第 5 章 哈希函数及其应用

在本章中，我们介绍**密码学哈希函数**并探讨它们的几种应用。在最基本的层面上，哈希函数提供了一种将长输入字符串映射到较短输出字符串的方法，后者有时称为**摘要 (digest)**。主要要求是避免**碰撞 (collisions)**，即两个输入映射到相同的摘要。**抗碰撞哈希函数 (collision-resistant hash functions)** 有许多用途。我们将在这里看到的一个例子是另一种实现消息认证码**域扩展 (domain extension)** 的方法，即标准化的 HMAC。

除此之外，哈希函数在密码学中变得无处不在，并且通常用于需要比抗碰撞性强得多的属性的场景。将密码学哈希函数建模为“完全不可预测的”（又名**随机预言机 (random oracles)**）已成为惯例，我们将在本章稍后详细讨论这个框架——以及围绕它的争议。我们在这里只涉及随机预言机模型的少数几个应用，但在转向**公钥密码学**时会再次遇到它。

哈希函数之所以引人入胜，是因为它们既可以被视为介于私钥密码学和公钥密码学之间，又可以被视为两者兼有。一方面，正如我们将在第 6 章中看到的那样，它们（在实践中）是使用对称密钥技术构建的，并且哈希函数的许多规范应用都在对称密钥设置中。另一方面，从理论的角度来看，抗碰撞哈希函数的存在似乎代表了一种**定性上更强**的假设，因为它比伪随机函数的存在更强（但比公钥加密的存在更弱）。

5.1 定义 (Definitions)

哈希函数只是接受某种长度的输入并将其压缩成较短的固定长度输出的函数。哈希函数的经典用途是在数据结构中，它们可用于构建哈希表，从而在存储一组元素时实现 $O(1)$ 的查找时间。具体来说，如果哈希函数 H 的范围大小为 N，则元素 x 存储在大小为 N 的表的第 $H(x)$ 行中。要检索 x，只需计算 $H(x)$ 并探测存储在那里的元素的该行。用于此目的的“好”哈希函数应该产生很少的**碰撞**，其中碰撞是一对不同的项 x 和 x'，使得 $H(x) = H(x')$；在这种情况下，我们也说 x 和 x' 发生了**冲突**。（当发生碰撞时，两个元素最终存储在同一个单元格中，从而增加了查找时间。）

抗碰撞哈希函数在精神上是相似的。同样，目标是避免碰撞。然而，存在根本的区别。首先，在数据结构设置中将碰撞最小化的愿望变成了密码学设置中避免碰撞的要求。此外，在数据结构的环境中，我们可以假设数据元素集是独立于哈希函数选择的，并且无意造成碰撞。相比之下，在密码学的环境中，我们面临着一个攻击者，他选择元素明确地以造成碰撞为目标。这意味着抗碰撞哈希函数设计起来要困难得多。

5.1.1 抗碰撞性 (Collision Resistance)

非形式化地，如果任何概率多项式时间算法都无法在 H 中找到碰撞，则函数 H 是**抗碰撞的 (collision resistant)**。我们将只对域大于其范围的哈希函数感兴趣。在这种情况下，碰撞必然存在，但应该很难找到。

形式上，我们考虑**带密钥的哈希函数 (keyed hash functions)**。也就是说，H 是一个双输入函数，它以密钥 s 和字符串 x 作为输入，并输出字符串 $H^s(x) \stackrel{\text{def}}{=} H(s,x)$。要求是对于随机生成的密钥 s，在 H^s 中找到碰撞必须是困难的。在*密钥*在这个背景下与我们迄今为止使用过的密钥至少有两个不同之处。首先，并非所有字符串都必然对应于有效密钥（即 H^s 可能未针对某些 s 定义），因此密钥 s 通常由算法 Gen 生成，而不是均匀选择。第二，也许更重要的是，这个密钥 s（通常）**不会**保密，即使攻击者获得了 s，也需要抗碰撞性。为了强调这一点，我们将密钥上标并写成 H^s 而不是 H_s。

定义 5.1 具有输出长度 ℓ 的哈希函数是一个概率多项式时间算法对 (Gen, H)，满足以下条件：

- Gen 是一个概率算法，它以安全参数 1^n 作为输入并输出一个密钥 s。我们假设 1^n 在 s 中是隐式的。
- H 接受密钥 s 和字符串 $x \in \{0,1\}^*$ 作为输入，并输出字符串 $H^s(x) \in \{0,1\}^{\ell(n)}$（其中 n 是 s 中隐含的安全参数的值）。

如果 H^s 仅针对输入 $x \in \{0,1\}^{\ell'(n)}$ 定义，并且 $\ell'(n) > \ell(n)$，则我们称 (Gen, H) 是**用于长度为 ℓ' 的输入固定长度哈希函数 (fixed-length hash function for inputs of length ℓ')**。在这种情况下，我们也称 H 为**压缩函数 (compression function)**。

在固定长度的情况下，我们要求 ℓ' 大于 ℓ。这确保了该函数会**压缩**其输入。在一般情况下，该函数接受任意长度的输入字符串。因此，它也进行压缩（尽管只是长度大于 $\ell(n)$ 的字符串）。请注意，如果没有压缩，抗碰撞性是微不足道的（因为可以简单地采用恒等函数 $H^s(x) = x$ ）。

我们现在继续定义安全性。像往常一样，我们首先为哈希函数 $\Pi = (\mathrm{Gen}, H)$、攻击者 $\mathcal{A}$ 和安全参数 n 定义一个实验：

碰撞寻找实验 $\mathrm{Hash\text{-}coll}_{\mathcal{A},\Pi}(n)$：

1. 运行 $\mathrm{Gen}(1^n)$ 生成密钥 s。
2. 攻击者 $\mathcal{A}$ 获得 s 作为输入，并输出 x, x'。 （如果 Π 是用于长度为 $\ell'(n)$ 的输入的固定

长度哈希函数，那么我们要求 $x, x' \in \{0,1\}^{\ell'(n)}$。）
3. 当且仅当 $x \neq x'$ 且 $H^s(x) = H^s(x')$ 时，实验的输出定义为 1。在这种情况下，我们说 $\mathcal{A}$ **找到了一个碰撞 (found a collision)**。

抗碰撞性的定义指出，除了可忽略的概率外，任何高效的攻击者都无法在上述实验中找到碰撞。

定义 5.2 哈希函数 $\Pi = (\mathrm{Gen}, H)$ 是**抗碰撞**的，如果对于所有概率多项式时间攻击者 $\mathcal{A}$，存在一个可忽略函数 negl，使得

$$\Pr[\text{Hash-coll}_{\mathcal{A},\Pi}(n) = 1] \leq \mathrm{negl}(n).$$

为简单起见，我们有时将 H 或 H^s 称为“抗碰撞哈希函数”，尽管从技术上讲，我们应该只说 (Gen, H) 是抗碰撞的。这不应造成任何混淆。

密码学哈希函数的设计有明确的抗碰撞目标（以及其他目标）。我们将在第 6 章中讨论一些常见的现实世界哈希函数。我们将在 8.4.2 节中看到，基于某个数论问题的困难性假设，可以构造出具有可证明抗碰撞性的哈希函数。

无密钥哈希函数 (Unkeyed hash functions)。 实践中使用的密码学哈希函数通常具有固定的输出长度（就像块密码具有固定的密钥长度一样），并且通常是**无密钥的 (unkeyed)**，这意味着哈希函数只是一个固定函数 $H : \{0,1\}^* \to \{0,1\}^\ell$。从理论的角度来看，这是有问题的，因为对于任何此类函数，总是存在一个在常数时间内输出碰撞的算法：该算法只是将碰撞对 (x, x') **硬编码**到算法本身中。使用带密钥的哈希函数解决了这个技术问题，因为对于每个可能的密钥，都无法使用合理的空间量来硬编码碰撞对（并且在渐进设置中，对于安全参数的每个值，都无法硬编码碰撞对）。

尽管有上述情况，但在实际应用中使用的（无密钥）密码学哈希函数对于所有实际目的都是抗碰撞的，因为碰撞对是未知的（并且计算上难以找到），即使它们必然存在。基于哈希函数的抗碰撞性的某些构造的安全性证明是有意义的，只要使用一个**无密钥**的哈希函数 H，只要证明任何“破坏”原语的高效攻击者可以被用来高效地在 H 中找到碰撞即可。（本书中的所有证明都满足此条件。）在这种情况下，安全性证明的解释是：如果攻击者可以在实践中破坏该方案，那么就可以用它在实践中找到碰撞，而我们认为这是很难做到的。

5.1.2 较弱的安全性概念 (Weaker Notions of Security)

在某些应用中，只需要依赖比抗碰撞性弱的安全性要求。这些包括：

- **第二原像（或目标碰撞）抗性 (Second-preimage or target-collision**

resistance)： 非形式化地，哈希函数是第二原像抗性的，如果给定 s 和均匀的 x，对于 ppt 攻击者来说，找到 $x' \neq x$ 使得 $H^s(x') = H^s(x)$ 是不可行的。
- **原像抗性 (Preimage resistance)：** 非形式化地，哈希函数是原像抗性的，如果给定 s 和均匀的 y，对于 ppt 攻击者来说，找到值 x 使得 $H^s(x) = y$ 是不可行的。（展望第 7 章，这基本上意味着 H^s 是**单向**的。）

任何抗碰撞的哈希函数也是第二原像抗性的。这是因为，如果给定均匀的 x，攻击者可以找到 $x' \neq x$ 使得 $H^s(x') = H^s(x)$，那么他就可以清楚地找到碰撞对 x 和 x'。同样，任何第二原像抗性的哈希函数也是原像抗性的。这是因为，如果可以在给定 y 的情况下找到 x 使得 $H^s(x) = y$，那么也可以采用一个给定的输入 x'，计算 $y := H^s(x')$，然后得到一个 x 使得 $H^s(x) = y$。在很高的概率下，$x \neq x'$（这依赖于 H 进行压缩，因此多个输入映射到相同的输出），在这种情况下找到了一个第二原像。

我们不会正式定义上述概念或证明上述蕴涵关系，因为它们在本书的其余部分中没有使用。要求你在练习 5.1 中将上述概念形式化。

5.2 域扩展：Merkle-Damgård 变换 (Domain Extension: The Merkle–Damgård Transform)

哈希函数通常分两步构建。首先，设计一个抗碰撞的**压缩函数** h，它处理固定长度的输入；然后，使用**域扩展 (domain extension)** 来处理任意长度的输入。在本节中，我们展示了域扩展问题的一种解决方案。我们将在 6.3 节中回到设计抗碰撞压缩函数的问题。

Merkle–Damgård 变换是一种将压缩函数扩展为成熟哈希函数的常见方法，同时保持前者的抗碰撞属性。它在实践中被广泛用于包括 MD5 和 SHA 家族（参见 6.3 节）在内的哈希函数。这种变换的存在意味着在设计抗碰撞哈希函数时，我们可以将注意力限制在固定长度的情况。这反过来又使设计抗碰撞哈希函数的工作变得更容易。从理论的角度来看，Merkle–Damgård 变换也很有趣，因为它意味着单比特压缩与任意量压缩一样容易（或困难）。

为具体起见，假设压缩函数 (Gen, h) 将其输入压缩一半；假设其输入长度为 $2n$，输出长度为 n。（无论输入/输出长度如何，只要 h 进行了压缩，该构造都有效。）我们构造一个将任意长度输入映射到长度 n 输出的抗碰撞哈希函数 (Gen, H)。(Gen 保持不变。)
Merkle–Damgård 变换在**构造 5.3** 中定义并如图 5.1 所示。在构造的第 2 步中使用的值 z_0 称为**初始化向量 (initialization vector, IV)**，是任意的并且可以被任何常数代替。

