 0, 1, \ldots$。

LFSR的状态由n个比特组成；因此，LFSR在重复之前最多可以循环2^n个可能的状态。当状态重复时，输出比特也会重复，这意味着输出序列在生成最多2^n个输出比特后将开始重复。**最大长度LFSR**在重复之前会循环遍历所有$2^n - 1$个非零状态。（请注意，如果所有0状态被实现，那么LFSR将永远保持在该状态，这也是我们排除它的原因。）LFSR是否是最大长度仅取决于反馈系数；如果是最大长度，那么一旦它以任何非零状态初始化，它将循环遍历所有$2^n - 1$个非零状态。如何设置反馈系数以获得最大长度LFSR是众所周知的，尽管细节超出了本书的范围。

重构攻击。度为n的最大长度LFSR的输出具有良好的统计特性；例如，每个n比特字符串在LFSR的输出流中出现的频率大致相等。然而，LFSRs并不是好的用于密码学目的的伪随机生成器，因为它们的输出是可预测的。这是因为攻击者在观察最多$2n$个输出比特后，可以重构度为n的LFSR的整个状态。为此，假设某个LFSR的初始状态和反馈系数都是未知的。LFSR的前n

个输出比特$y_1, \ldots, y_n$恰好揭示了初始状态。给定接下来的n个输出比特$y_{n+1}, \ldots, y_{2n}$，攻击者可以建立一个包含n个未知数$c_0, \ldots, c_{n-1}$的n个线性方程组：

$$y_{n+1} = c_{n-1}y_n \oplus \cdots \oplus c_0 y_1$$

$$\vdots$$

$$y_{2n} = c_{n-1}y_{2n-1} \oplus \cdots \oplus c_0 y_n$$

可以证明，对于最大长度LFSR，上述方程（模2）是线性独立的，因此可以唯一确定反馈系数。（可以使用线性代数高效地找到解。）已知反馈系数后，LFSR的所有后续输出比特都可以轻松计算。

6.1.2 增加非线性

LFSR输出比特之间的线性关系正是导致容易攻击的原因。为了阻止此类攻击，我们必须引入一些**非线性**，即除XOR之外的一些操作。有几种不同的方法可以实现这一点，我们只探讨其中的一些。

非线性反馈。修改LFSR的一个明显方法是使反馈回路非线性。非线性反馈移位寄存器（FSR）仍将由一个寄存器阵列组成，每个寄存器包含一个比特。和以前一样，FSR的状态在每个时钟周期中通过将所有寄存器中的值向右移动来更新；然而，现在最左边寄存器的新值是当前寄存器的非线性函数。换句话说，如果时间t的状态是$s_0^{(t)}, \ldots, s_{n-1}^{(t)}$，那么下一个时钟周期后的状态$s_0^{(t+1)}, \ldots, s_{n-1}^{(t+1)}$满足：

$$s_i^{(t+1)} := s_{i+1}^{(t)}, \quad i = 0, \ldots, n-2$$

$$s_{n-1}^{(t+1)} := g(s_0^{(t)}, \ldots, s_{n-1}^{(t)})$$

对于某个非线性函数g。和以前一样，FSR在每个时钟周期输出最右边寄存器s_0的值。

有可能设计出具有最大长度且输出具有良好统计特性的非线性FSR。

非线性组合生成器。另一种方法是在输出序列中引入非线性。在最基本的情况下，我们可以像以前一样有一个LFSR（其中最左边寄存器的新值仍然计算为当前寄存器的线性函数），但是每个时钟周期的输出是所有当前寄存器的非线性函数g，而不是仅仅是最右边的寄存器。这里重要的是g是**平衡**的，即$\Pr[g(s_0, \ldots, s_{n-1}) = 1] \approx 1/2$（概率是针对$s_0, \ldots, s_{n-1}$的均匀选择）；否则，尽管可能难以重构基于输出的LFSR的整个状态，但输出流将是**有偏**的，因此很容易与均匀分布区分开来。

上述方法的一个变体是使用多个LFSR（每个LFSR的单个输出流像以前一样，通过简单地取每个LFSR的最右边寄存器的值来计算），并通过某种非线性方式组合单个LFSR的输出来生成实际的输出流。这产生了所谓的（非线性）**组合生成器**。单个LFSR不需要具有相同的度，事实上，如果它们不具有相同的度，组合生成器的周期长度将最大化。在这里，必须注意确保组合生成器的输出流不会与单个LFSR的任何输出流高度相关；高相关性可能导致对单个LFSR的攻击，从而破坏在构造中使用多个LFSR的目的。

6.1.3 Trivium

为了说明上一节中的想法，我们简要描述流密码**Trivium**。这个流密码是在eSTREAM项目（一个欧洲努力，于2008年完成，目标是识别新的流密码）的组合中被选中的。Trivium旨在具有简单的描述并支持紧凑的硬件实现。

图6.2：Trivium的示意图，从上到下有三个耦合的非线性FSR A, B和C。

Trivium使用三个耦合的非线性FSR，分别表示为A, B和C，度分别为93、84和111。（参见图6.2。）Trivium的状态st就是组成所有这些FSR中的值的288个比特。Trivium的GetBits算法的工作方式如下：在每个时钟周期，每个FSR的输出是其最右边寄存器和另一个额外寄存器的XOR；Trivium的输出是三个FSR的输出比特的XOR。这些FSR是**耦合**的：在每个时钟周期，每个FSR最左边寄存器的新值被计算为**同一**FSR中一个寄存器和**第二个**FSR中一个寄存器子集的函数。（A的反馈函数取决于A中的一个寄存器和C中的四个寄存器；B的反馈函数取决于B中的一个寄存器和A中的四个寄存器；C的反馈函数取决于C中的一个寄存器和B中的四个寄存器。）在每种情况下，反馈函数都是非线性的。

Trivium的Init算法接受一个80比特密钥和一个80比特IV。密钥加载到A的80个最左边寄存器中，IV加载到B的80个最左边寄存器中。其余寄存器设置为0，除了C的三个最右边寄存器设置为1。然后运行GetBits 4.288次（输出被丢弃），并将生成的状态作为st_0。

迄今为止，针对完整Trivium密码，尚未发现比穷举密钥搜索更有效的密码分析攻击。

6.1.4 RC4

LFSRs在硬件中实现效率高，但在软件中性能差。出于这个原因，人们探索了流密码的替代设计。一个突出的例子是**RC4**，由Ron Rivest于1987年设计。RC4以其速度和简单性而著称，并抵抗了多年的严重攻击。它今天被广泛使用，我们讨论它也是出于这个原因；然而，我们提醒读者，最近的攻击已经显示出RC4中存在严重的密码学弱点，因此**不应再使用它**。

算法6.1 RC4的Init算法 输入： 16字节密钥k **输出：** 初始状态(S, i, j)（注意：所有加法都

是模256进行） **for** $i = 0$ **to** 255: $S[i] := i$ $k[i] := k[i \bmod 16]$ $j := 0$ **for** $i = 0$ **to** 255: $j := j + S[i] + k[i]$ **Swap** $S[i]$ **and** $S[j]$ $i := 0, j := 0$ **return** (S, i, j)

RC4的状态是一个256字节的数组S，它总是包含元素$0, \ldots, 255$的一个置换，以及两个值$i, j \in \{0, \ldots, 255\}$。RC4的Init算法如算法6.1所示。为简单起见，我们假设密钥k是16字节（128比特）长，尽管该算法可以处理1字节到256字节长的密钥。我们将S的字节索引为$S, \ldots, S$，将密钥的字节索引为$k, \ldots, k$。

在初始化过程中，S首先设置为恒等置换（即$S[i] = i$对所有i成立），k通过重复扩展到256个字节。然后，S的每个条目至少与S的另一个条目在一个“伪随机”位置进行交换。索引i, j设置为0，(S, i, j)作为初始状态输出。

然后使用该状态生成一系列输出比特，如算法6.2所示。索引i简单地递增（模256），j以某种“伪随机”方式改变。$S[i]$和$S[j]$的条目被交换，并且S在位置$S[i] + S[j]$（同样模256计算）的值被输出。请注意，S的每个条目在每256次迭代中至少与S的某个其他条目（可能就是它自己）交换一次，确保置换S的良好“混合”。

算法6.2 RC4的GetBits算法 输入： 当前状态(S, i, j) **输出：** 输出字节y；更新状态(S, i, j)（注意：所有加法都是模256进行） $i := i + 1$ $j := j + S[i]$ **Swap** $S[i]$ **and** $S[j]$ $t := S[i] + S[j]$ $y := S[t]$ **return** $(S, i, j), y$

RC4最初设计时没有将IV作为输入；然而，在实践中，IV通常通过简单地将其与实际密钥k'串联起来进行合并，然后再进行初始化。也就是说，选择所需长度的随机IV，将k设置为IV和k'的串联（可以通过前置或附加IV来完成），然后像算法6.1中那样运行Init以生成初始状态。然后像以前一样使用算法6.2生成输出比特。假设RC4以非同步模式使用（参见第3.6.1节），那么IV将以明文形式发送给接收方——接收方可能已经拥有实际的密钥k'——从而使他们能够生成相同的初始状态，因此也能生成相同的输出流。这种合并IV的方法用于**有线等效隐私**（WEP）加密标准中，用于保护802.11无线网络中的通信。

人们应该关注这种相对**临时**（ad hoc）的方式修改RC4以接受IV。即使RC4在最初设计时仅使用密钥是安全的流密码，也没有理由相信它在以这种方式修改为使用IV时仍是安全的。事实上，与密钥相反，IV会向攻击者泄露（因为它以明文形式发送）；此外，在非同步模式下使用RC4时，使用不同的IV和相同的固定密钥k'——意味着使用**相关**的值k来初始化RC4的状态。正如我们将在下面看到的，这两个问题都会导致在以这种方式使用RC4时产生攻击。

针对RC4的攻击。尽管RC4在现代系统中普遍存在，但针对RC4的各种攻击已经为人所知多年。因此，RC4不应再使用；而应使用更现代的流密码或分组密码来替代它。

我们首先演示一个不依赖于诚实方使用IV的简单统计攻击。该攻击利用了RC4的第二个输出字节（略微）偏向于0的事实。令S_t表示GetBits迭代t次后的数组S的状态，其中S_0表示初始状态。启发式地将S_0视为$\{0,\ldots,255\}$的均匀置换，它以概率$1/256 \cdot (1-1/255) \approx 1/256$满足$S_0[1] \neq 0$且$X \stackrel{\text{def}}{=} S_0[1] \neq 2$。假设我们暂时认为这是事实。在GetBits的第一次迭代中，i递增到1，j设置为$S_0[i] = S_0[1] = X$。然后$S[1]$和$S[X]$被交换，因此在迭代结束时我们有$S_1[X] = S_0[1] = X$。在第二次迭代中，i递增到2，j被赋值为

$$j + S_1[i] = X + S_1[2] = X + S_0[2] = X$$

因为$S_0[2] = 0$。然后$S_1[2]$和$S_1[X]$被交换，因此$S_2[X] = S_1[2] = S_0[2] = 0$且$S_2[2] = S_1[X] = X$。最后，输出$S_2[i] + S_2[j] = S_2[2] + S_2[X] = X$位置的值；这恰好是$S_2[X] = 0$的值。

当$S_0[1] \neq 0$时，第二个输出字节是均匀分布的。总的来说，第二个输出字节是0的概率是

$$\Pr[S_0[1] = 0 \text{ and } S_0[1] \neq 2] + \frac{1}{256} \cdot (1 - \Pr[S_0[1] = 0 \text{ and } S_0[1] \neq 2]) = \frac{1}{256} + \frac{1}{256} \cdot (1-0) \approx \frac{2}{256},$$