构造 5.3

设 (Gen, h) 是一个用于长度为 $2n$ 的输入且输出长度为 n 的固定长度哈希函数。构造哈希函数 (Gen, H) 如下：

- Gen：保持不变。
- H：输入密钥 s 和长度 $L < 2^n$ 的字符串 $x \in \{0,1\}^*$，执行以下操作：
  i. 设 $B := \lceil L/n \rceil$（即 x 中的块数）。用零填充 x，使其长度是 n 的倍数。将填充后的结果解析为 n 比特块序列 $x_1, \ldots, x_B$。设 $x_{B+1} := L$，其中 L 编码为一个 n 比特字符串。
  ii. 设 $z_0 := 0^n$。（这也称为 IV。）
  iii. 对于 $i = 1, \ldots, B+1$，计算 $z_i := h^s(z_{i-1} \| x_i)$。
  iv. 输出 z_{B+1}。

Merkle–Damgård 变换。

定理 5.4 如果 (Gen, h) 是抗碰撞的，那么 (Gen, H) 也是抗碰撞的。

证明 我们证明对于任何 s，在 H^s 中的碰撞会导致在 h^s 中的碰撞。设 x 和 x' 是两个不同的长度分别为 L 和 L' 的字符串，使得 $H^s(x) = H^s(x')$。设 $x_1, \ldots, x_B$ 是填充后的 x 的 B 个块，设 $x'_1, \ldots, x'_{B'}$ 是填充后的 x' 的 B' 个块。回想一下 $x_{B+1} = L$ 和 $x'_{B'+1} = L'$。考虑以下两种情况：

1. **情况 1：** $L = L'$。在这种情况下，$H^s(x)$ 的计算的最后一步是 $z_{B+1} := h^s(z_B \| L)$，而 $H^s(x')$ 的计算的最后一步是 $z'_{B'+1} := h^s(z'_{B'} \| L')$。由于 $H^s(x) = H^s(x')$，因此 $h^s(z_B \| L) = h^s(z'_{B'} \| L')$。然而，$L = L'$ 且 $x \neq x'$，因此 $z_B \| L$ 和 $z'_{B'} \| L'$ 是在 h^s 下发生碰撞的两个不同字符串。
2. **情况 2：** $L \neq L'$。这意味着 $B \neq B'$。设 $z_0, \ldots, z_{B+1}$ 是在计算 $H^s(x)$ 期间定义的值，设 $I_i \stackrel{\text{def}}{=} z_{i-1} \| x_i$ 是 h^s 的第 i 个输入，并设 $I_{B+2} \stackrel{\text{def}}{=} z_{B+1}$。类似地，对于 x' 定义 $I'_1, \ldots, I'_{B'+2}$。设 N 是使得 $I_N = I'_N$ 的最大索引。由于 $|x| = |x'|$ 但 $x \neq x'$，因此存在一个 i 使得 $x_i \neq x'_i$，因此这样的 N 必然存在。因为

$$I_{B+2} = z_{B+1} = H^s(x) = H^s(x') = z'_{B'+1} = I'_{B'+2},$$

   我们有 $N \leq B+1$。根据 N 的最大性，我们有 $I_{N+1} = I'_{N+1}$，特别是 $z_N = z'_N$。但这仅仅意味着 I_N, I'_N 是 h^s 中的一个碰撞。

我们将其留作将上述内容转化为形式化归约的练习。

5.3 使用哈希函数进行消息认证 (Message Authentication Using Hash Functions)

在上一章中，我们介绍了两种构造任意长度消息认证码的方法。第一种方法是通用的，但效率低下。第二种方法 CBC-MAC 基于伪随机函数。在这里，我们将看到另一种方法，我们称之为“**哈希-认证码 (hash-and-MAC)**”，它依赖于抗碰撞哈希和任何消息认证码。然后我们讨论一个称为 **HMAC** 的标准化且广泛使用的构造，可以将其视为这种方法的具体实例化。

5.3.1 哈希-认证码 (Hash-and-MAC)

哈希-认证码方法背后的思想很简单。首先，使用抗碰撞哈希函数将任意长的消息 m 哈希成固定长度的字符串 $H^s(m)$。然后，将一个（固定长度）MAC 应用于该结果。**构造 5.5** 提供了正式的描述。

构造 5.5

设 $\Pi = (\mathrm{Mac}, \mathrm{Vrfy})$ 是用于长度为 $\ell(n)$ 的消息的 MAC，设 $\Pi_H = (\mathrm{Gen}_H, H)$ 是输出长度为 $\ell(n)$ 的哈希函数。构造用于任意长度消息的 MAC $\Pi' = (\mathrm{Gen}', \mathrm{Mac}', \mathrm{Vrfy}')$ 如下：

- Gen'： 输入 1^n，选择均匀的 $k \in \{0,1\}^n$ 并运行 $\mathrm{Gen}_H(1^n)$ 获取 s；密钥为 $k' := (k, s)$。
- Mac'： 输入密钥 (k, s) 和消息 $m \in \{0,1\}^*$，输出 $t \leftarrow \mathrm{Mac}_k(H^s(m))$。
- Vrfy'： 输入密钥 (k, s)、消息 $m \in \{0,1\}^*$ 和 MAC 标签 t，当且仅当 $\mathrm{Vrfy}_k(H^s(m), t) = 1$ 时输出 1。

哈希-认证码范式 (The hash-and-MAC paradigm)。

如果 Π 是安全 MAC 且 (Gen, H) 是抗碰撞的，则**构造 5.5** 是安全的。直观上，由于哈希函数是抗碰撞的，因此认证 $H^s(m)$ 与认证 m 本身一样好：如果发送方能够确保接收方获得正确的 $H^s(m)$ 值，则抗碰撞性保证攻击者无法找到另一个哈希到相同值 $H^s(m)$ 的不同消息 m'。更正式地，假设发送方使用**构造 5.5** 来认证一组消息 $\mathcal{Q}$，并且攻击者 $\mathcal{A}$ 能够对一个**新消息** $m^* \notin \mathcal{Q}$ 伪造有效标签。有两种可能的情况：

情况 1： 存在一个消息 $m \in \mathcal{Q}$ 使得 $H^s(m^*) = H^s(m)$。那么 $\mathcal{A}$ 在 H^s 中找到了一个碰撞，这与 (Gen, H) 的抗碰撞性相矛盾。

情况 2： 对于每个消息 $m \in \mathcal{Q}$，都有 $H^s(m^*) \neq H^s(m)$。设 $H^s(\mathcal{Q}) \stackrel{\text{def}}{=} \{H^s(m) \mid$

$m \in \mathcal{Q}\}$。那么 $H^s(m^*) \notin H^s(\mathcal{Q})$。在这种情况下，$\mathcal{A}$ 对“新消息” $H^s(m^*)$ 伪造了一个有效标签，这与固定长度消息认证码 Π 是安全 MAC 的假设相矛盾。

我们现在将上述直觉转化为形式证明。

定理 5.6 如果 Π 是用于长度为 ℓ 的消息的安全 MAC 且 Π_H 是抗碰撞的，则**构造 5.5** 是安全 MAC（适用于任意长度的消息）。

证明 设 Π' 是**构造 5.5** 所示的方案，设 $\mathcal{A}'$ 是攻击 Π' 的 ppt 攻击者。在实验 $\text{Mac-forge}_{\mathcal{A}',\Pi'}(n)$ 的一次执行中，设 $k' = (k, s)$ 是 MAC 密钥，设 $\mathcal{Q}$ 是 $\mathcal{A}'$ 请求标签的消息集合，设 (m^*, t) 是 $\mathcal{A}'$ 的最终输出。我们假设 $m^* \notin \mathcal{Q}$。定义 coll 是指在实验 $\text{Mac-forge}_{\mathcal{A}',\Pi'}(n)$ 中存在一个 $m \in \mathcal{Q}$ 使得 $H^s(m^*) = H^s(m)$ 的事件。我们有

$$\Pr[\text{Mac-forge}_{\mathcal{A}',\Pi'}(n) = 1] = \Pr[\text{Mac-forge}_{\mathcal{A}',\Pi'}(n) = 1 \wedge \text{coll}] + \Pr[\text{Mac-forge}_{\mathcal{A}',\Pi'}(n) = 1 \wedge$$

我们证明方程 (5.1) 中的两项都是可忽略的，从而完成证明。直观上，第一项由于 Π_H 的抗碰撞性而是可忽略的，第二项由于 Π 的安全性而是可忽略的。

考虑以下用于在 Π_H 中寻找碰撞的算法 $\mathcal{C}$：

算法 $\mathcal{C}$：

该算法给定 s 作为输入（n 是隐式的）。

- 选择均匀的 $k \in \{0,1\}^n$。
- 运行 $\mathcal{A}'(1^n)$。当 $\mathcal{A}'$ 请求第 i 条消息 $m_i \in \{0,1\}^*$ 上的标签时，计算 $t_i \leftarrow \text{Mac}_k(H^s(m_i))$ 并将 t_i 给予 $\mathcal{A}'$。
- 当 $\mathcal{A}'$ 输出 (m^*, t) 时，如果存在一个 i 使得 $H^s(m^*) = H^s(m_i)$，则输出 (m^*, m_i)。

显然 $\mathcal{C}$ 在多项式时间内运行。当 $\mathcal{C}$ 的输入是通过运行 $\text{Gen}_H(1^n)$ 获取 s 来生成时，当 $\mathcal{A}'$ 作为 $\mathcal{C}$ 的子例程运行时，其视图与 $\mathcal{A}'$ 在实验 $\text{Mac-forge}_{\mathcal{A}',\Pi'}(n)$ 中的视图完全相同。特别是，$\mathcal{A}'$ 从 $\mathcal{C}$ 获得的标签分布与 $\mathcal{A}'$ 在 $\text{Mac-forge}_{\mathcal{A}',\Pi'}(n)$ 中获得的标签分布相同。由于 $\mathcal{C}$ 仅在 coll 发生时输出碰撞，因此我们有

$$\Pr[\text{Hash-coll}_{\mathcal{C},\Pi_H}(n) = 1] = \Pr[\text{coll}].$$

由于 Π_H 是抗碰撞的，我们得出结论 $\Pr[\text{coll}]$ 是可忽略的。

我们现在继续证明方程 (5.1) 中的第二项是可忽略的。考虑以下攻击 Π 的攻击者 $\mathcal{A}$，它在实验 $\text{Mac-forge}_{\mathcal{A},\Pi}(n)$ 中运行：

攻击者 $\mathcal{A}$：

攻击者 $\mathcal{A}$ 具有对 MAC 预言机 $\mathrm{Mac}_k(\cdot)$ 的访问权限。

- 计算 $\mathrm{Gen}_H(1^n)$ 获取 s。
- 运行 $\mathcal{A}'(1^n)$。当 $\mathcal{A}'$ 请求第 i 条消息 $m_i \in \{0,1\}^*$ 上的标签时，则：(1) 计算 $\hat{m}_i := H^s(m_i)$；(2) 从 MAC 预言机获取 $\hat{m}_i$ 上的标签 t_i；(3) 将 t_i 给予 $\mathcal{A}'$。
- 当 $\mathcal{A}'$ 输出 (m^*, t) 时，则输出 $(H^s(m^*), t)$。