大约是均匀值期望值的两倍。

仅凭上述攻击，可能不会被视为特别严重的攻击，尽管它似乎表明RC4存在潜在的结构问题。当IV通过前置到密钥中合并时，可以针对RC4进行更严重的攻击。这种攻击可以用于恢复密钥，无论其长度如何，因此比上述区分攻击更严重。重要的是，这种攻击可以用于完全**破解**前面提到的WEP加密标准，并对促使该标准被替换产生了影响。

该攻击的核心是扩展密钥k的前n个字节知识到k的前$(n+1)$个字节知识的方法。请注意，当IV被前置到实际密钥k'（因此$k = IV||k'$）时，密钥k的前几个字节是免费提供给攻击者的！如果IV的长度是n个字节，那么对手可以使用此攻击首先恢复k的第$(n+1)$个字节（即实际密钥k'的第一个字节），然后是k的下一个字节，依此类推，直到推导出整个密钥。

假设IV是3个字节长，如WEP的情况。攻击者等到IV的前两个字节具有特定形式。该攻击可以通过IV的前两个字节的几种可能性进行，但我们关注IV采取形式$IV = (3, 255, X)$的情况，其中X是任意字节。这意味着，当然，在算法6.1中，$k[0] = 3, k[1] = 255$且$k[2] = X$。可以验证，在Init的第二个循环的前四次迭代之后，我们有

$$S[0] = 3, S[1] = 0, S[3] = X + 6 + k[0]. \quad (6.1)$$

在Init算法接下来的252次迭代中，i总是大于3。因此，只要j不取值0、1或3，$S[0], S[1]$和$S[3]$的值就不会随后被修改。如果我们（启发式地）将j视为在每次迭代中取均匀值，这意味着$S[0], S[1]$和$S[3]$随后不被修改的概率是$(253/256)^{252} \approx 0.05$，或5%的时间。假设是这种情况，GetBits输出

的第一个字节将是$S = X + 6 + k$；由于X是已知的，这揭示了k。

因此，攻击者知道在5%的时间内，输出的第一个字节与k相关，如上所述。（这比随机猜测要好得多，随机猜测正确的概率是$1/256 = 0.4\%$。）因此，通过收集足够多的具有正确形式的IV的第一个输出字节样本，攻击者可以高置信度地估计k。

6.2 分组密码

回想第3.5.1节，分组密码是一个高效的、带密钥的置换$F : \{0,1\}^n \times \{0,1\}^l \to \{0,1\}^l$。这意味着由$F_k(x) \stackrel{\text{def}}{=} F(k,x)$定义的函数$F_k$是一个双射（即一个置换），并且给定$k$，$F_k$及其逆$F_k^{-1}$都可以高效计算。我们称$n$为**密钥长度**，$l$为$F$的**分组长度**，这里我们明确允许它们不同。密钥长度和分组长度现在是固定常数，而在第3章中，它们被视为安全参数的函数。这使我们处于**具体安全**（concrete security）而不是渐近安全（asymptotic security）的设置中。

分组密码的安全要求非常严格，并且一个分组密码通常只有在已知最佳攻击（无需预处理）的时间复杂度大致相当于对密钥的穷举搜索时，才被认为是“好的”。因此，如果一个密钥长度为$n = 256$的密码可以在2^{128}时间内被破解，那么即使2^{128}时间的攻击仍然是不可行的，该密码（通常）也被认为是不安全的。相比之下，在渐近设置中，$2^{n/2}$复杂度的攻击不被认为是高效的，因为它需要指数时间（因此，一个可能存在这种攻击的密码可能仍然满足伪随机置换的定义）。然而，在具体设置中，我们必须关注攻击的**实际**复杂度（而不是它的渐近行为）。此外，人们担心这种攻击的存在可能表明密码设计中存在一些更根本的弱点。

分组密码被设计为至少表现为（强）伪随机置换；参见定义3.28。将分组密码建模为伪随机置换允许对基于分组密码的构造进行安全性证明，并且也明确了分组密码的必要要求。对分组密码应该实现什么有一个扎实的理解，对于它们的设计至关重要。将分组密码建模为伪随机置换的观点，至少在最近，已成为影响它们设计的一个主要因素。作为一个例子，我们将在本章稍后遇到的最近的**高级加密标准**（AES）的提案征集，说明了以下评估标准：

> 算法提供的安全性是最重要的因素。... 算法将根据以下因素进行评判...
>
> - 算法输出与随机置换无法区分的程度...

现代分组密码适用于我们在这本书中看到的，所有使用伪随机置换（或伪随机函数）的构造。通常，分组密码的设计（并假设）满足更强的安全属性，我们将在第6.3.1节中简要讨论。

尽管分组密码本身不是加密方案，但对分组密码F的攻击的标准术语是：

- 在**已知明文攻击**中，攻击者被赋予输入/输出对$\{(x_i, F_k(x_i))\}$（对于未知密钥k），其中

$\{x_i\}$在攻击者的控制之外。

- 在**选择明文攻击**中，攻击者被赋予一系列由攻击者选择的输入$\{x_i\}$的$\{F_k(x_i))\}$（同样，对于未知密钥k）。
- 在**选择密文攻击**中，攻击者被赋予由攻击者选择的$\{x_i\}$的$\{F_k(x_i))\}$，以及由选择的$\{y_i\}$的$\{F_k^{-1}(y_i))\}$。

除了使用上述方法来区分F_k和均匀置换外，我们还将对**密钥恢复攻击**感兴趣，其中攻击者在与F_k交互后能够恢复密钥k。（这比能够区分F_k与均匀置换更强。）

关于这种分类法，**伪随机置换**在选择明文攻击下无法与均匀置换区分，而**强伪随机置换**甚至在选择密文攻击下也无法区分。

6.2.1 代换-置换网络

分组密码必须表现得像随机置换。在l比特字符串上有$2^l!$个置换，因此用l比特分组长度表示一个任意置换需要$\log(2^l!) \approx l \cdot 2^l$比特。这对于$l > 20$是不切实际的，对于$l > 50$是不可行的。（展望未来，现代分组密码的分组长度$l \geq 128$。）设计分组密码的挑战是构造一个具有简洁描述（即短密钥）的置换集合，使其表现得像随机置换。特别是，正如在两个仅在一个比特上不同的输入处评估随机置换应该产生两个（几乎）独立的输出（它们不完全独立，因为它们不能相等）一样，将$F_k(\cdot)$的输入更改一个比特，其中k是均匀且对攻击者未知，也应该产生一个（几乎）独立的结果。这意味着输入中的一个比特更改应该"影响"输出的**每个**比特。（请注意，这并不意味着所有输出比特都会更改——这与随机置换所期望的行为不同。相反，我们只是非正式地意味着输出的每个比特以大约一半的概率更改。）实现这一点需要一些工作。

混淆-扩散范式。除了在完美保密方面的工作之外，香农还引入了一个构造简洁、看似随机的置换的基本范式。基本思想是利用许多具有小分组长度的较小随机（或看似随机）置换$\{f_i\}$来构造一个具有大分组长度的看似随机的置换F。让我们看看这在最基本的层面上是如何工作的。假设我们希望F具有128比特的分组长度。我们可以如下定义F：F的密钥k将指定16个置换$f_1, \ldots, f_{16}$，每个置换具有8比特（1字节）的分组长度。给定输入$x \in \{0,1\}^{128}$，我们将其解析为16个字节$x_1 \ldots x_{16}$，然后设置

$$F_k(x) = f_1(x_1) \| \ldots \| f_{16}(x_{16}). \quad (6.2)$$

这些轮函数$\{f_i\}$被称为对F引入了**混淆**。

然而，应该立即清楚，如上定义的F不会是伪随机的。具体来说，如果x和x'仅在第一个比特上不同，那么$F_k(x)$和$F_k(x')$将仅在第一个字节上不同（无论密钥k如何）。相比之下，如果F是一个真正的随机置换，那么输入的第一个比特的更改应该会影响输出的所有字节。

因此，引入了一个**扩散**步骤，通过使用**混合置换**来置换或“混合”输出的比特。这具有将局部更改（例如，第一个字节的更改）传播到整个分组的效果。混淆/扩散步骤——一起称为一**轮**——重复多次。这有助于确保输入中的单个比特更改将影响输出的所有比特。

作为一个例子，遵循这种方法的两轮分组密码将如下操作。首先，通过计算中间结果$f_1(x_1)\| \ldots \| f_{16}(x_{16})$来引入混淆，如等式（6.2）所示。然后，将结果的比特“洗牌”或重新排序，得到x'。然后，计算$f'_1(x'_1)\| \ldots \| f'_{16}(x'_{16})$（其中$x' = x'_1 \ldots x'_{16}$），使用可能不同的函数$f'_i$，并将结果的比特置换以给出输出$x''$。$\{f_i\}, \{f'_i\}$和混合置换可以像我们上面描述的那样是随机的并且取决于密钥。然而，在实践中，它们是经过特殊设计和固定的，并且密钥以不同的方式合并，我们将在下面描述。

代换-置换网络。代换-置换网络**（SPN）可以看作是混淆-扩散范式的直接实现。不同之处在于，现在的轮函数具有特定的形式，而不是从某个域上所有可能的置换集合中选择。具体来说，我们不是让密钥k的（一部分）指定一个任意置换f，而是固定一个公共的“代换函数”（即置换）S，称为S-盒，然后让k定义由$f(x) = S(k \oplus x)$给出的函数f。

为了具体了解这是如何工作的，考虑一个基于8个8比特（1字节）S-盒$S_1, \ldots, S_8$集合的64比特分组长度的SPN。（参见图6.3。）评估密码分一系列**轮**进行，在每轮中，我们对该轮的64比特输入x（第一轮的输入就是密码的输入）应用以下操作序列：

1. **密钥混合：** 设置$x := x \oplus k$，其中k是当前轮的**子密钥**；
2. **代换：** 设置$x := S_1(x_1)\| \ldots \| S_8(x_8)$，其中$x_i$是$x$的第$i$个字节；
3. **置换：** 置换x的比特以获得该轮的输出。

每轮的输出作为下一轮的输入。在最后一轮之后，有一个最终的密钥混合步骤，结果是密码的输出。（根据**科克霍夫原则**，我们假设S-盒和混合置换是公开的，并且对任何攻击者都是已知的。这意味着，如果没有最终的密钥混合步骤，最后的代换和置换步骤将不会提供额外的安全性，因为它们不依赖于密钥。）图6.4显示了一个具有16比特分组长度的SPN的高级结构，并在每轮中使用了不同的4比特S-盒集合。

每轮使用不同的**子密钥**（或**轮密钥**）。分组密码的实际密钥有时称为**主密钥**。轮子密钥根据**密钥编排**从主密钥派生。密钥编排通常很简单，可能只是采用主密钥比特的不同子集，尽管也可以定义更复杂的编排。一个r轮SPN有r个（完整的）密钥混合轮、S-盒代换轮和混合置换应用轮，接着是最后的密钥混合步骤。（这意味着在一个r轮SPN中，使用了$r + 1$个子密钥。）

任何SPN都是**可逆**的（给定密钥）。为了证明这一点，我们展示了给定SPN的输出和密钥，可以恢复输入。只需证明网络的每一轮都可以反转；这意味着整个SPN可以通过从最后一轮向后工作来反转。反转单轮很容易：混合置换很容易反转，因为它只是比特的重新排序。由于S-盒是置换

（即一对一），它们也可以反转。然后可以将结果与适当的子密钥进行XOR以获得原始输入。因此：

命题6.3 令F是一个由SPN定义的带密钥函数，其中S-盒都是置换。那么无论密钥编排和轮数如何，F_k对于任何k都是一个置换。

图6.3：代换-置换网络的单轮。

图6.4：代换-置换网络。

轮数，以及S-盒、混合置换和密钥编排的精确选择，最终决定了一个给定的分组密码是**微不足道地可破解**还是**高度安全**。我们现在讨论S-盒和混合置换设计背后的一个基本原则。

雪崩效应。正如反复指出的那样，任何分组密码的一个重要特性是，输入中的微小更改必须“影响”输出的每个比特。我们称之为雪崩效应。在代换-置换网络中引发雪崩效应的一种方法是确保以下两个属性成立（并使用足够多的轮数）：

1. S-盒的设计使得更改S-盒输入的单个比特，会更改S-盒输出中**至少两个**比特。
2. 混合置换的设计使得任何给定S-盒的输出比特用作下一轮中**多个**S-盒的输入。

为了说明这如何产生雪崩效应，至少在启发式上，假设所有S-盒都满足更改S-盒输入的单个比特会导致S-盒输出中恰好两个比特的更改，并且混合置换是按照上述要求选择的。为了具体，假设S-盒的输入/输出大小是8比特，分组密码的分组长度是128比特。现在考虑将分组密码应用于仅在一个比特上不同的两个输入时会发生什么：

1. 在第一轮之后，中间值恰好在**两个**比特位置不同。这是因为XORing当前子密钥保持了中间值中的1比特差异，因此除一个S-盒外，所有S-盒的输入都相同。在该输入不同的S-盒中，S-盒的输出导致2比特差异。应用于结果的混合置换改变了这些差异的位置，但保持了2比特差异。
2. 在第一轮结束时应用的混合置换将中间结果不同的两个比特位置传播到第二轮中的**两个不同**的S-盒。即使在与前一轮结果进行适当的子密钥XOR之后，这也仍然成立。因此，在第二轮中，现在有两个S-盒接收到仅有一个比特不同的输入。因此，在第二轮结束时，中间值在**4个比特**上不同。
3. 继续同样的论证，我们预计在第3轮之后中间值将有8个比特受到影响，第4轮之后有16个比特受到影响，并且在第7轮结束时输出的所有128个比特都将受到影响。

最后一点不是很精确，并且肯定可能在某些轮结束时差异少于预期。（事实上，情况必须如此，因为输出的比特也不能全部不同。）因此，通常习惯使用**远多于**7轮。然而，上述分析给出了一个**下界**：如果使用的轮数少于7，那么肯定有一些输出比特不会受到输入中单个比特更改的影

响，这意味着有可能将该密码与随机置换区分开来。

人们可能期望设计S-盒的“最佳”方法是随机选择它们（受到它们是置换的限制）。有趣的是，事实证明情况并非如此，至少如果我们想满足前面提到的设计标准。考虑S-盒对4比特输入进行操作的情况，令x和x'是两个不同的值。令$y = S(x)$，现在考虑选择均匀的$y' \neq y$作为$S(x')$的值。有4个字符串与y仅在1比特上不同，因此以概率$4/15$我们将选择y'，它与y相差不少于两个比特。当我们考虑所有相差单个比特的输入对时，问题会更加复杂。

我们根据这个例子得出结论，通常，S-盒必须**精心设计**，而不是盲目随机选择。随机S-盒也不利于防御我们将在第6.2.6节中展示的攻击。

如果分组密码也应该是**强伪随机**的，那么雪崩效应也必须适用于其**逆**。也就是说，更改输出的单个比特应该影响输入的每个比特。为此，如果S-盒被设计成使得更改S-盒输出的单个比特会更改S-盒输入的至少两个比特，则会很有用。在两个方向上实现雪崩效应是进一步增加轮数的另一个原因。

攻击减少轮数的SPN

经验以及多年的密码分析努力表明，只要在S-盒、混合置换和密钥编排的选择上小心谨慎，代换-置换网络是构造伪随机置换的良好选择。**高级加密标准**（Advanced Encryption Standard，AES，将在第6.2.5节中描述）在结构上类似于上述代换-置换网络，并且被广泛认为是强大的伪随机置换。

以这种方式构造的密码F的强度在很大程度上取决于**轮数**。为了更深入地了解代换-置换网络，我们将演示对具有很少轮数的SPN的攻击。这些攻击很简单，但值得一看，因为它们明确地证明了为什么需要大量的轮数。

一个琐碎的案例。我们首先考虑一个琐碎的案例，其中F包含一整轮，但没有最终的密钥混合步骤。我们证明，一个攻击者只需给定一个输入/输出对(x, y)就可以轻松学到密钥k，使得$y = F_k(x)$。攻击者从输出值y开始，然后反转混合置换和S-盒。正如前面所指出的，她可以做到这一点，因为混合置换和S-盒的完整规范是公开的。攻击者计算出的中间值恰好是$x \oplus k$（假设主密钥在网络中的唯一一轮中用作子密钥，不失一般性）。由于攻击者也知道输入x，她可以立即推导出密钥k。因此这是一个**彻底的破解**。

尽管这是一个琐碎的攻击，但它证明了在任何代换-置换网络中，在最终的子密钥混合之后执行S-盒代换或应用混合置换不会增加安全性。

攻击单轮SPN。现在我们有一个完整的轮，接着是一个密钥混合步骤。为了具体起见，我们假设分组长度为64比特，S-盒的输入/输出长度为8比特（1字节）。我们假设为两个密钥混合步

骤使用了独立的64比特子密钥k_1, k_2，因此SPN的主密钥$k_1||k_2$长128比特。

第一个观察是，我们可以扩展上述琐碎案例中的攻击，以在这里给出使用远少于2^{128}工作量的密钥恢复攻击。思路如下：给定一个输入/输出对(x, y)，像以前一样，攻击者枚举第二轮子密钥k_2的所有可能值。对于每个这样的值，攻击者可以反转最终的密钥混合步骤，得到一个候选中间值y'。我们已经在上面看到，给定一个输入x和一个（完整的）SPN轮的输出y'，可以轻松识别出唯一的可能的子密钥k_1。因此，对于k_2的每个可能的选择，攻击者导出了一个唯一的对应k_1，使得$k_1||k_2$可以是主密钥。通过这种方式，攻击者可以在2^{64}时间内获得一个包含2^{64}个主密钥可能性的列表。这可以通过使用额外的输入/输出对，在大约2^{64}的额外时间内缩小范围；另见下文。

通过注意到输出的单个比特仅取决于主密钥的**部分**，可以进行更好的攻击。像以前一样，固定一些给定的输入/输出对(x, y)。现在，对手将枚举k_2的第一个字节的所有可能值。它可以将每个这样的值与y的第一个字节进行XOR，以获得第一个S-盒输出的候选值。反转这个S-盒，攻击者学到了进入该S-盒的候选值。由于该S-盒的输入是x的8个比特和k_1的8个比特的XOR（其中这些比特的位置取决于第一轮混合置换，并且对攻击者是已知的），这为k_1的8个比特提供了候选值。

总结一下：对于k_2的第一个字节的每个候选值，存在一个唯一的可能对应值，对应于k_1的某些8个比特。换句话说，对于主密钥的某些16个比特，攻击者已将这些比特的可能值的数量从2^{16}减少到2^8。攻击者可以在2^8时间内将所有这些可行值制成表格。这可以对k_2的每个字节重复进行，给出8个列表——每个列表包含2^8个值——它们共同刻画了整个主密钥的可能值。因此，攻击者已将可能的主密钥数量减少到$(2^8)^8 = 2^{64}$，与早期攻击一样。然而，完成这项工作的总时间现在是$8 \cdot 2^8 = 2^{11}$，这是一个显著的改进。

攻击者可以使用额外的输入/输出对进一步减少可能密钥的空间。考虑主密钥的某个16个比特集合的2^8个可行值列表。攻击者知道列表中正确的值必须与攻击者学到的任何额外输入/输出对(x', y')一致。启发式地，列表中的任何不正确值与某个额外输入/输出对(x', y')一致的概率不比随机猜测好；由于表中的每个16比特值都可以用于计算给定输入x'的一个字节的输出，我们期望不正确值与实际输出y'一致的概率为2^{-8}。因此，少量额外的输入/输出对将足以将所有表格缩小到每个表格只有一个值，此时整个主密钥是已知的。

这里有一个重要的教训。由于密钥的不同部分可以从其他部分中分离出来，所以攻击是可能的。因此，需要**进一步的扩散**以确保密钥的所有比特影响输出的所有比特。为此需要多轮。

攻击两轮SPN。可以扩展上述思路，以对在每轮中使用独立子密钥的两轮SPN进行优于穷举搜索的密钥恢复攻击；我们将其留作练习。

相反，我们只是简单地指出，两轮SPN不会是一个好的伪随机置换。这里我们依赖于前面提到的事实，即雪崩效应不会在仅两轮之后发生（当然，这取决于密码的分组长度和S-盒的输入/输出长度，但在合理的参数下会是这种情况）。如果攻击者了解在两个仅在一个比特上不同的输入上评估SPN的结果，他可以区分两轮SPN与均匀置换：在两轮SPN中，两个输出的许多比特将是相同的，这对于随机置换来说是意料之外的。

6.2.2 Feistel网络

Feistel网络提供了另一种构造分组密码的方法。Feistel网络相对于代换-置换网络的一个优势是，用于Feistel网络的基础函数——与SPN中使用的S-盒相反——**不需要是可逆的**。因此，Feistel网络提供了一种从不可逆组件构造可逆函数的方法。这很重要，因为一个好的分组密码应该具有“非结构化”行为（因此看起来是随机的），但要求构造的所有组件都是可逆的，本质上引入了结构。要求可逆性也对S-盒引入了额外的约束，使它们更难设计。

Feistel网络分一系列轮操作。在每轮中，以如下所述的方式应用带密钥的轮函数。轮函数不需要是可逆的。它们通常由S-盒和混合置换等组件构成，但Feistel网络可以处理**任何**轮函数，无论其设计如何。

在**平衡Feistel网络**中（我们将只考虑这种类型），第i轮函数f_i接受一个子密钥k_i和一个$l/2$比特字符串作为输入，并输出一个$l/2$比特字符串。与SPN的情况一样，使用一个主密钥k来派生每轮的子密钥。当选择某个主密钥从而确定每个子密钥k_i时，我们通过$f_i(R) \stackrel{\text{def}}{=} f_i(k_i, R)$定义$f_i : \{0,1\}^{l/2} \to \{0,1\}^{l/2}$。请注意，轮函数$f_i$是固定且公开已知的，但$f_i$取决于主密钥，因此攻击者不知道。

Feistel网络的第i轮操作如下。该轮的输入分为两半，表示为L_{i-1}和R_{i-1}（分别为“左”和“右”两半）。如果密码的分组长度是l比特，那么L_{i-1}和R_{i-1}的长度都是$l/2$。该轮的输出(L_i, R_i)是

$$L_i := R_{i-1} \quad \text{and} \quad R_i := L_{i-1} \oplus f_i(R_{i-1}). \quad (6.3)$$

在一个r轮Feistel网络中，网络的l比特输入被解析为(L_0, R_0)，输出是通过应用所有r轮后获得的l比特值(L_r, R_r)。图6.5展示了一个三轮Feistel网络。

图6.5：三轮Feistel网络。

反转Feistel网络。Feistel网络是可逆的，**无论**$\{f_i\}$如何（因此也无论轮函数$\{f_i\}$如何）。为了证明这一点，我们只需要证明网络的每一轮都可以反转，如果$\{f_i\}$是已知的。给定第i轮的输出(L_i, R_i)，我们可以如下计算(L_{i-1}, R_{i-1})：首先设置$R_{i-1} := L_i$。然后计算