显然 $\mathcal{A}$ 在多项式时间内运行。考虑实验 $\text{Mac-forge}_{\mathcal{A},\Pi}(n)$。在该实验中，当 $\mathcal{A}$ 作为 $\mathcal{A}'$ 的子例程运行时，$\mathcal{A}'$ 的视图与 $\mathcal{A}'$ 在实验 $\text{Mac-forge}_{\mathcal{A}',\Pi'}(n)$ 中的视图完全相同。此外，当 $\text{Mac-forge}_{\mathcal{A}',\Pi'}(n) = 1$ 且 coll 未发生时，$\mathcal{A}$ 输出一个有效伪造。（在这种情况下，t 是方案 Π 中 k 的 $H^s(m^*)$ 上的有效标签。 coll 未发生的事实意味着 $\mathcal{A}'$ 以前从未向其自己的 MAC 预言机询问 $H^s(m^*)$，因此这确实是一个伪造。）因此，

$$\Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1] = \Pr[\text{Mac-forge}_{\mathcal{A}',\Pi'}(n) = 1 \wedge \overline{\text{coll}}],$$

且 Π 的安全性意味着前者的概率是可忽略的。这就完成了定理的证明。

5.3.2 HMAC

我们目前看到的所有消息认证码构造最终都基于某种块密码。是否有可能**直接**基于哈希函数构造一个安全 MAC（用于任意长度消息）？ 第一个想法可能是定义 $\mathrm{Mac}_k(m) = H(k\|m)$；我们可能期望，如果 H 是一个“好”的哈希函数，那么对于攻击者来说，在给定 $H(k\|m)$ 的值的情况下，预测 $H(k\|m')$ 的值对于任何 $m' \neq m$ 都是困难的，假设 k 是随机选择的（并且攻击者不知道）。不幸的是，如果 H 是使用 Merkle–Damgård 变换构造的——正如大多数现实世界哈希函数一样——那么以这种方式设计的 MAC 是完全不安全的，正如你在练习 5.10 中被要求证明的那样。

相反，我们可以尝试使用两层哈希。参见**构造 5.7**，它基于这一思想提出了一种称为 **HMAC** 的标准化方案。

构造 5.7

设 (Gen_H, H) 是一个通过对压缩函数 (Gen_H, h) 应用 Merkle–Damgård 变换构造的哈希函数，它接受长度为 $n + n'$ 的输入。（参见正文。）设 opad 和 ipad 是长度为 n' 的固定常数。定义 MAC 如下：

- Gen： 输入 1^n，运行 $\mathrm{Gen}_H(1^n)$ 获取密钥 s。另选均匀的 $k \in \{0,1\}^n$。输出密钥 s, k

。

- Mac：输入密钥 s, k 和消息 $m \in \{0,1\}^*$，输出

$$t := H^s((k \oplus \mathrm{opad}) \| H^s((k \oplus \mathrm{ipad}) \| m)).$$

- Vrfy：输入密钥 s, k、消息 $m \in \{0,1\}^*$ 和标签 t，当且仅当 $t = H^s((k \oplus \mathrm{opad}) \| H^s((k \oplus \mathrm{ipad}) \| m))$ 时输出 1。

HMAC。

图 5.2：HMAC，图示。

我们为什么要相信 HMAC 是安全的呢？一个原因是我们可以将 HMAC 视为上一节中哈希-认证码范式的一种特定实例化。为此，我们将“深入了解”消息认证时发生的情况；参见图 5.2。我们还必须更仔细地指定参数，并更详细地讨论 Merkle–Damgård 变换在实践中的实现方式。

假设 (Gen_H, H) 是基于压缩函数 (Gen_H, h) 构造的，其中 h 将长度为 $n + n'$ 的输入映射到长度为 n 的输出（其中，形式上 n' 是 n 的函数）。当我们在 5.2 节中描述 Merkle–Damgård 变换时，我们假设了 $n' = n$，但这并非总是必需的。我们还说过，被哈希消息的长度被编码为一个额外的消息块，并附加到消息末尾。在实践中，长度反而被编码在块的一部分，使用 $\ell < n'$ 比特。也就是说， $H^s(x)$ 的计算开始于用零填充 x，使其长度恰好比 n' 的倍数少 ℓ；然后它附加长度 $L = |x|$，编码使用恰好 ℓ 比特。然后，以**构造 5.3** 中的方式计算所得 n' 比特块序列 $x_1, \ldots$ 的哈希值。我们假设 $n + \ell \leq n'$。这意味着，特别是，如果我们哈希一个长度为 n 的输入 x，则填充后的结果（包括长度）将恰好是 $2n'$ 比特长。**定理 5.4** 的证明仍然成立，即如果 (Gen_H, h) 是抗碰撞的，则 (Gen_H, H) 是抗碰撞的。

回到 HMAC，并查看图 5.2，我们可以看到 HMAC 的一般形式涉及将任意长度消息哈希到短字符串 $y \stackrel{\mathrm{def}}{=} H^s((k \oplus \mathrm{ipad}) \| m)$，然后计算结果的（秘密密钥）函数 $H^s((k \oplus \mathrm{opad}) \| y)$。但我们可以说得更多。首先注意“内部”计算

$$\tilde{H}^s(m) \stackrel{\mathrm{def}}{=} H^s((k \oplus \mathrm{ipad}) \| m)$$

对于 $k \oplus \mathrm{ipad}$ 的任何值，它都是抗碰撞的（假设 h 是抗碰撞的）。此外，“外部”计算 $H^s((k \oplus \mathrm{opad}) \| y)$ 的第一步是计算一个值 $k_{\mathrm{out}} \stackrel{\mathrm{def}}{=} h^s(IV \| (k \oplus \mathrm{opad}))$。然后，我们评估 $h^s(k_{\mathrm{out}} \| \hat{y})$，其中 $\hat{y}$ 指的是 y 的填充值（即包括 $(k \oplus \mathrm{opad}) \| y$ 的长度，其长度总是 $n' + n$ 比特，编码使用恰好 ℓ 比特）。因此，如果我们把 k_{out} 视为均匀分布的——我们将在下面更正式地说明这一点——并假设

$$\mathrm{Mac}_{k_{\mathrm{out}}}(y) \stackrel{\mathrm{def}}{=} h^s(k_{\mathrm{out}} \| \hat{y})$$

是安全固定长度 MAC，那么 HMAC 可以被视为哈希-认证码方法的实例化

$$\mathrm{HMAC}_{s,k}(m) = \mathrm{Mac}_{k_{\mathrm{out}}}(\tilde{H}^s(m))$$

（其中 $k_{\mathrm{out}} = h^s(IV \| (k \oplus \mathrm{opad}))$）。正如前面所指出的，可以证明这种构造是安全的（使用哈希-认证码方法的证明变体），如果 H 是**弱抗碰撞的**并且在方程 (5.2) 中定义的 MAC 是安全固定长度 MAC。

ipad 和 opad 的作用。 鉴于上述情况，有人可能会疑问为什么需要在“内部”计算 $H^s((k \oplus \mathrm{ipad}) \| m)$ 中包含 k。（特别是，为了使哈希-认证码方法安全，我们在第一步中需要抗碰撞性，这不需要任何秘密密钥。）原因是这使得 HMAC 的安全性可以基于**弱抗碰撞**的较弱假设（如果 h 是抗碰撞的，则显然是弱抗碰撞的；然而，后者是一个较弱的条件，可能更容易满足）。这是一个很好的*稳健安全工程*的例子。这种防御性设计策略在发现哈希函数 MD5（参见 6.3.2 节）不抗碰撞时得到了回报。对 MD5 的碰撞寻找攻击并未破坏弱抗碰撞性，因此 HMAC-MD5 即使在 MD5 存在弱点的情况下也没有被破解。这为开发人员提供了时间来替换 HMAC 实现中的 MD5，而没有立即的攻击担忧。（尽管如此，由于 MD5 中已知的弱点，HMAC-MD5 不应再使用。）

上述讨论表明，在外部和内部计算中应该使用**独立密钥**。出于效率原因，HMAC 使用单个密钥 k，但该密钥与 ipad 和 opad 结合使用以导出另外两个密钥。定义

$$G^s(k) \stackrel{\mathrm{def}}{=} h^s(IV \| (k \oplus \mathrm{opad})) \| h^s(IV \| (k \oplus \mathrm{ipad})) = k_{\mathrm{out}} \| k_{\mathrm{in}}.$$

如果我们假设 G^s 对于任何 s 都是一个伪随机生成器，那么当 k 是均匀的时， k_{out} 和 k_{in} 可以被视为独立且均匀的密钥。HMAC 的安全性随后归结为以下构造的安全性：

$$\mathrm{Mac}_{s,k_{\mathrm{in}},k_{\mathrm{out}}}(m) = h^s(k_{\mathrm{out}} \| \tilde{H}^s_{k_{\mathrm{in}}}(m)).$$

（与方程 (5.3) 比较。）正如前面所指出的，如果 H 是弱抗碰撞的并且方程 (5.2) 中定义的 MAC 是安全固定长度 MAC，则可以证明这种构造是安全的（使用哈希-认证码证明的变体）。

定理 5.8 假设方程 (5.4) 中定义的 G^s 是对于任何 s 的伪随机生成器，方程 (5.2) 中定义的 MAC 是用于长度为 n 的消息的安全固定长度 MAC，并且 (Gen_H, H) 是弱抗碰撞的。那么 HMAC 是一个安全 MAC（用于任意长度的消息）。

实践中的 HMAC (HMAC in practice)。 HMAC 是一个行业标准，在实践中得到广泛使用。它高效且易于实现，并且其安全性基于被认为对实用哈希函数成立的假设获得了证明。

HMAC 的重要性部分归因于其出现及时。在 HMAC 推出之前，许多实践者拒绝使用 CBC-MAC（声称它“太慢”），而是使用了不安全的启发式构造。HMAC 提供了一种标准化、安全的使用哈希函数进行消息认证的方法。

5.4 哈希函数的通用攻击 (Generic Attacks on Hash Functions)

我们能期望哈希函数 H 提供最好的安全性是什么？我们通过展示两种对**任意**哈希函数都适用的攻击来探讨这个问题。这些攻击的存在意味着要达到某些期望的安全级别，所需的 H 的输出长度有一个下限，因此具有重要的实际影响。

5.4.1 寻找碰撞的生日攻击 (Birthday Attacks for Finding Collisions)

设 $H : \{0,1\}^* \to \{0,1\}^\ell$ 是一个哈希函数。（在这里和本章的其余部分，我们省略了对哈希密钥 s 的明确提及，因为它不直接相关。也可以将 s 视为在应用这些算法之前生成并固定的。）有一个在 $O(2^\ell)$ 时间内运行的简单碰撞寻找攻击：只需在 $2^\ell + 1$ 个不同输入上评估 H；根据鸽巢原理，其中两个输出必然相等。我们能做得更好吗？

推广上述算法，假设我们选择 q 个不同的输入 $x_1, \ldots, x_q$，计算 $y_i := H(x_i)$，并检查是否存在任何两个 y_i 值相等。此算法找到碰撞的概率是多少？正如我们刚才所说，如果 $q > 2^\ell$，则碰撞发生的概率为 1。当 q 较小时，碰撞的概率是多少？精确分析这个概率有些困难，因此我们转而分析一个理想化的案例，其中 H 被视为**随机函数 (random function)**。[1]也就是说，对于每个 i，我们假设值 $y_i = H(x_i)$ 在 $\{0,1\}^\ell$ 中均匀分布，并且独立于任何先前的输出值 $\{y_j\}_{j<i}$（回想一下，我们假设所有 $\{x_i\}$ 都是不同的）。因此，我们将问题简化为以下问题：如果我们从 $\{0,1\}^\ell$ 中均匀随机选择 q 个值 $y_1, \ldots, y_q$，那么存在不同的 i, j 使得 $y_i = y_j$ 的概率是多少？