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

这给出了该轮的输入值(L_{i-1}, R_{i-1})（即，它计算了等式（6.3）的逆）。请注意，f_i仅在**正向**方向上进行评估，因此它不需要是可逆的。因此我们有：

命题6.4 令F是一个由Feistel网络定义的带密钥函数。那么无论轮函数$\{f_i\}$和轮数如何，F_k对于所有k都是一个高效可逆的置换。

与代换-置换网络的情况一样，当轮数太少时，Feistel网络也可能受到攻击。我们将在下一节讨论DES时看到此类攻击。有关Feistel网络安全性的理论结果将在第7.6节中讨论。

6.2.3 DES – 数据加密标准

数据加密标准（Data Encryption Standard，DES）由IBM（在美国国家安全局的帮助下）在1970年代开发，并于1977年被采纳为美国的联邦信息处理标准。在其基本形式中，DES因其56比特的短密钥长度而不再被认为是安全的，这使其容易受到穷举搜索攻击。然而，它今天仍然以加强形式的**三重DES**（Triple-DES，在第6.2.4节中描述）被广泛使用。

DES具有重要的历史意义。它在密码学界受到了密集的审查，可以说是历史上任何其他密码算法所没有的。普遍的共识是，除了其密钥长度之外，DES是一个设计得**极其出色**的密码。事实上，即使多年以后，DES在实践中已知的最佳攻击仍然是对所有2^{56}个可能密钥的穷举搜索。（正如我们将看到的，存在对DES的重要的理论攻击，需要较少的计算；然而，这些攻击假设了在实践中似乎难以实现的某些条件。）

在本节中，我们提供DES主要组件的高级概述。我们强调，我们将不提供在每个细节上都正确的完整规范，并且我们的描述将省略设计的某些部分。我们的目标是介绍DES构造背后的基本思想，而不是所有低级细节；对这些细节感兴趣的读者可以查阅本章末尾的参考资料。

DES的设计

DES分组密码是一个16轮的Feistel网络，分组长度为64比特，密钥长度为56比特。在所有16轮中使用了相同的轮函数f。该轮函数接受一个48比特子密钥，以及一个32比特输入（即半分组）。DES的**密钥编排**用于从56比特主密钥派生出一系列48比特子密钥$k_1, \ldots, k_{16}$。DES的密钥编排相对简单，每个子密钥k_i是主密钥的48比特的置换子集。就我们的目的而言，只需注意56比特的主密钥被分成两半——一个包含28比特的“左半部分”和一个“右半部分”。（这个划分发生在对密钥应用初始置换之后，但我们在描述中忽略了这一点。）在每轮中，子密钥的左侧24比特取自主密钥左侧28比特的某个子集，子密钥的右侧24比特取自主密钥右侧28比特的某个子集。我们强调，整个密钥编排（包括将比特划分成左右两半的方式，以及哪些比特用于形成

每个子密钥k_i的方式）是固定且公开的，唯一的秘密是主密钥本身。

DES轮函数。DES轮函数f——有时称为DES混淆函数——是使用我们以前分析过的范式构造的：它（本质上）只是一个代换-置换网络！更详细地说，计算$f(k_i, R)$，其中$k_i \in \{0,1\}^{48}$和$R \in \{0,1\}^{32}$，过程如下：首先，R被扩展到48比特值R'。这是通过简单地复制R的一半比特来完成的；我们通过$R' := E(R)$表示，其中E称为**扩展函数**。接着，计算过程与我们前面讨论的SPN完全一样：扩展值R'与k_i进行XOR，它也是48比特长，得到的结果被分成8个块，每个块6比特长。每个块通过一个（不同）S-盒，该S-盒接受一个6比特输入并产生一个4比特输出；连接8个S-盒的输出给出一个32比特的结果。然后将混合置换应用于该结果的比特，以获得最终输出。参见图6.6。

图6.6：DES混淆函数。

与我们最初讨论SPN相比的一个区别是，这里的S-盒**不可逆**；事实上，它们不可能是可逆的，因为它们的输入比输出长。关于S-盒结构细节的进一步讨论将在下面给出。

我们再次强调，上述描述中的所有内容（包括S-盒本身以及混合置换）都是**公开已知**的。唯一的秘密是用于派生所有子密钥的主密钥。

S-盒和混合置换。构成f“核心”的八个S-盒是DES构造的关键元素，并且经过了非常精心的设计。对DES的研究表明，如果S-盒稍作修改，DES将更容易受到攻击。

这应该对任何希望设计分组密码的人起到警告作用：看似任意的选择根本不是任意的，如果做得不对，可能会使整个构造不安全。

回想一下，每个S-盒将一个6比特输入映射到4比特输出。每个S-盒可以看作是一个具有4行16列的表格，其中表格的每个单元格包含一个4比特的条目。一个6比特输入可以通过以下方式视为索引表格的$2^6 = 64 = 4 \times 16$个单元格之一：第一个和最后一个输入比特用于选择表格的行，比特2-5用于选择表格的列。表格中某个位置的4比特条目表示与该位置关联的输入的输出值。

DES S-盒具有以下属性（其中包括）：

1. 每个S-盒是一个4对1函数。（即，恰好4个输入映射到每个可能的输出。）这源于以下属性。
2. 表格中的每一行恰好包含16个可能的4比特字符串**一次**。
3. 更改S-盒任何输入的**一个**比特总是更改输出的**至少两个**比特。

混合置换也经过了精心设计。特别是，它具有以下特性：任何S-盒的四个输出比特将影响下一轮

的六个S-盒的输入。（这是可能的，因为在计算S-盒之前，在下一轮中应用了扩展函数。）

DES雪崩效应。混淆函数的设计确保了DES表现出强大的雪崩效应。为了理解这一点，我们将追踪DES计算中仅相差单个比特的两个输入之间的中间值差异。我们将密码的两个输入表示为(L_0, R_0)和(L_0', R_0')，其中我们假设$R_0 = R_0'$，因此单个比特差异发生在输入的左半部分（这可能有助于参考等式（6.3）和图6.6）。在第一轮之后，中间值(L_1, R_1)和(L_1', R_1')仍然只相差单个比特，尽管现在这个差异在右半部分。在DES的第二轮中，每个输入的右半部分都通过f运行。假设R_1和R_1'不同的比特在扩展步骤中没有被复制，那么在应用S-盒之前的中间值仍然只相差单个比特。根据S-盒的属性3，S-盒计算之后的中间值相差**至少两个**比特。结果是中间值(L_2, R_2)和(L_2', R_2')相差**三个**比特：L_2和L_2'之间有1比特差异（由R_1和R_1'之间的差异携带过来），以及R_2和R_2'之间的2比特差异。

混合置换传播了R_2和R_2'之间的两比特差异，使得在下一轮中，这两个比特的每一个都用作**不同**S-盒的输入，导致中间值右半部分的差异**至少为4个比特**。（如果R_2和R_2'不同的两个比特中的任何一个或两个都被E复制，差异可能会更大。）现在左半部分也有2比特差异。与代换-置换网络一样，我们有一个指数效应，因此在7轮之后，我们预计右半部分的所有32个比特都会受到影响（并且在8轮之后，左半部分的所有32个比特也会受到影响）。

DES有16轮，因此雪崩效应在计算的早期就发生了。这确保了DES对相似输入的计算产生看似独立的结果。

攻击减少轮数的DES

对于了解DES的构造及其安全性来说，一个有用的练习是查看只有几轮的DES的行为。我们将展示对一轮、两轮和三轮DES变体的攻击（回想一下，真正的DES有16轮）。三轮或更少轮的DES不能是伪随机函数，因为三轮不足以实现雪崩效应。因此，我们将关注演示更困难（和更有破坏性）的**密钥恢复攻击**，这些攻击仅使用少量使用该密钥计算的输入/输出对来计算密钥k。其中一些攻击类似于我们在代换-置换网络背景下看到的攻击；然而，在这里，我们将看到如何将它们应用于具体的分组密码，而不是抽象设计。

下面的攻击将是**已知明文攻击**，其中对手知道一些明文/密文对$\{(x_i, y_i)\}$，其中$y_i = \mathrm{DES}_k(x_i)$，对应于某个秘密密钥$k$。当我们描述攻击时，我们将重点关注一个特定的输入/输出对(x, y)，并描述对手可以从该对中导出的关于密钥的信息。继续使用前面开发的符号，我们将输入x的左半部分和右半部分分别表示为L_0和R_0，并让L_i, R_i表示第i轮之后的左半部分和右半部分。回想一下，E表示DES扩展函数，k_i表示在第i轮中使用的子密钥，而$f_i(R) = f(k_i, R)$表示在Feistel网络的第i轮中应用的实际函数。

一轮DES。假设我们得到了一个输入/输出对(x, y)。在一轮DES中，我们有$y = (L_1, R_1)$

，其中$L_1 = R_0$且$R_1 = L_0 \oplus f_1(R_0)$。因此我们知道$f_1$的一个输入/输出对；具体来说，我们知道$f_1(R_0) = R_1 \oplus L_0$。通过对输出$R_1 \oplus L_0$应用混合置换的逆，我们获得了包含所有S-盒输出的中间值，其中前4个比特是第一个S-盒的输出，接下来的4个比特是第二个S-盒的输出，依此类推。

考虑第一个S-盒的（已知）4比特输出。由于每个S-盒是一个4对1函数，这意味着恰好有四个可能的输入到这个S-盒会导致给定的输出，对于所有其他S-盒也是如此；每个这样的输入都是6比特长。S-盒的输入简单地是$E(R_0)$与子密钥k_1的XOR。由于R_0，因此$E(R_0)$是已知的，我们可以计算k_1的每个6比特部分的四组可能值。这意味着我们将可能的密钥k_1的数量从2^{48}减少到$4^{48/6} = 4^8 = 2^{16}$（因为k_1的八个6比特部分中的每一个有四种可能性）。这已经是一个小数目，因此我们可以尝试所有可能性，在不同的输入/输出对(x', y')上找到正确的密钥。因此，我们仅使用两个已知明文，在大约2^{16}时间内获得了密钥。

两轮DES。在两轮DES中，输出y等于(L_2, R_2)，其中

$$L_1 = R_0$$

$$R_1 = L_0 \oplus f_1(R_0)$$

$$L_2 = R_1 = L_0 \oplus f_1(R_0)$$

$$R_2 = L_1 \oplus f_2(R_1).$$

从给定的输入/输出对(x, y)中，L_0, R_0, L_2和R_2是已知的，因此我们也知道$L_1 = R_0$和$R_1 = L_2$。这意味着我们知道f_1和f_2的输入/输出，因此用于攻击一轮DES的相同方法可用于此，在大约$2 \cdot 2^{16}$时间内确定k_1和k_2。即使k_1和k_2是完全独立的密钥，这种攻击也有效，尽管事实上DES的密钥编排确保了k_1和k_2的许多比特是相等的（这可用于进一步加快攻击）。

三轮DES。参考图6.5，输出值y现在等于(L_3, R_3)。由于$L_1 = R_0$且$R_2 = L_3$，图中的唯一未知值是R_1和L_2（它们是相等的）。

现在我们不再有任何轮函数f_i的完整输入/输出。例如，f_2的输出值等于$L_1 \oplus R_2$，其中这两个值都是已知的。然而，我们不知道作为f_2输入的R_1的值。同样，我们可以确定f_1和f_3的输入，但不能确定这些函数的输出。因此，我们用于破解一轮和两轮DES的攻击在这里将无效。