这个问题已被广泛研究，并与附录 A.4 中详细讨论的所谓**生日问题 (birthday problem)** 相关。我们描述的碰撞寻找算法通常被称为**生日攻击 (birthday attack)**。生日问题是：如果一个房间里有 q 个人，他们中两人拥有相同生日的概率是多少？（假设生日均匀且独立地分布在非闰年的 365 天中。）这与我们的问题完全相似：如果 y_i 代表第 i 个人的生日，那么我们有 q 个均匀选择的值 $y_1, \ldots, y_q \in \{1, \ldots, 365\}$，匹配的生日对应于不同的 i, j 使得 $y_i = y_j$（即匹配的生日对应于碰撞）。

在附录 A.4 中，我们证明了对于在 $\{1, \ldots, N\}$ 中均匀选择的 $y_1, \ldots, y_q$，碰撞的概率在

$q = \Theta(\sqrt{N})$ 时约为 $1/2$。在生日案例中，只要有 23 个人，他们中就有人拥有相同生日的概率就大于 $1/2$。在我们的设置中，这意味着当哈希函数的输出长度为 ℓ（因此范围大小为 2^ℓ）时，取 $q = \Theta(2^{\ell/2})$ 会导致碰撞的概率约为 $1/2$。

从**具体安全性 (concrete-security)** 的角度来看，上述情况意味着，为了使哈希函数抵抗在时间 T 内运行的碰撞寻找攻击（其中我们以评估 H 的时间为单位），哈希函数的输出长度需要至少为 $2\log T$ 比特（因为 $2^{(2\log T)/2} = T$）。取特定的参数，这意味着如果我们希望找到碰撞的难度与对 128 比特密钥进行穷举搜索的难度相当，那么我们需要的哈希函数输出长度至少为 256 比特。我们强调，拥有如此长的输出仅仅是**必要**条件，而不是充分条件。我们还注意到，生日攻击仅适用于寻找碰撞。对于哈希函数 H 的第二原像抗性或原像抗性，没有通用攻击需要少于 2^ℓ 次 H 评估（尽管参见 5.4.3 节）。

寻找有意义的碰撞 (Finding meaningful collisions)。 刚才描述的生日攻击找到的碰撞不一定有用。但是，同样的想法可以用于寻找“有意义”的碰撞。假设爱丽丝希望找到两个消息 x 和 x' 使得 $H(x) = H(x')$，并且 x 应该是她雇主解释她为何被解雇的信，而 x' 应该是一封赞美的推荐信。（如果她的雇主使用哈希-认证码方法来认证消息，这可能允许爱丽丝伪造适当的推荐信标签。）观察到生日攻击只要求哈希输入 $x_1, \ldots, x_q$ 互不相同；它们不需要是随机的。爱丽丝可以进行一次生日类型攻击，生成 $q = \Theta(2^{\ell/2})$ 个第一种类型的消息和 q 个第二种类型的消息，然后寻找这两种类型消息之间的碰撞。对附录 A.4 的分析进行一个小小的修改表明，这会以大约 $1/2$ 的概率产生两种不同类型消息之间的碰撞。

稍加思考就会发现，以多种不同方式编写同一消息是很容易的。例如，考虑以下句子：

It is **hard/difficult/challenging/impossible** *to* **imagine/believe** *that we will* **find/locate/hire** *another* **employee/person** *having similar* **abilities/skills/character** *as Alice. She has done a* **great/super** *job.*

带斜体字的单词的任何组合都是可能的，并且表达了相同的观点。因此，可以用 $4 \cdot 2 \cdot 3 \cdot 2 \cdot 3 \cdot 2 = 288$ 种不同的方式来编写该句子。这只是一个句子，因此实际上很容易生成 2^{64} 种不同的方式来重写同一消息——只需要 64 个每个都带有一个同义词的单词。爱丽丝可以准备 $2^{\ell/2}$ 封解释她为何被解雇的信和另外 $2^{\ell/2}$ 封推荐信；以高概率，将找到这两种类型的信件之间的碰撞。

5.4.2 小空间生日攻击 (Small-Space Birthday Attacks)

上面描述的生日攻击需要大量的内存；具体来说，它们要求攻击者存储所有 $O(q) = O(2^{\ell/2})$

个值 $\{y_i\}$，因为攻击者事先不知道哪一对值会产生碰撞。这是一个重大的缺点，因为内存通常是比时间更稀缺的资源。可以说，为 2^{60} 字节分配和管理存储比执行 2^{60} 个 CPU 指令要困难得多。此外，算法的内存要求必须在其需要时立即得到满足，而计算可以无限期地运行。

我们在这里展示一种内存需求大大减少的更好生日攻击。事实上，它的时间复杂度和成功概率与以前相似，但仅使用恒定的内存。攻击开始于选择一个随机值 x_0，然后计算 $x_i := H(x_{i-1})$ 和 $x_{2i} := H(H(x_{2(i-1)}))$，其中 $i = 1, 2, \ldots$。（请注意，对于所有 i， $x_i = H^{(i)}(x_0)$，其中 $H^{(i)}$ 指的是 H 的 i 次迭代。）在每一步中，比较 x_i 和 x_{2i} 的值；如果它们相等，则在序列 $x_0, x_1, \ldots, x_{2i-1}$ 中的某个位置存在碰撞。然后，算法找到使得 $x_j = x_{j+i}$ 的最小 j（请注意 $j \leq i$ 因为 $j = i$ 是有效的），并输出 x_{j-1}, x_{j+i-1} 作为碰撞。形式上，该算法描述为**算法 5.9** 并在下面进行分析，它在每次迭代中只需要存储两个哈希值。

算法 5.9

小空间生日攻击 (A small-space birthday attack)

输入：哈希函数 $H : \{0,1\}^* \to \{0,1\}^\ell$

输出：不同的 x, x' 使得 $H(x) = H(x')$

$x_0 \leftarrow \{0,1\}^{\ell+1}$

$x' := x := x_0$

for $i = 1, 2, \ldots$ **do**:

  $x := H(x)$

  $x' := H(H(x'))$

  //现在 $x = H^{(i)}(x_0)$ 且 $x' = H^{(2i)}(x_0)$

  if $x = x'$ **break**

$x' := x, x := x_0$

for $j = 1$ **to** i:

  if $H(x) = H(x')$ **return** x, x' **and halt**

  else $x := H(x), x' := H(x')$

//现在 $x = H^{(j)}(x_0)$ 且 $x' = H^{(i+j)}(x_0)$

我们期望在 $x' = x$ 之前发生多少次第一循环的迭代？考虑值序列 $x_1, x_2, \ldots$，其中 $x_i = H^{(i)}(x_0)$ 如前所定义。如果我们将 H 建模为随机函数，那么这些值中的每一个都均匀且独立地分布在 $\{0,1\}^\ell$ 中，直到第一次重复发生。因此，我们期望在序列的前 $q = \Theta(2^{\ell/2})$ 项中发生重复。我们证明，当重复发生在第一个 q 元素中时，该算法在最多 q 次第一循环的迭代中找到重复：

断言 5.10 设 $x_1, \ldots, x_q$ 是一个值序列，其中 $x_m = H(x_{m-1})$。如果存在 $1 \le I < J \le q$ 使得 $x_I = x_J$，那么存在 $i < J$ 使得 $x_i = x_{2i}$。

证明 序列 $x_I, x_{I+1}, \ldots$ 以周期 $\Delta \stackrel{\text{def}}{=} J - I$ 重复。也就是说，对于所有 $i \ge I$ 和 $k \ge 0$，都有 $x_i = x_{i+k\cdot\Delta}$。设 i 是 Δ 的最小倍数，并且也大于或等于 I。我们有 $i < J$，因为 Δ 的值序列 $I, I+1, \ldots, I+(\Delta-1) = J-1$ 包含 Δ 的倍数。由于 $i \ge I$ 且 $2i - i = i$ 是 Δ 的倍数，因此 $x_i = x_{2i}$。

因此，如果序列 $x_1, \ldots, x_q$ 中存在重复值，那么存在某个 $i < q$ 使得 $x_i = x_{2i}$。但在我们的算法的第 i 次迭代中，我们有 $x = x'$，并且算法跳出了第一个循环。在算法的那个点上，我们知道 $x_i = x_{i+i}$。然后算法设置 $x' := x(= x_i)$ 和 $x := x_0$，并继续寻找使得 $x_j = x_{j+i}$ 的最小 $j \ge 0$。（注意 $j = 0$ 是不可能的，因为 $|x_0| = \ell + 1$。）它输出 x_{j-1}, x_{j+i-1} 作为碰撞。

寻找有意义的碰撞。 刚才描述的算法可能看起来不适合寻找有意义的碰撞，因为它无法控制采样的元素。然而，我们证明了寻找有意义的碰撞是可能的。诀窍是在*正确的函数*中寻找碰撞！

假设如前所述，爱丽丝希望在两种不同“类型”的消息之间找到碰撞，例如，一封信解释爱丽丝为何被解雇，另一封信是赞美的推荐信，这两封信的哈希值相同。然后，爱丽丝编写每条消息，使其每条消息中都有 $\ell - 1$ 个可互换的单词；即每种类型都有 $2^{\ell-1}$ 条消息。定义**单射函数** g：$\{0,1\}^\ell \to \{0,1\}^*$，使得输入的第 ℓ 个比特选择类型 0 或类型 1 的消息，并且第 i 个比特（对于 $1 \le i \le \ell - 1$）选择适当类型的消息中第 i 个可互换单词的选项。例如，考虑以下句子：

0：Bob is a **good/hardworking** and **honest/trustworthy worker/employee**.
1：Bob is a **difficult/problematic** and **taxing/irritating worker/employee**.

定义一个函数 g 接受 4 比特输入，其中最后一个比特决定输出句子的类型，前三个比特决定该句子中单词的选择。例如：

$g(0000) =$ Bob is a good and honest worker. $g(0001) =$ Bob is a difficult and taxing worker. $g(1010) =$ Bob is a hardworking and honest employee. $g(1011) =$ Bob is a problematic and taxing employee.

现在定义 $f : \{0,1\}^\ell \to \{0,1\}^\ell$ 为 $f(x) \stackrel{\text{def}}{=} H(g(x))$。爱丽丝可以使用前面展示的小空间生日攻击在 f 中找到碰撞。这里的关键是 f 中的任何碰撞 x, x' 都会产生两个消息 $g(x), g(x')$ 在 H 下发生碰撞。如果 x, x' 是随机碰撞，那么我们期望碰撞消息 $g(x), g(x')$ 有 $1/2$ 的概率属于不同类型（因为 x 和 x' 在最后一个比特上以该概率不同）。如果碰撞消息不属于不同类型，可以从头开始重复该过程。

5.4.3 *求函数逆的时间/空间权衡 (Time/Space Tradeoffs for Inverting Functions)

在本节中，我们考虑**原像抗性**问题，即我们对**函数求逆**的算法感兴趣。在这里，算法给定 $y = H(x)$（x 是均匀的）作为输入，目标是找到任何 x' 使得 $H(x') = y$。我们首先假设 H 的输入和输出长度相等，并在最后简要考虑更一般的情况。

设 $H : \{0,1\}^\ell \to \{0,1\}^\ell$ 是一个函数。在不利用 H 的任何弱点的情况下，可以通过对域进行穷举搜索，在 $O(2^\ell)$ 时间内找到 y 的原像。我们证明，通过大量的预处理和相对较大的内存，可以做得更好。

需要明确的是：我们把预处理看作是一次性操作，并且不会过分关注它的成本。相反，我们关注的是在预处理完成后，在**线 (on-line)** 上反转 H 在点 y 处所需的时间。如果预处理的成本可以通过反转许多点来摊销，或者如果我们愿意在知道 y 之前投入计算资源进行预处理，以便之后更快地进行反转，则这是合理的。

事实上，使用预处理可以在很少的时间内实现函数反转是微不足道的。我们需要做的就是在预处理阶段在每个点上评估 H，然后将对 $\{(x, H(x))\}$ 存储在一个表中，并按它们的第二个条目排序。在接收到任何点 y 后，可以通过在表中搜索第二个条目为 y 的对来轻松找到 y 的原像。这里的缺点是我们需要为表中的 $O(2^\ell)$ 对分配空间，这对于大的 ℓ 来说是令人望而却步的（例如，$\ell = 80$）。

最初的暴力攻击使用恒定的内存和 $O(2^\ell)$ 时间，而刚才描述的攻击存储 $O(2^\ell)$ 个点，并以基本上恒定的时间启用反转。我们现在提出一种新的方法，允许攻击者进行**时间/空间权衡**。具体来说，我们展示了如何存储 $O(2^{2\ell/3})$ 个点，并在 $O(2^{2\ell/3})$ 时间内找到原像；其他权衡也是可能的。

热身 (A warmup)。 我们首先考虑一个简单的情况，即函数 H 定义了一个**循环 (cycle)**，这意味着 $x, H(x), H(H(x)), \ldots$ 覆盖了 $\{0,1\}^\ell$ 的所有点（注意大多数函数不定义循环，但我们假设如此，以在一个非常简单的情况下演示该想法）。为清楚起见，设 $N = 2^\ell$ 表示域的大小。

在预处理阶段，攻击者简单地穷尽整个循环，从一个任意的起点 x_0 开始，计算 $x_1 := H(x_0), x_2 := H(H(x_0))$, 直到 $x_N = H^{(N)}(x_0)$，其中 $H^{(i)}$ 指的是 H 的 i 次评估。设 $x_i \stackrel{\text{def}}{=} H^{(i)}(x_0)$。我们设想将循环划分为 $\sqrt{N}$ 个长度为 $\sqrt{N}$ 的段，并让攻击者存储每段的起点和终点。也就是说，攻击者将形式为 $\left(x_{i\cdot\sqrt{N}}, x_{(i+1)\cdot\sqrt{N}}\right)$ 的对存储在一个表中，对于 $i = 0$ 到 $\sqrt{N} - 1$，按每对的第二个分量排序。所得表包含 $O(\sqrt{N})$ 个点。

当攻击者在在线阶段获得要反转的点 y 时，它检查 $y, H(y), H^{(2)}(y), \ldots$ 中哪个对应于某个段的终点。（每次检查仅涉及对存储的对进行表查找。）由于 y 位于某个段中，因此保证在 $\sqrt{N}$ 步内会遇到一个终点。一旦识别出终点 $x = x_{(i+1)\cdot\sqrt{N}}$，攻击者就会获取相应段的起点 $x' = x_{i\cdot\sqrt{N}}$，并计算 $H(x'), H^{(2)}(x'), \ldots$ 直到到达 y；这立即给出了所需的原像。请注意，这需要最多 $O(\sqrt{N})$ 次 H 评估。

总之，这种攻击存储了 $O(\sqrt{N})$ 个点，并使用 $O(\sqrt{N})$ 次哈希计算以 1 的概率找到原像。

Hellman 的时间/空间权衡 (Hellman's time/space tradeoff)。 Martin Hellman 引入了一种更通用的时间/空间权衡方法，适用于任意函数 H（尽管分析将 H 视为随机函数）。Hellman 的攻击仍然存储几个段的起点和终点，但在这方面，这些段是“独立的”，而不是一个大循环的一部分。更详细地说：设 s, t 是我们稍后将设置的参数。攻击首先选择 s 个均匀的起点 $\mathrm{SP}_1, \ldots, \mathrm{SP}_s \in \{0,1\}^\ell$。对于每个这样的点 SP_i，它计算一个相应的终点 $\mathrm{EP}_i := H^{(t)}(\mathrm{SP}_i)$，使用 H 的 t 次应用。（参见图 5.3。）然后，攻击者将这些值 $\{(\mathrm{SP}_i, \mathrm{EP}_i)\}_{i=1}^s$ 存储在一个表中，按每对的第二个条目排序。

在接收到要反转的值 y 后，攻击以类似于前面讨论的简单情况进行。具体来说，它检查 $y, H(y), \ldots, H^{(t-1)}(y)$ 中是否有任何一个等于某个段的终点（一旦找到第一个匹配就停止）。这些值中可能没有一个等于终点（正如我们将在下面讨论的那样）。然而，如果 $H^{(j)}(y) = \mathrm{EP}_i = H^{(t)}(\mathrm{SP}_i)$ 对于某些 i, j 成立，那么攻击者计算 $H^{(t-j-1)}(\mathrm{SP}_i)$ 并检查它是否是 y 的原像。整个过程最多需要 t 次 H 评估。

这似乎是有效的，但我们忽略了一些微妙之处。首先，可能 $y, H(y), \ldots, H^{(t-1)}(y)$ 中没有一个终点。如果 y 不在最初生成表格时获得的 $s \cdot t$ 个值（不包括起点）的集合中，就会发生这种情况。我们可以设置 $s \cdot t \geq N$ 以尝试将每个 ℓ 比特字符串包含在表中，但这并不能解决问题，因为表本身中可能会发生碰撞——事实上，对于 $s \cdot t \geq N^{1/2}$，我们对生日问题的先前分

析告诉我们碰撞是很有可能的——这将减少集合中不同值的数量。第二个问题，即使 y 在表中，也会出现，即即使我们找到了匹配的终点，并且 $H^{(j)}(y) = \mathrm{EP}_i = H^{(t)}(\mathrm{SP}_i)$ 对于某些 i, j 成立，也不能保证 $H^{(t-j-1)}(\mathrm{SP}_i)$ 是 y 的原像。这里的问是 $y, H(y), \ldots, H^{(t-1)}(y)$ 这个段本身可能与第 i 个段发生碰撞，即使 y 本身不位于该段中；参见图 5.4。（即使 y 位于某个段中，第一个匹配的终点也可能不在该段中。）我们称之为**假阳性 (false positive)**。有人可能认为如果 H 是抗碰撞的，这种情况就不太可能发生；然而，我们正在处理一个涉及超过 $\sqrt{N}$ 个点的场景，因此碰撞实际上变得很有可能。

图 5.4：在线阶段的碰撞 (Colliding in the on-line phase)。

假阳性问题可以通过修改算法来解决，使其**始终**计算整个序列 $y, H(y), \ldots, H^{(t-1)}(y)$，并检查对于使得 $H^{(j)}(y) = \mathrm{EP}_i$ 的每个 i, j 是否 $H^{(t-j-1)}(\mathrm{SP}_i)$ 是 y 的原像。只要 y 在预处理阶段生成的值（不包括终点）集合中，就保证找到一个原像。现在一个令人担忧的问题是算法的运行时间可能会增加，因为每个假阳性都会引入额外的 $O(t)$ 次哈希评估。可以证明，假阳性的期望数量是 $O(st^2/N)$。（序列 $y, H(y), \ldots, H^{(t-1)}(y)$ 中有 t 个值，表中有最多 st 个不同的点。将 H 视为随机函数，序列中的任何点等于表中某个点的概率是 $1/N$。因此，假阳性的期望数量是 $t \cdot st \cdot 1/N = st^2/N$。）因此，只要 $st^2 \approx N$，（出于下面将介绍的其他原因，我们将确保 $st^2 \approx N$），处理假阳性预计只需要 $O(t)$ 次额外的哈希计算。

在给定上述修改后，反转 $y = H(x)$ 的概率至少是 x 位于预处理阶段生成（不包括终点）的点集合中的概率。我们现在给出该概率的下界，该概率是针对预处理阶段的随机性以及 x 的均匀选择计算的，将 H 视为随机函数进行分析。我们首先计算表中的不同点的期望数量。考虑生成表的第 i 行时发生的情况。起点 SP_i 是均匀的，表中已有的不同点（不包括终点）最多有 $(i-1) \cdot t$ 个，因此 SP_i 是“新”的（即不等于任何先前的值）概率至少为 $1 - (i-1) \cdot t/N$。$H(\mathrm{SP}_i)$ 是新的概率是多少？如果 SP_i 是新的，那么 $H(\mathrm{SP}_i)$ 是均匀的（因为我们把 H 视为随机函数），因此是新的概率至少为 $1 - ((i-1) \cdot t + 1)/N$。（我们现在有额外的点 SP_i。）因此， $H(\mathrm{SP}_i)$ 是新的概率至少为

$$\Pr[\mathrm{SP}_i \text{ is new}] \cdot \Pr[H(\mathrm{SP}_i) \text{ is new} \mid \mathrm{SP}_i \text{ is new}] \geq \left(1 - \frac{(i-1) \cdot t}{N}\right) \cdot \left(1 - \frac{(i-1) \cdot t + 1}{N}\right)$$

以这种方式继续， $H^{(t-1)}(\mathrm{SP}_i)$ 是新的概率至少为

$$\left(1 - \frac{i \cdot t}{N}\right)^t = \left(1 - \frac{i \cdot t}{N}\right)^{\frac{N}{i \cdot t} \cdot \frac{i \cdot t^2}{N}} \approx e^{-it^2/N}.$$

这里要注意的是，当 $it^2 \leq N/2$ 时，这个概率至少为 $1/2$；另一方面，一旦 $it^2 > N$，这个概率就会相当小。考虑最后一行，当 $i = s$ 时，这意味着如果 $st^2 > N$ 时，我们将不会获得

太多的额外覆盖。因此，参数的一个好的设置是 $st^2 = N/2$。假设如此，表中不同点的期望数量是

$$\sum_{i=1}^{s}\sum_{j=0}^{t-1}\Pr[H^{(j)}(\mathrm{SP}_i)\text{ is new}] \geq \sum_{i=1}^{s}\sum_{j=0}^{t-1} 1/2 = \frac{st}{2}.$$

x 被“覆盖”的概率是至少 $\frac{st}{2N} = \frac{1}{4t}$。

这提供了一个**弱**时间/空间权衡，其中我们可以使用更多的空间（因此需要更少的时间）来减少反转 y 的概率。但我们可以通过生成 $T = 4t$ 个“独立”表来做得更好。（这使空间和时间都增加了 T 的因子。）只要我们可以将 x 位于每个相关表中的概率视为独立，那么至少有一个表包含 x 的概率是

$$1 - \Pr[\text{no table contains } x] = 1 - \left(1 - \frac{1}{4t}\right)^{4t} \approx 1 - e^{-1} = 0.63.$$

剩下的唯一问题是如何生成一个独立表。（请注意，像以前一样生成表与向原始表中添加 s 个额外的行是相同的，这没有帮助。）我们可以通过在 H 的每次评估之后应用某个函数 F_i 来实现这一点，其中 $F_1, \ldots, F_T$ 都是不同的。（一个好的选择可能是设置 $F_i(x) = x \oplus c_i$ 对于某个固定的常数 c_i，每个表都不同。）设 $H_i \stackrel{\text{def}}{=} F_i \circ H$，即 $H_i(x) = F_i(H(x))$。然后对于第 i 个表，我们再次选择 s 个随机起点 SP，但对于每个这样的点，我们现在计算 $H_i(\mathrm{SP}), H_i^{(2)}(\mathrm{SP}), \ldots$ 等。在接收到要反转的值 $y = H(x)$ 后，攻击者首先计算 $y' = F_i(y)$，然后检查 $y', H_i(y'), \ldots, H_i^{(t-1)}(y')$ 中是否有任何一个对应于第 i 个表中的终点；对 $i = 1, \ldots, T$ 重复此操作。（我们省略了进一步的细节。）虽然很难正式论证独立性，但在实践中，这种方法会带来良好的结果。