与其依赖于对一个轮函数的输入和输出的完全了解，不如我们将利用f_1和f_3的输入和输出之间某个关系的知识。观察到f_1的输出等于$L_0 \oplus R_1 = L_0 \oplus L_2$，而$f_3$的输出等于$L_2 \oplus R_3$。因此，

$$f_1(R_0) \oplus f_3(R_2) = (L_0 \oplus L_2) \oplus (L_2 \oplus R_3) = L_0 \oplus R_3,$$

其中L_0和R_3都是已知的。也就是说，我们知道f_1和f_3的输出的XOR。此外，f_1的输入是R_0，而f_3的输入是L_3，两者都是已知的。我们得出结论，我们可以确定f_1和f_3的输入以及它们输出的XOR。我们现在描述一种基于此信息找到密钥的攻击。

回想一下，DES的密钥编排具有将主密钥划分为一个"左半部分"k_L和一个"右半部分"k_R的属性，每个包含28比特。此外，每轮中使用的子密钥的左侧24比特仅取自k_L，而每个子密钥的右侧24比特仅取自k_R。这意味着k_L仅影响任何轮中前四个S-盒的输入，而k_R仅影响最后四个S-盒的输入。由于混合置换是已知的，我们也知道每轮函数的输出中哪些比特来自每个S-盒。

攻击背后的想法是分别遍历主密钥的两半的密钥空间，从而给出一个复杂度大致为$2 \cdot 2^{28}$而不是2^{56}的攻击。如果我们能验证对主密钥的一半的猜测，这样的攻击就可能实现，我们现在展示如何做到这一点。假设我们猜测k_L的某个值，即主密钥的左半部分。我们知道f_1的输入R_0，因此使用我们对k_L的猜测，我们可以计算前四个S-盒的输入。这意味着我们可以计算f_1输出的一半比特（混合置换传播了我们知道的比特，但由于混合置换是已知的，我们确切地知道这些比特是什么）。同样，我们可以使用已知的输入L_3和对k_L的相同猜测来计算f_3输出中的相同位置。最后，我们可以计算这些输出值和f_1和f_3输出的XOR的已知值中的相应比特的XOR。如果它们不相等，那么我们对k_L的猜测是**不正确**的。对k_L的正确猜测将始终通过此测试，因此不会被消除，但**不正确**猜测预计仅以大致2^{-16}的概率通过此测试（因为我们在两个计算值中检查了16个比特的相等性）。k_L有2^{28}个可能值，因此如果每个不正确值以2^{-16}的概率保持为可行候选者，那么我们期望在上述步骤之后只剩下$2^{28} \cdot 2^{-16} = 2^{12}$个$k_L$的可能性。

通过对主密钥的每一半执行上述操作，我们在$2 \cdot 2^{28}$时间内分别获得了左半部分的大约2^{12}个候选者和右半部分的大约2^{12}个候选者。由于左半部分和右半部分的每种组合都是可能的，因此我们总共有2^{24}个候选密钥，并且可以使用额外的输入/输出对(x', y')对该集合运行穷举搜索。（另一种更有效的方法是简单地使用每个半密钥的2^{12}个剩余候选者重复先前的攻击。）该攻击的时间复杂度大致为$2 \cdot 2^{28} + 2^{24} < 2^{30}$，远小于$2^{56}$。

DES的安全性

经过近30年的深入研究，DES已知的最佳**实际**攻击仍然是对其密钥空间的穷举搜索。（我们在第6.2.6节中讨论一些重要的理论攻击。这些攻击需要大量的输入/输出对，这在任何使用DES的现实世界系统上的攻击中都很难获得。）不幸的是，DES的56比特密钥长度太短，以至于对所有2^{56}个可能密钥的穷举搜索现在是可行的。早在1970年代后期，就有人强烈反对选择如此短的DES密钥。当时，这种反对是理论上的，因为搜索那么多密钥所需的计算能力通常是不可用的。然而，DES上穷举搜索攻击的实用性在1997年得到了证明，当时RSA Security设置的一

系列DES挑战中的第一个由DESCHALL项目使用数千台通过互联网协调的计算机解决；计算耗时96天。第二年，distributed.net项目在短短41天内打破了第二个挑战。1998年出现了一个重大突破，第三个挑战在短短56小时内被解决。这一令人印象深刻的壮举是通过一个名为Deep Crack的专用DES破解机器实现的，该机器由电子前沿基金会（Electronic Frontier Foundation）以25万美元的成本建造。1999年，Deep Crack和distributed.net联合努力，在22小时多一点的时间内解决了一个DES挑战。当前的最新技术是PICO Computing的DES破解盒，它使用48个FPGA，可以在大约23小时内找到一个DES密钥。

第5.4.3节中讨论的时间/空间权衡表明，可以使用预计算和额外内存来加速穷举密钥搜索攻击。由于DES的密钥长度短，时间/空间权衡可能特别有效。具体来说，使用预处理，可以生成一个几兆字节大的表，然后使用大约2^{38}次DES评估（可以在几分钟内计算）从单个输入/输出对中高概率恢复DES密钥。底线是DES的密钥太短，不应被视为适用于任何严肃应用的安全。

第二个令人担忧的原因是DES相对较短的分组长度。分组长度短是有问题的，因为许多基于分组密码的构造的具体安全性取决于分组长度——即使使用的密码是“完美”的。例如，CTR模式的安全性证明（参见定理3.32）表明，即使使用一个完全随机的函数，如果攻击者获得了q个明文/密文对，他也可以以概率$2q^2/2^l$打破该加密方案的安全性。在DES的情况下，其中$l = 64$，这意味着如果攻击者仅获得$q = 2^{30}$个明文/密文对，安全性就会以高概率受到威胁。如果对手窃听了包含已知头部、冗余等内容的加密消息，获取明文/密文对是相对容易的。

DES的不安全性与它的设计本身无关，而是由于其短密钥长度（以及在较小程度上，其短分组长度）。这对DES的设计者来说是一个巨大的赞扬，他们似乎成功地构造了一个几乎“完美”的分组密码（除了其密钥太短）。由于DES本身似乎没有显著的结构弱点，因此将DES用作构建具有更长密钥的分组密码的构建块是有意义的。我们将在第6.2.4节中进一步讨论这一点。

DES的替代品——**高级加密标准**（AES，将在本章稍后介绍）——被明确设计用于解决对DES的短密钥长度和分组长度的担忧。AES支持128、192或256比特密钥，分组长度为128比特。

对DES的优于穷举搜索的攻击最早由Biham和Shamir在1990年代初提出，他们开发了一种称为**差分密码分析**的技术。他们的攻击耗时2^{37}，需要2^{47}个选择明文。虽然从理论角度来看，这次攻击是一个突破，但在一个对手可以获得如此多加密的选择明文的现实场景中，它似乎没有太多的实际意义。

有趣的是，Biham和Shamir的工作表明，DES S-盒被专门设计用于抵抗差分密码分析，这表明DES的设计者（尽管没有公开透露）知道差分密码分析技术。在Biham和Shamir宣布他们的结果后，这一怀疑得到了证实。

线性密码分析由Matsui在1990年代中期开发，并成功应用于DES。这种攻击的优势在于它使用

已知明文而不是选择明文。尽管如此，所需的明文/密文对的数量——大约2^{43}——仍然是巨大的。

我们将在第6.2.6节中简要描述差分和线性密码分析。

6.2.4 3DES：增加分组密码的密钥长度

DES的主要弱点是其短密钥。因此，尝试使用DES作为构建块设计一个具有更长密钥长度的分组密码是有意义的。本节讨论了实现这一目标的一些方法。虽然我们在讨论中经常提到DES，并且DES是这些技术应用的最突出的分组密码，但我们在这里所说的一切都普遍适用于任何分组密码。

内部修改与“黑盒”构造。构建另一个基于DES的密码，可以采取两种通用方法。第一种方法是某种程度上修改DES的内部结构，同时增加密钥长度。例如，可以保持轮函数不变，仅使用128比特主密钥和不同的密钥编排（每轮仍然选择一个48比特子密钥）。或者，可以更改S-盒本身，并在每轮中使用更大的子密钥。这些方法的缺点是，通过修改DES——即使是以最小的方式——我们失去了对DES的信心，因为DES多年来一直抵抗攻击。密码构造非常敏感，即使是温和、看似微不足道的更改也可能使构造完全不安全。因此，**不推荐**更改分组密码的内部结构。

另一种不遭受上述问题的方法是，将DES用作一个“黑盒”，完全不触及其内部结构。在这种方法中，我们将DES视为一个具有56比特密钥的“完美”分组密码，并构造一个仅调用原始、未修改DES的新分组密码。由于DES本身没有被篡改，这是一种更谨慎的方法，也是我们在这里将采用的方法。

双重加密

令F是一个具有n比特密钥长度和l比特分组长度的分组密码。然后可以定义一个具有$2n$长度密钥的新分组密码F'，定义为

$$F'_{k_1,k_2}(x) \stackrel{\text{def}}{=} F_{k_2}(F_{k_1}(x)),$$

其中k_1和k_2是独立的密钥。对于F是DES的情况，我们得到了一个名为**2DES**的密码F'，它接受一个112比特密钥；如果穷举密钥搜索是最佳可用攻击，那么112比特的密钥长度将是足够的，因为需要2^{112}时间的攻击是完全无法实现的。不幸的是，我们现在展示一个针对F'的攻击，它在大约2^n时间内运行，远少于人们希望的穷举搜索$2n$比特密钥所需的2^{2n}时间。这意味着新分组密码的安全性基本上不比旧的好，即使它的密钥长了两倍。

这种攻击被称为“**中途相遇攻击**”（meet-in-the-middle attack），原因很快就会清楚。假设

对手获得了单个输入/输出对(x, y)，其中$y = F'_{k_1^*,k_2^*}(x) = F_{k_2^*}(F_{k_1^*}(x))$，对应于未知密钥$k_1^*, k_2^*$。对手可以通过以下方式缩小可能密钥的集合：

1. 对于每个$k_1 \in \{0,1\}^n$，计算$z := F_{k_1}(x)$并将(z, k_1)存储在列表L中。
2. 对于每个$k_2 \in \{0,1\}^n$，计算$z := F_{k_2}^{-1}(y)$并将(z, k_2)存储在列表L'中。
3. 如果$z_1 = z_2$，则列表L中的条目(z_1, k_1)和列表L'中的条目(z_2, k_2)是**匹配**。对于每个这样的匹配，将(k_1, k_2)添加到集合S中。（通过按第一个组件排序L和L'，可以轻松找到匹配项。）

参见图6.7的图形描述。

该攻击耗时$O(n \cdot 2^n)$，需要空间$O((n+l) \cdot 2^n)$。该算法输出的集合S恰好包含那些满足以下等式的(k_1, k_2)值

$$F_{k_1}(x) = F_{k_2}^{-1}(y) \quad (6.4)$$

或者等价地，满足$y = F_{k_2}(F_{k_1}(x))$的(k_1, k_2)。特别是，$(k_1^*, k_2^*) \in S$。另一方面，如果我们将$F_{k_1}(x)$和$F_{k_2}^{-1}(y)$视为均匀l比特字符串，那么 heuristically 期望一个对$(k_1, k_2) \neq (k_1^*, k_2^*)$以概率$2^{-l}$满足等式（6.4）。因此，集合$S$的期望大小是$2^{2n} \cdot 2^{-l} = 2^{2n-l}$。使用另外少量的输入/输出对，并取所获得的集合的交集，可以高概率地识别出正确的(k_1^*, k_2^*)。

图6.7：中途相遇攻击。

三重加密

前述方法的明显推广是连续应用分组密码**三次**。这种方法有两种常见变体：

变体1：三密钥。选择三个独立密钥k_1, k_2, k_3并定义

$$F'_{k_1,k_2,k_3}(x) \stackrel{\text{def}}{=} F_{k_3}(F_{k_2}^{-1}(F_{k_1}(x))).$$

变体2：两密钥。选择两个独立密钥k_1, k_2然后定义

$$F'_{k_1,k_2}(x) \stackrel{\text{def}}{=} F_{k_1}(F_{k_2}^{-1}(F_{k_1}(x))).$$

在比较这两种替代方案的安全性之前，我们注意到F的中间调用是**逆向**的。如果F是一个足够好的密码，这对于安全性来说没有区别，因为如果F是一个强大的伪随机置换，那么F^{-1}也必须是。逆转F的第二次应用的原因是为了获得**向后兼容性**：如果设置$k_1 = k_2 = k_3$，则生成的函数等同于使用密钥k_1的F的单次调用。

第一个变体的安全性。第一个变体的密钥长度是$3n$，因此我们可能希望针对该密码的最佳攻击需要2^{3n}时间。然而，该密码与双重加密的情况一样容易受到中途相遇攻击，尽管这次攻击耗时2^{2n}。这是已知最好的攻击。因此，尽管此变体的安全性不如人们希望的那么高，但它获得了足够的安全性，对于所有实际目的都是如此，即使对于$n = 56$也是如此（当然，假设原始密码F没有弱点）。

第二个变体的安全性。该变体的密钥长度是$2n$，因此我们能希望的最好结果是抵抗运行时间为2^{2n}的攻击。当对手只获得了少量输入/输出对时，没有已知的攻击具有更好的时间复杂度。（参见练习6.13，它使用2^n个选择明文进行攻击。）因此，两密钥三重加密在实践中是一个合理的选择。

三重DES（3DES）。三重DES（或3DES）是基于DES的三重调用，使用两个或三个密钥，如上所述。3DES于1999年标准化，并于今天被广泛使用。它的主要缺点是其相对较小的分组长度以及它相对较慢，因为它需要3个完整的分组密码操作。由于现在推荐的最小密钥长度是128比特，2密钥3DES不再推荐（由于其密钥长度仅为112比特）。这些缺点导致了DES/三重DES被下一节介绍的高级加密标准所取代。

6.2.5 AES – 高级加密标准

1997年1月，美国国家标准与技术研究院（NIST）宣布将举行一场竞赛，以选择一个新的分组密码——称为**高级加密标准**（AES）——来取代DES。竞赛开始时公开征集团队提交候选分组密码进行评估。来自世界各地的共提交了15种不同的算法，其中包括来自许多最优秀的密码学家和密码分析专家的贡献。每个团队的候选密码都受到了NIST成员、公众（尤其是）其他团队的密集分析。举行了两次研讨会，一次在1998年，一次在1999年，讨论和分析各种提交的方案。在第二次研讨会之后，NIST将范围缩小到5个“决赛入围者”，并开始了第二轮竞赛。2000年4月举行了第三次AES研讨会，邀请对这五个决赛入围者进行额外审查。2000年10月，NIST宣布获胜算法是**Rijndael**（由比利时密码学家Vincent Rijmen和Joan Daemen设计的分组密码），尽管NIST承认这5个决赛入围者中的任何一个都会是一个很好的选择。特别是，在这5个决赛入围者中没有发现任何严重的安全漏洞，选择“获胜者”部分是基于效率、硬件性能、灵活性等属性。

选择AES的过程是巧妙的，因为任何提交算法并因此有兴趣使其算法被采纳的团体，都有强烈的动机去寻找对其他提交方案的攻击。通过这种方式，世界上最优秀的密码分析专家将注意力集中在发现提交给竞赛的候选密码中的哪怕是**最轻微**的弱点上。经过短短几年，每个候选算法都受到了密集研究，从而增加了我们对获胜算法安全性的信心。当然，算法使用和研究的时间越长而没有被破解，我们的信心就会继续增长。今天，AES被广泛使用，并且尚未发现任何显著的安全弱点。

AES构造。在本节中，我们介绍Rijndael/AES的高级结构。（从技术上讲，Rijndael和AES不是同一件事，但差异对于我们这里的讨论并不重要。）与DES一样，我们不会提供完整规范，我们的描述不应被用作实现的基础。我们的目标只是提供算法工作原理的一般概念。

AES分组密码具有128比特的分组长度，可以使用128、192或256比特密钥。密钥的长度影响密钥编排（即每轮中使用的子密钥）以及轮数，但不影响每轮的高级结构。

与使用Feistel结构的DES相比，AES本质上是一个**代换-置换网络**。在AES算法的计算过程中，一个称为**状态**的4x4字节数组在一系列轮中被修改。状态最初设置为等于密码的输入（注意输入是128比特，恰好是16字节）。然后在每轮中，以下操作以四个阶段应用于状态：

阶段1 – AddRoundKey： 在AES的每轮中，从主密钥派生出一个128比特的子密钥，并将其解释为4x4的字节数组。状态数组通过将其与该子密钥进行XOR来更新。

阶段2 – SubBytes： 在此步骤中，状态数组的每个字节都根据单个固定的查找表S替换为另一个字节。该代换表（或S-盒）是$\{0,1\}^8$上的一个双射。

阶段3 – ShiftRows： 在此步骤中，状态数组的每一行中的字节向左循环移动如下：数组的第一行保持不变，第二行向左移动一个位置，第三行向左移动两个位置，第四行向左移动三个位置。所有移位都是循环的，例如，在第二行中，第一个字节变为第四个字节。

阶段4 – MixColumns： 在此步骤中，对每列的四个字节应用可逆变换。（从技术上讲，这是一种线性变换——即在适当域上的矩阵乘法。）该变换具有以下特性：如果两个输入在$b > 0$个字节上不同，则应用该变换产生的两个输出至少在$5 - b$个字节上不同。

在最后一轮中，MixColumns被AddRoundKey取代。这阻止了攻击者简单地反转最后三个不依赖于密钥的阶段。

通过将阶段3和4一起视为一个“混合”步骤，我们看到AES的每一轮都具有代换-置换网络的结构：轮子密钥首先与当前轮的输入进行XOR；接下来，一个小的可逆函数应用于所得值的“块”；最后，将结果的比特混合以获得扩散。唯一的区别是，与我们前面描述的代换-置换网络不同，这里的混合步骤不包括简单的比特洗牌，而是通过洗牌加上可逆线性变换来执行。（稍微简化一下，看一个琐碎的3比特示例，将$x = x_1 \| x_2 \| x_3$的比特洗牌可能将x映射到$x' = x_2 \| x_1 \| x_3$。可逆线性变换可能将x映射到$x_1 \oplus x_2 \| x_2 \oplus x_3 \| x_1 \oplus x_2 \oplus x_3$。）

轮数取决于密钥长度。对于128比特密钥使用10轮，对于192比特密钥使用12轮，对于256比特密钥使用14轮。

AES的安全性。正如我们所提到的，AES密码在选择过程中受到了密集的审查，并一直在被研

究。迄今为止，还没有针对完整密钥的穷举搜索的实际密码分析攻击。

我们得出结论，截至今天，AES是任何需要（强）伪随机置换的密码方案的绝佳选择。它是免费的、标准化的、高效的、高度安全的。

6.2.6 *差分和线性密码分析

分组密码相对复杂，因此很难分析。然而，人们不应该被愚弄，认为复杂的密码就难以破解。相反，构造一个安全的块密码非常困难，而对大多数构造发现攻击却出奇地容易（无论它们看起来多么复杂）。这应该作为一个警告，非专业人士不应该尝试构造新的密码。鉴于三重DES和AES的可用性，很难证明使用其他任何东西是合理的。

在本节中，我们描述了现在作为密码分析专家工具箱的标准部分的两种工具。我们在这里的目标是提供一些高级密码分析的**品味**，并强调设计安全分组密码涉及对其组件进行仔细选择的想法。

差分密码分析。这种技术可以导致对分组密码的选择明文攻击，最早由Biham和Shamir在1980年代末提出，他们用它在1993年攻击了DES。该攻击的基本思想是列表化输入中的特定差异，这些差异导致输出中的特定差异，其概率大于随机置换所期望的概率。具体来说，假设差分(Δ_x, Δ_y)以概率p出现在某个带密钥置换F'中，如果对于均匀输入x_1和x_2满足$x_1 \oplus x_2 = \Delta_x$，并且均匀选择密钥$k$，那么$F_k(x_1) \oplus F_k(x_2) = \Delta_y$的概率是$p$。对于任何固定的$(\Delta_x, \Delta_y)$和满足$x_1 \oplus x_2 = \Delta_x$的$x_1, x_2$，如果我们选择一个均匀函数$f : \{0,1\}^l \to \{0,1\}^l$，我们有$\Pr[f(x_1) \oplus f(x_2) = \Delta_y] = 2^{-l}$。然而，在一个弱分组密码中，可能存在以显著更高概率发生的差分。这可以被利用来给出完整的密钥恢复攻击，正如我们现在为SPN所示。

我们描述基本思想，然后通过一个具体的例子进行说明。令F是一个r轮SPN的分组密码，分组长度为l比特，并让$F'_k(x)$表示$F_k(x)$计算中应用第r轮的密钥混合步骤之后的中间结果。（也就是说，F'排除了最后一轮的S-盒代换和混合置换，以及最终的密钥混合步骤。）假设在F'中存在以概率$p > 2^{-l}$发生的差分(Δ_x, Δ_y)。可以利用这种高概率差分来学习最终混合子密钥k_{r+1}的比特。高级思想如下：令$\{(x_i^1, x_i^2)\}_{i=1}^L$是一组具有差分$\Delta_x$的$L$对随机输入，即$x_i^1 \oplus x_i^2 = \Delta_x$对于所有$i$成立。使用选择明文攻击，获取值$y_i^1 = F_k(x_i^1)$和$y_i^2 = F_k(x_i^2)$对于所有$i$。现在，对于所有可能的比特串$k^* \in \{0,1\}^l$，假设最终子密钥$k_{r+1}$的值是$k^*$，计算$\tilde{y}_i^1 = F'_k(x_i^1)$和$\tilde{y}_i^2 = F'_k(x_i^2)$。这是通过使用$k^*$反转最终密钥混合步骤，然后反转第$r$轮的混合置换和S-盒来完成的，这些不依赖于主密钥。当$k^* = k_{r+1}$时，我们期望p分数对将满足$\tilde{y}_i^1 \oplus \tilde{y}_i^2 = \Delta_y$。另一方面，当$k^* \neq k_{r+1}$时，我们启发式地期望只有$2^{-l}$分数对会产生这个差分。通过设置足够大的$L$，可以确定最终子密钥$k_{r+1}$的正确值。

这可行，但效率不高，因为在每个步骤中我们枚举了2^l个可能值。我们可以通过一次猜测k_{r+1}的部分做得更好。更具体地说，假设F中的S-盒具有1字节输入/输出长度，并关注Δ_y的第一个

字节，我们假设它非零。通过猜测k_{r+1}的仅8个比特，即那些对应于（在第r轮混合置换之后）第一个S-盒输出的8个比特，可以验证该字节中是否存在差分。因此，像上面那样进行，我们可以通过枚举这些比特的所有可能值来学习这8个比特，并查看哪个值以最高的概率产生第一个字节中所需的差分。对于那些8个比特的不正确猜测，产生的预期差分在该字节中的概率是2^{-8}，但正确猜测将以大致$p + 2^{-8}$的概率给出预期的差分；这是因为以概率p差分在整个分组上成立（因此特别是对于第一个字节），而当这种情况不成立时，我们可以将第一个字节中的差分视为随机的。请注意，可能需要不同的差分来学习k_{r+1}的不同部分。