选择参数。 总结上述讨论，我们看到只要 $st^2 = N/2$，我们就有了一个算法，它在预处理阶段存储 $O(s \cdot T) = O(s \cdot t) = O(N/t)$ 个点，然后可以在 $O(t \cdot T) = O(t^2)$ 时间内以恒定的概率反转 y。一个参数设置是 $t = N^{1/3} = 2^{\ell/3}$，在这种情况下，我们有一个算法存储 $O(2^{2\ell/3})$ 个点，并使用 $O(2^{2\ell/3})$ 次哈希计算找到原像。如果使用输出为 80 比特的哈希函数，那么这在实践中是可行的。

处理不同的域和范围 (Handling different domain and range)。 在实践中，经常会遇到 H 的域和范围不同的情况。一个例子是在密码破解的背景下（参见 5.6.3 节），其中攻击者拥有 $H(\mathrm{pw})$ 但 $|\mathrm{pw}| \ll \ell$。在一般情况下，假设 x 是从某个域 $\mathcal{D}$ 中选择的，该域可能大于或小于 $\{0,1\}^\ell$。虽然通过人为地扩展域/范围以使其匹配是可能的，但这对于上述攻击没有用处。要了解原因，请考虑密码示例。为了使攻击成功，我们希望 pw 在预处理期间生成的某个

表格值中（可能只有 SP 本身除外）。如果我们通过简单地计算 $H(\mathrm{SP}), H^{(2)}(\mathrm{SP}), \ldots$ 来生成表的每一行，对于 $\mathrm{SP} \in \mathcal{D}$，那么这些值中没有一个（除了 SP 本身之外）会等于 pw。

我们可以通过像以前一样在 H 的每次评估之间应用一个函数 F_i 来解决这个问题，尽管现在我们选择 F_i 将 $\{0,1\}^{\ell}$ 映射到 $\mathcal{D}$。这解决了上述问题，因为 $F_i(H(\mathrm{SP})), (F_i \circ H)^{(2)}(\mathrm{SP}), \ldots$ 现在都位于 $\mathcal{D}$ 中。

对密钥恢复攻击的应用 (Applications to key-recovery attacks)。 时间/空间权衡提供了对加密原语（除了哈希函数之外）的攻击。Hellman 最初考虑的典型应用——事实上，是应用——是对任意块密码 F 的攻击，导致密钥恢复。定义 $H(k) \stackrel{\text{def}}{=} F_k(x)$，其中 x 是用于构建表格的某个任意但固定的输入。如果攻击者可以获得未知密钥 k 的 $F_k(x)$——无论是通过选择明文攻击还是通过选择 x 使得 $F_k(x)$ 很可能在已知明文攻击中获得——那么通过反转 H，攻击者就可以了解（k 的候选值）。请注意， F 的密钥长度可能与其块长度不同，但在这种情况下，我们可以使用刚才描述的处理不同域和范围的 H 的技术。

5.5 随机预言机模型 (The Random-Oracle Model)

有几个基于密码学哈希函数构建的构造，仅基于哈希函数是抗碰撞或原像抗性的假设，无法证明它们的安全性。（我们将在下一节中看到一些例子。）在许多情况下，似乎没有一个简单且合理的关于哈希函数的假设足以证明该构造是安全的。

面对这种情况，有几种选择。一种是寻找可以被证明安全，基于底层哈希函数的某些合理假设的方案。这是一种很好的方法，但它留下了在找到此类方案之前该怎么办的问题。此外，可证明安全的构造可能比其他尚未被证明安全的方法效率低得多。（这是一个突出的问题，我们将在公钥密码学设置中遇到。）

当然，另一种可能性是使用现有的密码系统，即使其安全性没有任何理由，除了可能设计者试图攻击它但没有成功这一事实。这与我们所说的关于密码学严谨、现代方法的重要性背道而驰，应该清楚这是不可接受的。

一种在实践中非常成功的方法，并且在完全严谨的安全性证明和完全没有证明之间提供了一个“中间立场”，是引入一个理想化的模型来证明密码学方案的安全性。虽然理想化可能不是对现实的准确反映，但我们至少可以从理想化模型中的证明中推导出对方案设计稳健性的一些衡量。只要模型合理，这样的证明肯定比完全没有证明要好。

这种方法最流行的例子是**随机预言机模型 (random-oracle model)**，它将密码学哈希函数

H 视为一个**真正的随机函数 (truly random function)**。（尽管我们在讨论时间/空间权衡时已经看到了一个例子，但在那里我们是在分析攻击而不是构造。）更具体地说，随机预言机模型假设存在一个公共的随机函数 H，只能通过“查询”一个预言机来评估它——可以将其视为一个“黑匣子”——它在给定输入 x 时返回 $H(x)$。（我们将在下一节中讨论如何解释这一点。）为了区分事物，我们迄今为止使用的模型（其中不存在随机预言机）通常被称为“**标准模型 (standard model)**”。

没有人声称随机预言机存在，尽管有人建议可以在实践中使用受信任方（即，互联网上的某个服务器）来实现随机预言机。相反，随机预言机模型提供了一种正式的方法论，可用于设计和验证密码学方案，采用以下两步方法：

1. **首先，在随机预言机模型中设计和证明方案是安全的。** 也就是说，我们假设世界包含一个随机预言机，并在该模型中构造和分析一个密码学方案。到目前为止我们已经看到的标准密码学假设也可以用于安全性证明。
2. **当我们想要在现实世界中实现该方案时，随机预言机是不可用的。** 相反，随机预言机被实例化为一个设计适当的密码学哈希函数 $\hat{H}$。（我们将在本节末尾回到这一点。）也就是说，在方案规定一方应该查询预言机以获取 $H(x)$ 的值的每一点上，该方改为**自己计算** $\hat{H}(x)$。

希望在第二步中使用的密码学哈希函数“足够好”，能够模拟随机预言机，使得第一步中给出的安全性证明可以延续到该方案的现实世界实例化。这里的困难在于，这种希望没有理论上的理由，事实上存在（人造的）方案，这些方案可以在随机预言机模型中被证明是安全的，但在第二步中无论随机预言机如何实例化都是**不安全**的。此外，（无论是数学上还是启发式地）都不清楚哈希函数“足够好”以模拟随机预言机意味着什么，也不清楚这是一个可实现的目标。特别是，任何具体的实例化 $\hat{H}$ 永远不能表现得像随机函数，因为 $\hat{H}$ 是确定性且固定的。由于这些原因，随机预言机模型中的安全性证明应被视为提供该方案**设计没有“固有缺陷”** 的证据，但**不是**任何现实世界实例化都是安全的**严格证明**。关于如何解释随机预言机模型中的证明的进一步讨论在 5.5.2 节中给出。

5.5.1 随机预言机模型的细节 (The Random-Oracle Model in Detail)

在继续之前，让我们精确地确定随机预言机模型包含什么。思考随机预言机模型的一个好方法如下：**“预言机”** 只是一个接受二进制字符串作为输入并返回二进制字符串作为输出的“黑匣子”。黑匣子的内部工作原理是未知的、不可穿透的。每个人——诚实方以及攻击者——都可以与该黑匣子进行交互，这种交互包括将二进制字符串 x 作为输入，并接收二进制字符串 y 作为输出；

我们将其称为**查询预言机** x，并将 x 本身称为**对预言机进行的查询**。对预言机的查询被认为是**私有**的，因此如果某方查询预言机输入 x，则没有其他人知道 x，甚至不知道该方查询了预言机。这是有道理的，因为对预言机的调用对应于（在现实世界实例化中）对密码学哈希函数的本地评估。

这个“黑匣子”的一个重要属性是它**一致**。也就是说，如果黑匣子曾经对特定输入 x 输出 y，那么当再次给定相同的输入 x 时，它始终输出相同的答案 y。这意味着我们可以将黑匣子视为实现了定义良好的函数 H；即，我们根据黑匣子的输入/输出特性来定义函数 H。为方便起见，我们因此说“查询 H”而不是查询黑匣子。没有人“知道”整个函数 H（除了黑匣子本身）；充其量，所知道的只是迄今为止明确查询过的字符串上的 H 值。

我们已经在第 3 章中讨论了选择随机函数 H 的含义。我们在这里只重申，思考 H 的均匀选择有两种等价的方式：要么将 H 想象为在某个指定域和范围上的所有函数集合中“一次性”均匀选择，要么想象在需要时“动态”生成 H 的输出。具体来说，在第二种情况下，我们可以将函数视为由最初为空的表定义。当预言机收到查询 x 时，它首先检查 x 是否等于表中的某个对 (x_i, y_i)；如果是，则返回相应的 y_i 值。否则，选择一个均匀字符串 $y \in \{0,1\}^\ell$（对于某个指定的 ℓ），返回答案 y，并将 (x, y) 存储在表中。第二个观点通常在概念上更容易推理，并且如果 H 在无限域（例如 $\{0,1\}^*$）上定义，则在技术上也更容易处理。

当我们在 3.5.1 节中定义**伪随机函数**时，我们也考虑了具有预言机访问随机函数的算法。为了避免混淆，我们注意到在这里使用随机函数与在那里使用随机函数非常不同。在那里，随机函数被用作定义（具体）带密钥函数是伪随机的含义的一种方式。相比之下，在随机预言机模型中，随机函数被用作构造本身的一部分，如果我们想要构造的**具体实现**在现实世界中，则必须以某种方式实例化。伪随机函数不是随机预言机，因为它只有在**密钥是秘密**的情况下才是伪随机的。然而，在随机预言机模型中，所有方都需要能够计算该函数；因此不能有秘密密钥。

5.5.2 随机预言机方法论是否合理？ (Is the Random-Oracle Methodology Sound?)

在现实世界中，在随机预言机模型中设计的方案是通过某个具体函数来实例化 H 的。有了随机预言机模型的机制，我们转向一个更基本的问题：

在随机预言机模型中的安全性证明对任何现实世界实例化能提供多大保证？

这个问题没有明确的答案：目前在密码学社区内，对于如何解释随机预言机模型中的证明，以及在现实世界中随机预言机实例化意味着什么，存在争论，并且这是一个活跃的研究领域。我们只能尝试给出双方观点的概貌。

对随机预言机模型的反对意见 (Objections to the random-oracle model)。 反对使用随机预言机进行论证的起点很简单：正如我们已经指出的，没有任何具体的哈希函数可以充当“真正”的随机预言机。稍加思考就会发现，任何具体的哈希函数都无法充当“真正”的随机预言机。例如，在随机预言机模型中，$H(x)$ 的值是“完全随机的”，如果 x 没有被明确查询。对应的要求是 $\hat{H}(x)$ 必须是随机的（或伪随机的），如果 $\hat{H}$ 没有被明确评估在 x 上。我们如何在现实世界中解释“明确评估”的含义？如果攻击者知道计算 $\hat{H}$ 的某个捷径，而这不涉及运行 $\hat{H}$ 的实际代码，那么“明确评估”又意味着什么呢？此外，$\hat{H}(x)$ 不可能是随机的（甚至不是伪随机的），因为一旦攻击者了解了 $\hat{H}$ 的描述，该函数在所有输入上的值就立即确定了。

当我们检查前面介绍的证明技术时，随机预言机模型的局限性变得更加清晰。回想一下，一种证明技术是利用一个归约可以“看到”攻击者 $\mathcal{A}$ 对随机预言机发出的查询这一事实。如果我们将随机预言机替换为一个特定的哈希函数 $\hat{H}$，这意味着我们必须在实验开始时向攻击者提供 $\hat{H}$ 的描述。但随后 $\mathcal{A}$ 可以自己评估 $\hat{H}$，而无需进行任何显式查询，因此归约将不再具有“看到” $\mathcal{A}$ 进行的任何查询的能力。（事实上，如前所述，$\mathcal{A}$ 执行显式评估 $\hat{H}$ 的概念可能不真实，并且肯定无法正式定义。）同样，随机预言机模型中的安全性证明允许归约方根据自己的意愿选择 H 的输出，这是使用具体函数时显然不可能的。

即使我们愿意忽略上述理论问题，一个实际问题是我们目前对“足够好”地实例化随机预言机的具体哈希函数意味着什么知之甚少。具体来说，假设我们想使用 SHA-1 的某种适当修改来实例化随机预言机（SHA-1 是 6.3.3 节中讨论的密码学哈希函数）。虽然对于某些特定方案 Π，假设使用 SHA-1 实例化 Π 是安全的可能是合理的，但假设 SHA-1 可以在随机预言机模型中设计的每个方案中替代随机预言机就**很不**合理了。事实上，正如我们之前所说，我们知道 SHA-1 不是随机预言机。而且，设计一个在随机预言机模型中安全，但在随机预言机被 SHA-1 替换时**不安全**的方案并不困难。

我们强调，形式为“SHA-1 的行为类似于随机预言机”的假设在性质上不同于“SHA-1 是抗碰撞的”或“AES 是伪随机函数”等假设。问题部分在于，对于前一个陈述，没有令人满意的定义来定义它，而我们对后两个陈述有这样的定义。

因此，与例如引入新的密码学假设以在标准模型中证明方案安全相比，使用随机预言机模型证明方案的安全性在性质上是不同的。因此，随机预言机模型中的安全性证明不如标准模型中的安全性证明令人满意。

对随机预言机模型的支持意见 (Support for the random-oracle model)。 考虑到随机预言机模型的所有问题，为什么要使用它呢？更重要的是：为什么随机预言机模型在现代密码学的发展（尤其是密码学当前的实际应用）中如此具有影响力，以及为什么它仍然被广泛使用？正如我们将看到的，随机预言机模型使得设计比我们在标准模型中已知构造的方案**显著更高效**。因

此，今天使用的（如果有的话）公钥密码系统很少有在标准模型中的安全性证明，而有大量的已部署方案在随机预言机模型中有安全性证明。此外，随机预言机模型中的证明几乎被普遍认为是为其标准化考虑的方案的安全性的信心来源。

这样做的根本原因是基于以下信念：

随机预言机模型中的安全性证明显著优于完全没有证明。

尽管有些人不同意，但我们提供以下支持这一论断的理由：

- 随机预言机模型中某个方案的安全性证明表明该方案的设计是“稳健的”，因为对该方案的现实世界实例化的唯一可能攻击是源于用于实例化随机预言机的哈希函数中的弱点。因此，如果使用一个“足够好”的哈希函数来实例化随机预言机，我们应该对方案的安全性有信心。此外，如果某个给定的实例化被成功攻击，我们可以简单地用一个“更好”的哈希函数替换正在使用的哈希函数。
- 重要的是，迄今为止，在随机预言机模型中被证明安全的方案（当随机预言机被正确实例化时）**尚未**发生成功的现实世界攻击。（我们在此不包括对“人造”方案的攻击，但请注意，在实例化随机预言机时必须非常小心，如练习 5.10 所指出的。）这为随机预言机模型在设计实用方案中的有用性提供了证据。

然而，上述内容最终仅代表对随机预言机模型中证明的有用性的直观猜测，并且——在所有条件相同的情况下——**没有**随机预言机的证明更受青睐。

实例化随机预言机 (Instantiating the Random Oracle)

正确实例化随机预言机是微妙的，全面的讨论超出了本书的范围。在这里，我们只提醒读者，**不加修改地**使用“现成的”密码学哈希函数，一般来说，在原则上并不是一种稳健的方法。首先，大多数密码学哈希函数是使用 Merkle–Damgård 范式构建的（参见 5.2 节），当允许变长输入时，可以很容易地将其与随机预言机区分开来。（参见练习 5.10。）此外，在某些构造中，随机预言机的输出需要位于某个特定范围（例如，预言机应该输出某个组的元素），这会导致额外的复杂性。

5.6 哈希函数的其他应用 (Additional Applications of Hash Functions)

我们以简要讨论密码学哈希函数在密码学和计算机安全中的一些其他应用来结束本章。

5.6.1 指纹识别和去重 (Fingerprinting and Deduplication)

在使用抗碰撞哈希函数 H 时，文件**哈希值**（或**摘要**）可作为该文件的唯一标识符。（如果发现任何其他文件具有相同的标识符，则意味着 H 中发生了碰撞。）文件 x 的哈希值 $H(x)$ 就像**指纹 (fingerprint)**，可以通过比较它们的摘要来检查两个文件是否相等。这个简单的想法有许多应用。

- **病毒指纹识别 (Virus fingerprinting)：** 病毒扫描程序识别病毒并阻止或隔离它们。实现此目标最基本的步骤之一是存储包含已知病毒哈希值的数据库，然后查找下载的应用程序或电子邮件附件的哈希值是否在该数据库中。由于每个病毒只需要记录（和/或分发）一个短字符串，因此所涉及的开销是可行的。
- **去重 (Deduplication)：** 数据去重用于消除重复的数据副本，尤其是在**云存储**环境中，其中多个用户依赖单个云服务来存储其数据。这里的观察结果是，如果多个用户希望存储相同的文件（例如，流行的视频），则该文件只需要存储一次，无需每个用户单独上传。去重可以通过以下方式实现：首先让用户上传他们想要存储的新文件的哈希值；如果具有此哈希值的文件已存储在云中，则云存储提供商可以简单地向现有文件添加一个指针，表明该特定用户也存储了该文件。这节省了通信和存储，并且该方法稳健性来自哈希函数的抗碰撞性。
- **点对点 (P2P) 文件共享 (Peer-to-peer (P2P) file sharing)：** 在 P2P 文件共享系统中，服务器保留表格以提供文件查找服务。这些表格包含可用文件的哈希值，再次提供了唯一标识符，而无需使用大量内存。

一个短小的摘要可以唯一地标识世界上的每个文件，这可能令人惊讶。但这正是抗碰撞哈希函数提供的保证，这使得它们在上述设置中非常有用。

5.6.2 Merkle 树 (Merkle Trees)

考虑一个客户端将文件 x 上传到服务器。当客户端稍后检索 x 时，它希望确保服务器返回原始的、未修改的文件。客户端可以本地存储 x 并检查检索到的文件是否等于 x，但这违背了最初使用服务器的目的。我们正在寻找一个解决方案，其中客户端的存储量很小。

一个自然的解决方案是使用前面描述的“指纹识别”方法。客户端可以本地存储短摘要 $h := H(x)$；当服务器返回一个候选文件 x' 时，客户端只需检查 $H(x') = h$ 即可。

如果我们要将此解决方案扩展到多个文件 $x_1, \ldots, x_t$ 会怎样？有两种显而易见的方法。一种是简单地独立地哈希每个文件；客户端将在本地存储摘要 $h_1, \ldots, h_t$，并像以前一样验证检索到的文件。这种方法的缺点是客户端的存储量随 t 线性增长。另一种可能性是将所有文件哈希在一起。也就是说，客户端可以计算 $h := H(x_1, \ldots, x_t)$ 并仅存储 h。现在的缺点是，当客户端想要检索和验证第 i 个文件 x_i 的正确性时，它需要检索**所有**文件以便重新计算摘要。

由 Ralph Merkle 引入的 **Merkle 树 (Merkle trees)** 在这些极端之间提供了一种权衡。对输入值 $x_1, \ldots, x_t$ 计算的 Merkle 树只是一个深度为 $\log t$ 的二叉树，其中输入放置在叶子处，每个内部节点的值是其两个子节点值的哈希值；参见图 5.5。（我们假设 t 是 2 的幂；如果不是，那么我们可以将某些输入值固定为 null 或使用不完整的二叉树，具体取决于应用程序。）

图 5.5：Merkle 树。

固定某个哈希函数 H，我们将 MT_t 表示为接受 t 个输入值 $x_1, \ldots, x_t$，计算结果 Merkle 树并输出树根值的函数。（带密钥的哈希函数以显而易见的方式产生带密钥的函数 MT_t。）我们有：

定理 5.11 设 (Gen_H, H) 是抗碰撞的。那么对于任何固定的 t，$(\text{Gen}_H, \text{MT}_t)$ 也是抗碰撞的。

因此，Merkle 树为实现抗碰撞哈希函数的域扩展提供了 Merkle–Damgård 变换的替代方案。（然而，如前所述，如果允许输入值的数量 t 变化，则 Merkle 树不是抗碰撞的。）

Merkle 树为我们最初的问题提供了一个有效的解决方案，因为它允许使用 $O(\log t)$ 的通信来验证任何原始 t 个输入中的任何一个。客户端计算 $h := \text{MT}_t(x_1, \ldots, x_t)$，将 $x_1, \ldots, x_t$ 上传到服务器，并在本地存储 h（以及文件数 t）。当客户端检索第 i 个文件 x_i 时，服务器将 x_i 以及证明 π_i（证明这是正确值）一起发送。该证明由 Merkle 树中**邻近**从 x_i 到根的路径的节点的值组成。客户端可以从这些值重新计算根的值，并验证它是否等于存储的值 h。例如，考虑图 5.5 中的 Merkle 树。客户端计算 $h_{1\ldots8} := \text{MT}_8(x_1, \ldots, x_8)$，将 $x_1, \ldots, x_8$ 上传到服务器，并在本地存储 $h_{1\ldots8}$。当客户端检索 x_3 时，服务器将 x_3 以及 $x_4, h_{1\ldots2} = H(x_1, x_2)$ 和 $h_{5\ldots8} = H(H(x_5, x_6), H(x_7, x_8))$ 一起发送。（如果文件很大，我们可能希望避免发送除客户端请求的文件之外的任何文件。如果我们将 Merkle 树定义在文件**哈希值**而不是文件本身之上，就可以轻松做到。我们省略了细节。）客户端计算 $h'_{1\ldots4} := H(h_{1\ldots2}, H(x_3, x_4))$ 和 $h'_{1\ldots8} := H(h'_{1\ldots4}, h_{5\ldots8})$，然后验证 $h_{1\ldots8} = h'_{1\ldots8}$。

如果 H 是抗碰撞的，那么服务器发送一个不正确文件（以及任何证明）会导致验证成功的行为是不可行的。使用这种方法，客户端的本地存储是**常数**（与文件数量 t 无关），并且从服务器到客户端的通信量与 $\log t$ 成比例。

5.6.3 密码哈希 (Password Hashing)

哈希函数在计算机安全中最常见和最重要的用途之一是**密码保护 (password protection)**。考虑用户在登录笔记本电脑之前输入密码。为了认证用户，用户密码的某种形式必须存储在笔记

本电脑的某个位置。如果用户密码以明文形式存储，那么窃取笔记本电脑的攻击者可以从硬盘驱动器中读取用户密码，然后以该用户的身份登录。（对于已经可以读取硬盘驱动器内容的攻击者，尝试隐藏密码可能看起来毫无意义。然而，硬盘驱动器上的文件可能已使用用户密码派生的密钥进行加密，因此只有在输入密码后才能访问。此外，用户很可能会在其他站点使用相同的密码。）