在实践中，会执行各种优化以提高上述测试的有效性，或者更具体地说，增加不正确猜测产生差分的概率与正确猜测产生差分的概率之间的差距。一种优化是使用**低权重差分**，其中Δ_y在进入第r轮S-盒的位置中有许多零字节。满足这种差分的任何对y_1, y_2在进入第r轮的许多S-盒中将具有相等的值，因此将在相应的比特位置产生相等的输出值y_1, y_2（取决于最终混合置换）。这意味着在执行前面描述的测试时，可以简单地丢弃那些在那些比特位置不一致的对$\left(y_i^1, y_i^2\right)$（因为相应的中间值$\left(\tilde{y}_i^1, \tilde{y}_i^2\right)$不可能满足差分，对于最终子密钥的任何选择）。这显著提高了攻击的有效性。

一旦k_{r+1}已知，攻击者可以“剥离”最终的密钥混合步骤，以及第r轮的混合置换和S-盒代换步骤（因为这些不依赖于主密钥），然后应用相同的攻击——使用不同的差分——来找到第r轮子密钥k_r，依此类推，直到学到所有子密钥（或等效地，整个主密钥）。

一个工作示例。我们通过一个“玩具”示例进行说明，也说明如何找到一个好的差分。我们使用一个四轮SPN，分组长度为16比特，基于一个具有4比特输入/输出长度的单个S-盒。S-盒定义如下（表格显示了每个4比特输入如何映射到4比特输出）：

输入:	**0000**	**0001**	**0010**	**0011**	**0100**	**0101**	**0110**	**0111**
输出:	0000	1011	0101	0001	0110	1000	1101	0100

输入:	**1000**	**1001**	**1010**	**1011**	**1100**	**1101**	**1110**	**1111**
输出:	1111	0111	0010	1100	1001	0011	1110	1010

混合置换显示了16比特块中的每个比特移动的位置，如下所示：

In:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Out:	7	2	3	8	12	5	11	9	10	1	14	13	4	6	16

图6.8：输入差异$\Delta_x = 1111$在我们S-盒中的影响。

我们首先在S-盒中找到一个差分。令$S(x)$表示S-盒在输入x上的输出。考虑差分$\Delta_x = 1111$。那么，例如，我们有$S(0000) \oplus S(1111) = 0000 \oplus 1010 = 1010$，因此在这种情况下，输入中的1111差异导致输出中的1010差异。让我们看看这种关系是否经常成立。我们有$S(0001) = 1011$和$S(0001 \oplus 1111) = S(1110) = 1110$，因此在这里输入中的1111差异不会导致输出中的1010差异。然而，$S(0100) = 0110$和$S(0100 \oplus 1111) = S(1011) = 1100$，因此在这种情况下，输入中的1111差异导致输出中的1010差异。在图6.8中，我们列出了所有可能输入的表格结果。我们看到**一半**的时间输入中的1111差异导致输出中的1010差异。因此，$(1111, 1010)$是S中以概率$1/2$发生的差分。

图6.9：我们S-盒中的差分。

同样的过程序列可以用于所有2^4个输入差异Δ_x来计算每个差分的概率。即，对于每对(Δ_x, Δ_y)，我们列表化了4比特输入x，使得$S(x) \oplus S(x \oplus \Delta_x) = \Delta_y$的数量。我们在图6.9中为我们的示例S-盒完成了此操作。（为简洁起见，我们使用十六进制表示法表示(Δ_x, Δ_y)。表格应如下阅读：条目(i, j)计数有多少输入具有差异i，映射到差异j的输出。例如，观察到有8个输入具有差异$0\text{xF} = 1111$，映射到输出$0\text{xA} = 1010$，正如我们上面所显示的。这是最高概率的差分（除了微不足道的差分$(0, 0)$）。但还有其他感兴趣的差分：输入差异$0\text{x}4 = 0100$映射到输出差异$0\text{x}6 = 0110$的概率为$6/16 = 3/8$，并且有几个差分以概率$4/16 = 1/4$发生。

现在我们将此扩展到找到SPN前三轮的一个好的差分。考虑评估SPN在两个输入上，它们具有差分$0000\ 1100\ 0000\ 0000$，并追踪该评估每一步的中间值之间的差分。（参考图6.10，它显示了SPN的四轮加上最终的密钥混合步骤。为了清晰起见，图中省略了第4轮中的混合置换；该混合置换只会洗牌差分的比特，因此在攻击中可以轻松考虑。）第一轮中的密钥混合步骤不影响差分，因此第一轮中第二个S-盒的输入具有差分1100。我们从图6.9中看到，S-盒输入中的差异$0\text{xC} = 1100$导致S-盒输出中的差异$0\text{x}8 = 1000$的概率是$1/4$。因此，以概率$1/4$，第1轮后第2个S-盒输出中的差分是单个比特，该比特通过混合置换从第5个位置移动到第12个位置。（其他S-盒的输入是相等的，因此它们的输出是相等的，输出的差分是0000。）假设情况如此，第二轮中第三个S-盒的输入差分是$0\text{x}1 = 0001$（再次，第二轮中的密钥混合步骤不影响差分）；使用图6.9，我们有概率$1/4$，该S-盒的输出差分是$0\text{x}4 = 0100$。因此，再次只有一个不同的输出比特，并且它通过混合置换从第10个位置移动到第1个位置。最后，再次查阅图6.9，我们看到S-盒的输入差分$0\text{x}8 = 1000$导致输出差分$0\text{xF} = 1111$的概率是$1/4$。然后位置1、2、3和4的比特通过混合置换移动到位置7、2、3和8。

图6.10：追踪通过使用本文中给出的S-盒和混合置换的四轮SPN的差分。

总而言之，我们看到输入差分$\Delta_x = 0000\ 1100\ 0000\ 0000$在三轮后产生输出差分$\Delta_y = 0110\ 0011\ 0000\ 0000$的概率**至少**是$\frac{1}{4} \cdot \frac{1}{4} \cdot \frac{1}{4} = \frac{1}{64}$。（我们乘以概率，因为我们启发式地假设每轮是独立的。）对于随机函数，任何给定差分发生的概率仅为$2^{-16} = 1/65536$。因此，我们发现的差分发生的概率显著高于随机函数所期望的概率。还要观察到，我们发现了一个**低权重**差分。

我们可以使用这个差分来找到最终子密钥k_5的前8个比特。如前所述，我们开始于令$\{(x_i^1, x_i^2)\}_{i=1}^L$是一组具有差分$\Delta_x$的$L$对随机输入。使用选择明文攻击，然后我们获取值$y_i^1 = F_k(x_i^1)$和$y_i^2 = F_k(x_i^2)$对于所有$i$。现在，对于$k_5$的初始8个比特的所有可能值，我们计算$\tilde{y}_i^1, \tilde{y}_i^2$的初始8个比特，即第4轮密钥混合步骤之后的中间值。（我们可以这样做，因为我们只需要反转第4轮的两个最左边的S-盒来推导那8个比特。）当我们猜中k_5的初始8个比特的正确值时，我们期望8比特差分$0110\ 0011$以概率至少$1/64$发生。启发式地，不正确的猜测只以$2^{-8} = 1/256$的概率产生预期的差分。通过设置足够大的L，我们可以（以高概率）识别出正确的值。

差分攻击在实践中。差分密码分析非常强大，并已被用于攻击真实的密码。一个突出的例子是FEAL-8，它于1987年被提议作为DES的替代品。发现对FEAL-8的差分攻击仅需要1,000个选择明文。1991年，使用这种攻击找到整个密钥耗时不到2分钟。今天，任何提议的密码都经过差分密码分析的抵抗性测试。

差分攻击是第一个对DES的攻击，需要比简单穷举搜索更少的时间。虽然这是一个有趣的理论结果，但由于它需要2^{47}个选择明文，因此在实践中并不构成重大担忧。在大多数现实世界应用中，攻击者很难获得如此多的选择明文/密文对。有趣的是，对DES的S-盒进行微小修改会使密码更容易受到差分攻击。DES设计者的个人证词（在差分攻击在外部世界被发现之后）已证实DES的S-盒被专门设计用于阻止差分攻击。

线性密码分析。线性密码分析由Matsui在1990年代初开发。我们将只在高层次上描述该技术。基本思想是考虑输入和输出之间以高于随机函数所期望的概率成立的线性关系。更详细地说，假设比特位置$i_1, \ldots, i_{\text{in}}$和$i'_1, \ldots, i'_{\text{out}}$具有线性偏差$\varepsilon$，如果对于均匀$x$和$k$，以及$y \stackrel{\text{def}}{=} F_k(x)$，成立

$$\left| \Pr[x_{i_1} \oplus \cdots \oplus x_{i_{\text{in}}} \oplus y_{i'_1} \oplus \cdots \oplus y_{i'_{\text{out}}} = 0] - \frac{1}{2} \right| = \varepsilon,$$

其中x_i, y_i表示x和y的第i个比特。对于随机函数和任何固定的比特位置集合，我们期望偏差接近于0。Matsui展示了如何使用密码中足够大的偏差来找到密钥。除了给出另一种攻击密码的方法之外，这种攻击的一个重要特征是它**不需要**选择明文，而是**已知明文**就足够了。这非常重要，

因为加密文件可以提供大量的已知明文，而获取选择明文的加密要困难得多。Matsui表明DES可以用仅2^{43}个已知明文/密文对被破解。

对分组密码设计的影响。现代分组密码的设计和评估部分基于它们对差分和线性密码分析的抵抗性。在构造分组密码时，设计者选择S-盒和其他组件，以最小化差分概率和线性偏差。我们注意到，不可能消除S-盒中的所有高概率差分：任何S-盒都会有一些差分比其他的发生更频繁。尽管如此，这些偏差可以最小化。此外，增加轮数（并仔细选择混合置换）可以减少差分概率，并使密码分析专家更难找到可利用的差分。

6.3 哈希函数

回想第5章，哈希函数H的主要安全要求是**抗碰撞性**；也就是说，应该很难找到碰撞，即不同的输入x, x'使得$H(x) = H(x')$。（我们在这里省略对任何密钥的提及，因为现实世界中的哈希函数通常是无密钥的。）如果哈希函数具有l比特的输出长度，那么我们能希望的最好结果是，使用远少于$2^{l/2}$次H的调用来找到碰撞是不可行的。（参见第5.4.1节。）我们还希望哈希函数实现（第二）**原像抵抗**，以抵抗运行时间远少于2^l的攻击，尽管我们在此讨论中不考虑此类攻击。

哈希函数通常分两步构造。首先，设计一个**压缩函数**（即固定长度的哈希函数）h；接下来，使用某种机制扩展h以处理任意输入长度。在第5.2节中，我们已经展示了用于第二步的一种方法——**Merkle-Damgård变换**。在这里，我们探索设计底层压缩函数的技术。我们还将讨论实践中使用的一些哈希函数。第8.4.2节给出了基于数论假设的压缩函数的理论构造。

6.3.1 分组密码的哈希函数

也许令人惊讶的是，可以从满足某些附加属性的分组密码构建**抗碰撞**压缩函数。有几种方法可以做到这一点；最常见的方法之一是通过**Davies-Meyer构造**。令F是一个具有n比特密钥长度和l比特分组长度的分组密码。然后我们可以通过$h(k, x) \stackrel{\text{def}}{=} F_k(x) \oplus x$定义压缩函数$h: \{0,1\}^{n+l} \to \{0,1\}^l$。（参见图6.11。）

图6.11：Davies-Meyer构造。

我们不知道如何仅基于F是强大的伪随机置换的假设来证明所得压缩函数的抗碰撞性，事实上，有理由相信这样的证明是**不可能**的。但是，如果我们将F建模为**理想密码**，我们可以证明抗碰撞性。**理想密码模型**是对随机预言模型的加强（参见第5.5节），其中我们假设所有参与方都可以访问随机带密钥置换$F: \{0,1\}^n \times \{0,1\}^l \to \{0,1\}^l$及其逆$F^{-1}$的预言机（即，使得$F^{-1}(k, F(k, x)) = x$对于所有$k, x$成立）。另一种思考方式是，每个密钥$k \in \{0,1\}^n$指定了

l比特字符串上的一个独立、均匀置换$F(k,\cdot)$。与随机预言模型一样，计算F（或F^{-1}）的唯一方法是明确地查询带有(k,x)的预言机并接收$F(k,x)$（或$F^{-1}(k,x)$）。

在理想密码模型中分析构造具有在随机预言模型中工作的所有优点和缺点，正如在第5.5节中详细讨论的那样。我们在这里只补充一点，理想密码模型意味着**不存在相关密钥攻击**，即（正如我们刚才所说）置换$F(k,\cdot)$和$F(k',\cdot)$必须表现出独立性，即使例如k和k'仅在一个比特上不同。此外，不能存在"弱密钥"k（例如，全0密钥），使得$F(k,\cdot)$容易与随机区分开来。它还意味着，即使k已知，$F(k,\cdot)$也应该"随机行为"。对于任何现实世界密码F，即使F是强大的伪随机置换，这些属性也**不一定**成立（甚至没有明确定义），读者可能会注意到，我们在任何现实世界分组密码构造的分析中都没有讨论这些属性。（事实上，DES和三重DES不满足这些属性。）用于实例化理想密码的任何分组密码都必须根据这些更严格的要求进行评估。

我们在具体设置中证明以下定理，但可以很容易地将其证明适用于渐近设置。

定理6.5 如果F被建模为**理想密码**，那么Davies-Meyer构造产生了**抗碰撞**压缩函数。具体来说，任何对理想密码预言机进行$q < 2^{l/2}$次查询的攻击者，找到碰撞的概率**至多**为$q^2/2^l$。

证明 为了清楚起见，我们在这里考虑概率实验，其中F是随机采样的（更准确地说，对于每个$k \in \{0,1\}^n$，函数$F(k,\cdot):\{0,1\}^l \to \{0,1\}^l$是从$l$比特字符串上的置换集合$\mathrm{Perm}_l$中均匀选择的），然后攻击者被赋予对$F$和$F^{-1}$的预言机访问权限。攻击者然后尝试找到一个碰撞对$(k,x),(k',x')$，即满足$F(k,x)\oplus x = F(k',x')\oplus x'$。除了限制攻击者进行的预言机查询数量外，不对攻击者施加任何计算限制。我们假设如果攻击者输出一个碰撞对$(k,x),(k',x')$，那么她以前已经进行了计算值$F(k,x)$和$F(k',x')$所需的预言机查询。我们还假设攻击者从不查询同一个预言机多次，并且一旦学到了$y=F(k,x)$（反之亦然），就从不查询$F^{-1}(k,y)$。所有这些假设都没有损失一般性。

考虑攻击者对其预言机进行的第i次查询。对F的查询(k_i,x_i)仅揭示哈希值$h_i \stackrel{\text{def}}{=} h(k_i,x_i) = F(k_i,x_i)\oplus x_i$；类似地，对$F^{-1}$的查询，给出结果$x_i = F^{-1}(k_i,y_i)$，仅揭示哈希值$h_i \stackrel{\text{def}}{=} h(k_i,x_i) = y_i \oplus F^{-1}(k_i,y_i)$。只有当$h_i = h_j$对于某些$i \neq j$时，攻击者才会获得碰撞。

固定i,j且$i>j$，并考虑$h_i = h_j$的概率。在第i次查询时，h_j的值是固定的。仅当攻击者查询F的(k_i,x_i)并获得结果$F(k_i,x_i) = h_j \oplus x_i$，或者查询$F^{-1}$的$(k_i,y_i)$并获得结果$F^{-1}(k_i,y_i) = h_j \oplus y_i$时，才会在第$i$次查询上获得$h_i$和$h_j$之间的碰撞。任一事件发生的概率至多为$1/(2^l-(i-1))$，因为例如$F(k_i,x_i)$在$\{0,1\}^l$上是均匀的，除了它不能等于攻击者已经使用密钥$k_i$定义的前$(i-1)$个预言机查询的任何值。由于$i \le q < 2^{l/2}$，因此$h_i = h_j$的概率至多为$2/2^l$。

对所有$\binom{q}{2} < q^2/2$个不同的对i, j进行联合界限，得到定理中陈述的结果。

Davies-Meyer和DES。正如我们上面提到的，在用任何具体分组密码实例化Davies-Meyer构造时，必须小心，因为该密码必须满足附加属性（超出了强大的伪随机置换），以使所得构造安全。在练习6.21中，我们探讨了当在Davies-Meyer构造中使用DES时会出错的地方。

这应该作为一个警告，即Davies-Meyer构造在理想密码模型中的安全证明不一定转化为用真实密码实例化时的真实安全性。尽管如此，正如我们将在下面描述的，这种范式已被用于构造已抵抗攻击的实用哈希函数（但特别是在构造中心的块密码是专门为此目的设计的）。

总之，Davies-Meyer构造是构造抗碰撞压缩函数的有用范式。然而，**不应**将其应用于为加密而设计的分组密码，如DES和AES。

6.3.2 MD5

MD5是一个输出长度为128比特的哈希函数。它于1991年设计，并在一段时间内被认为是抗碰撞的。多年来，在MD5中开始发现各种弱点，但这些弱点似乎没有导致任何容易找到碰撞的方法。令人震惊的是，2004年，一个中国密码分析专家团队提出了一种在MD5中找到碰撞的新方法；他们通过展示一个**明确的碰撞**很容易说服其他人他们的方法是正确的！从那时起，攻击得到了改进，今天可以在台式PC上不到一分钟内找到碰撞。此外，攻击已被扩展到甚至可以找到“受控碰撞”（例如，生成任意可见内容的两个postscript文件）。

由于这些攻击，在需要密码安全性的任何地方都不应使用MD5。我们提到MD5只是因为它仍然出现在遗留代码中。

6.3.3 SHA-0、SHA-1和SHA-2

安全哈希算法（Secure Hash Algorithm，SHA）是指NIST标准化的一系列加密哈希函数。其中最著名的是**SHA-1**，它于1995年推出。该算法具有160比特的输出长度，取代了前身SHA-0，SHA-0因发现未指明的缺陷而被撤回。

在撰写本文时，尚未在SHA-1中找到明确的碰撞。然而，过去几年的理论分析表明，可以使用明显少于生日攻击所需的2^{80}次哈希函数评估来找到SHA-1中的碰撞，并且推测碰撞很快就会被找到。因此，建议迁移到**SHA-2**，SHA-2目前似乎没有同样的弱点。SHA-2由两个相关函数组成：SHA-256和SHA-512，输出长度分别为256和512比特。

SHA家族中的所有哈希函数都使用相同的基础设计，其中包含我们已经看到的组件：首先通过将Davies-Meyer构造应用于分组密码来定义压缩函数；然后使用Merkle-Damgård变换扩展

它以支持任意长度输入。这里有趣的一点是，在每种情况下，分组密码都是专门为构建压缩函数而设计的。事实上，直到后来才追溯地将压缩函数中的底层组件分离并分析为分组密码**SHACAL-1**（用于SHA-1）和**SHACAL-2**（用于SHA-2）。这些密码本身就很吸引人，因为它们具有较大的分组长度（分别为160和256比特）和512比特密钥。

6.3.4 SHA-3 (Keccak)

在MD5的碰撞攻击和SHA-1中发现的理论弱点之后，NIST于2007年末宣布举行公开竞赛，以设计一个新的加密哈希函数，命名为**SHA-3**。提交的算法被要求支持至少256和512比特的输出长度。与大约10年前的AES竞赛一样，竞赛是完全开放和透明的；任何人都可以提交算法供考虑，并邀请公众提交对任何候选算法的意见。2008年12月，51个第一轮候选算法被缩小到14个，并于2010年进一步减少到5个决赛入围者。在接下来的两年里，密码学界对剩余的候选算法进行了密集的审查。2012年10月，NIST宣布选择**Keccak**作为竞赛的获胜者。在撰写本文时，该算法正在标准化，作为SHA-2的下一代替代品。

Keccak在几个方面是不寻常的。（有趣的是，选择Keccak的原因之一是它的结构与SHA-1和SHA-2的结构**非常不同**。）其核心是基于一个具有1600比特大分组长度的**无密钥置换**f；这与Davies-Meyer构造（依赖于**带密钥置换**）等有根本区别。此外，Keccak不使用Merkle-Damgård变换来处理任意输入长度。相反，它使用了一种称为**海绵构造**（sponge construction）的较新方法。Keccak——以及更普遍的海绵构造——可以在**随机置换模型**中进行分析，在该模型中，我们假设参与方可以访问随机置换$f:\{0,1\}^l \to \{0,1\}^l$（以及可能是其逆）的预言机。这比理想密码模型弱；事实上，通过简单地将密钥固定为密码的任何常数值，我们可以在理想密码模型中轻松获得随机置换。

观察新哈希标准的发展，并观察开发人员从SHA-1/SHA-2适应到更新的SHA-3的速度，将是很有趣的。

参考文献和延伸阅读

有关LFSR和流密码的更多信息，请参阅《应用密码学手册》或Paar和Pelzl的最新文本。有关eSTREAM的更多详细信息，以及Trivium的详细规范，可在http://www.ecrypt.eu.org/stream上找到。有关RC4攻击的最新调查，请参阅AlFardan等人的工作。

混淆-扩散范式和代换-置换网络由Shannon和Feistel引入。有关SPN设计的更多信息，请参阅Heys的论文。有关三轮SPN的更好通用攻击，而不是我们在此处展示的攻击，是已知的。Miles和Viola给出了SPN的理论分析。

Feistel网络最早在中描述。Luby和Rackoff给出了Feistel网络的理论分析；参见第7章。

有关DES、AES和分组密码构造的一般信息的更多细节，可在Knudsen和Robshaw的文本中找到。双重加密的中途相遇攻击归因于Diffie和Hellman。（在练习6.13中探讨的）文本中提到的两密钥三重加密的攻击归因于Merkle和Hellman。可以在中找到双重和三重加密安全性的理论分析。

DESX是另一种增加DES有效密钥长度的技术。密钥由值$k_i, k_o \in \{0,1\}^{64}$和$k \in \{0,1\}^{56}$组成，密码定义为

$$\mathrm{DESX}_{k_i,k,k_o}(x) \stackrel{\mathrm{def}}{=} k_o \oplus \mathrm{DES}_k(x \oplus k_i).$$

这种方法最早由Even和Mansour在略有不同的背景下进行了研究。它在DES上的应用是由Rivest在未发表的工作中提出的，其安全性后来由Kilian和Rogaway进行了分析。

差分密码分析由Biham和Shamir引入，其在DES上的应用由这些作者的一本书描述。Coppersmith描述了DES S-盒的设计原则，以应对差分密码分析的公开发现。线性密码分析由Matsui发现，他展示了其在DES上的应用。有关这些高级密码分析技术的更多信息，我们请读者参考Heys的差分和线性密码分析教程或前面提到的Knudsen和Robshaw的著作。

有关MD5和SHA-1的更多信息，请参阅。但是请注意，他们的处理早于Wang等人的攻击。由分组密码构造压缩函数在中进行了分析。Bertoni等人描述和分析了海绵构造。有关SHA-3竞赛的更多详细信息，请参阅NIST网页http://csrc.nist.gov/groups/ST/hash/sha-3/index.html。