可以通过存储密码的**哈希值**而不是密码本身来减轻这种风险。也就是说，硬盘驱动器中存储值 $h_{\mathrm{pw}} = H(\mathrm{pw})$ 的密码文件；稍后，当用户输入其密码 pw 时，操作系统会检查 $H(\mathrm{pw}) = h_{\mathrm{pw}}$ 是否成立，然后授予访问权限。对于基于密码的身份验证，在网络服务器上也会使用相同的基本方法。现在，如果攻击者窃取硬盘驱动器（或侵入网络服务器），他获得的只是密码的哈希值而不是密码本身。

如果密码是从某个相对较小的可能性空间 $\mathcal{D}$ 中选择的（例如，$\mathcal{D}$ 可能是一个英文字典，在这种情况下 $|\mathcal{D}| \approx 80,000$），攻击者可以枚举所有可能的密码 $\mathrm{pw}_1, \mathrm{pw}_2, \ldots \in \mathcal{D}$，并对于每个候选 pw_i，检查 $H(\mathrm{pw}_i) = h_{\mathrm{pw}}$ 是否成立。我们希望断言攻击者不能做得比这更好。（这也将确保攻击者无法了解选择强密码的任何用户的密码，该强密码来自一个大空间。）不幸的是，H 的**原像抗性**（即单向性）不足以蕴涵我们想要的东西。首先，原像抗性仅说明当 x 是从 $\{0,1\}^n$ 这样的大域中均匀选择时， $H(x)$ 难以反转。它没有说明当 x 是从其他空间中选择时，或当 x 根据其他分布选择时，反转 H 的难度。此外，原像抗性没有说明找到原像所需的**具体**时间量。例如，一个哈希函数 H 如果计算 $x \in \{0,1\}^n$ 在给定 $H(x)$ 的情况下需要 $2^{n/2}$ 时间，它仍然可以被认为是原像抗性的，但这可能意味着一个 30 比特的密码只需要 2^{15} 时间就可以恢复。

如果我们将 H 建模为随机预言机，那么我们可以正式证明我们想要的安全性，即恢复 pw 从 h_{pw}（假设 pw 是从 $\mathcal{D}$ 中均匀选择的）平均需要 $|\mathcal{D}|/2$ 次 H 评估。

上述讨论假设攻击者没有进行预处理。然而，正如我们在 5.4.3 节中看到的，预处理可用于生成大表，从而比穷举搜索更快地实现反转（甚至对于随机函数也是如此！）。这在实践中是一个重大的问题：即使一个用户选择的密码是 8 个字母数字字符的随机组合——给出一个大小为 $N = 62^8 \approx 2^{47.6}$ 的密码空间——存在一个使用时间和空间 $N^{2/3} \approx 2^{32}$ 的攻击，这将非常有效。表格只需要生成一次，然后可用于破解数十万个密码，以防服务器泄露。这种攻击在实践中是例行公事。

缓解措施 (Mitigation)。 我们简要描述两种用于减轻密码破解威胁的机制；进一步讨论可以在计算机安全文本中找到。一种技术是使用“慢”哈希函数，或通过使用多次迭代（即计算 $H^{(I)}(\mathrm{pw})$ 对于 $I \gg 1$）来减慢现有哈希函数。这会使合法用户变慢一个因子 I，如果 I 设置为某个“适中”值（例如 $1,000$），这不是问题。另一方面，它对试图一次破解数千个密码的攻

击者产生重大影响。

第二种机制是引入**盐值 (salt)**。当用户注册其密码时，笔记本电脑/服务器将为该用户生成一个长的随机值 s（一个“盐值”），并存储 $(s, h_{\mathrm{pw}} = H(s \| \mathrm{pw}))$，而不是像以前一样仅仅存储 $H(\mathrm{pw})$。由于攻击者事先不知道 s，预处理是无效的，攻击者能做的最好的事情是等到它获得密码文件，然后对域 $\mathcal{D}$ 进行线性时间的穷举搜索，如前所述。另请注意，由于为每个存储的密码使用了不同的盐值，因此需要单独的暴力搜索来恢复每个密码。

5.6.4 密钥派生 (Key Derivation)

我们所见过的所有对称密钥密码系统都要求秘密密钥是一个**均匀分布**的比特字符串。然而，通常对于双方来说，依赖于**非均匀分布**的共享信息（例如密码或生物识别数据）更方便。（在第 10 章中，我们将看到双方如何通过交互来生成一个**非均匀分布**但**高熵**的共享秘密。）双方可以尝试直接使用他们的共享信息作为秘密密钥，但通常这不会是安全的（例如，私钥方案都假设密钥是均匀分布的）。此外，共享数据甚至可能没有用作秘密密钥的正确格式（例如，它可能太长）。

截断共享秘密，或以某种其他**特设 (ad hoc)** 的方式将其映射到正确长度的字符串，可能会丢失大量的**熵 (entropy)**。（我们在下面更正式地定义一个熵的概念，但现在可以将熵视为可能共享秘密空间的对数。）例如，假设两方共享一个由 28 个随机大写字母组成的密码，并希望使用一个 128 比特密钥的密码系统。由于每个字符有 26 种可能性，因此有 $26^{28} > 2^{130}$ 个可能的密码。如果密码以 ASCII 格式共享，每个字符使用 8 比特存储，则密码的总长度为 224 比特。如果双方将密码截断为前 128 比特，他们将只使用密码的前 16 个字符。然而，这不会是一个均匀分布的 128 比特字符串！事实上，字母 $\mathrm{A} - \mathrm{Z}$ 的 ASCII 表示介于 $0\mathrm{x}41$ 和 $0\mathrm{x}5\mathrm{A}$ 之间；特别是，每个字节的前 3 个比特始终是 010。这意味着所得密钥的 37.5% 的比特是固定的，并且双方派生的 128 比特密钥只有大约 75 比特的熵（即，只有大约 2^{75} 种可能的密钥）。

我们需要一个**通用解决方案 (generic solution)**，用于从**高熵**（但不一定是均匀的）共享秘密中派生密钥。在继续之前，我们定义在此处考虑的熵的概念。

定义 5.12 概率分布 X 具有 m 比特的**最小熵 (min-entropy)**，如果对于每个固定值 x，$\Pr_{x \leftarrow X}[X = x] \leq 2^{-m}$ 成立。也就是说，即使最可能的结果也以最多 2^{-m} 的概率发生。

大小为 $\mathcal{S}$ 的集合上的均匀分布具有最小熵 $\log |\mathcal{S}|$。一个分布中一个元素以 $1/10$ 的概率发生，而 90 个元素每个都以 $1/100$ 的概率发生的最小熵约为 $\log 10 \approx 3.3$。分布的最小熵衡量了攻击者可以猜中从该分布中采样的值的概率；攻击者最好的策略是猜测最可能的值，因此如果分布具有最小熵 m，则攻击者正确猜测的概率最多为 2^{-m}。这解释了为什么最小熵（而不是其他熵概念）在我们的上下文中很有用。最小熵的一个扩展，称为**计算最小熵 (computational**

min-entropy)，其定义与上述相同，只是要求该分布在计算上**不可区分**于具有给定最小熵的分布。（计算不可区分性的概念在 7.8 节中正式定义。）

密钥派生函数 (key-derivation function) 提供了一种从任何具有高（计算）最小熵的分布中获得均匀分布字符串的方法。不难看出，如果我们将哈希函数 H 建模为随机预言机，那么 H 就充当了一个很好的密钥派生函数。考虑攻击者对 $H(X)$ 的不确定性，其中 X 是从具有最小熵 m 的分布中采样的（作为一个技术点，我们要求该分布独立于 H）。攻击者对 H 的每次查询都可以被视为对 X 值的“猜测”；根据分布的最小熵假设，攻击者对 H 进行 q 次查询，猜中 X 的概率最多为 $q \cdot 2^{-m}$。如果攻击者没有查询 X 给 H，那么 $H(X)$ 就是一个均匀字符串。

也可以在不依赖随机预言机模型的情况下，使用称为**（强）提取器 (strong) extractors** 的带密钥哈希函数来设计密钥派生函数。提取器的密钥必须是均匀的，但不需要保密。这方面的标准之一称为 HKDF；请参阅本章末尾的参考文献。

5.6.5 承诺方案 (Commitment Schemes)

承诺方案 (commitment scheme) 允许一方通过发送**承诺值 (commitment value)** com 来对消息 m 进行“承诺 (commit)”，同时获得以下看似矛盾的属性：

- **隐藏性 (Hiding)：** 承诺没有泄露关于 m 的任何信息。
- **绑定性 (Binding)：** 对于承诺者来说，输出一个承诺 com，使其随后可以“打开 (open)”为两个不同的消息 m, m'，是不可行的。（从这个意义上说，com 真正“承诺”了一个明确定义的值。）

承诺方案可以被视为一个**数字信封 (digital envelope)**：将消息密封在一个信封中并将其交给另一方提供了隐私（直到信封被打开）和绑定性（因为信封被密封）。

形式上，**（非交互式）承诺方案**由一个概率算法 Gen 定义，该算法输出**公共参数 (public parameters)** params 和一个算法 Com，该算法接受 params 和一个消息 $m \in \{0,1\}^n$ 作为输入并输出一个承诺 com；我们将 Com 使用的随机性明确表示，并将其记为 r。发送方通过选择均匀的 r，计算 $\text{com} := \text{Com}(\text{params}, m; r)$，并将其发送给接收方来承诺 m。发送方可以稍后通过发送 m, r 给接收方来**解除承诺 (decommit)** com；接收方通过检查 $\text{Com}(\text{params}, m; r) = \text{com}$ 来验证。

非形式化地，隐藏性意味着 com 没有泄露关于 m 的任何信息；绑定性意味着不可能输出一个可以以两种不同方式打开的承诺 com。我们现在正式定义这些属性。

承诺隐藏实验 $\mathrm{Hiding}_{\mathcal{A},\mathrm{Com}}(n)$：

1. 生成参数 $\mathrm{params} \leftarrow \mathrm{Gen}(1^n)$。
2. 攻击者 $\mathcal{A}$ 获得输入 params，并输出一对消息 $m_0, m_1 \in \{0,1\}^n$。
3. 选择均匀的比特 $b \in \{0,1\}$，并计算 $\mathrm{com} \leftarrow \mathrm{Com}(\mathrm{params}, m_b; r)$。
4. 攻击者 $\mathcal{A}$ 获得 com，并输出一个比特 b'。
5. 当且仅当 $b' = b$ 时，实验的输出定义为 1。

承诺绑定实验 $\mathrm{Binding}_{\mathcal{A},\mathrm{Com}}(n)$：

1. 生成参数 $\mathrm{params} \leftarrow \mathrm{Gen}(1^n)$。
2. 攻击者 $\mathcal{A}$ 获得输入 params，并输出 $(\mathrm{com}, m, r, m', r')$。
3. 当且仅当 $m \neq m'$ 且 $\mathrm{Com}(\mathrm{params}, m; r) = \mathrm{com} = \mathrm{Com}(\mathrm{params}, m'; r')$ 时，实验的输出定义为 1。

定义 5.13 承诺方案 Com 是**安全**的，如果对于所有 ppt 攻击者 $\mathcal{A}$，存在一个可忽略函数 negl，使得

$$\Pr[\mathrm{Hiding}_{\mathcal{A},\mathrm{Com}}(n) = 1] \leq 1/2 + \mathrm{negl}(n)$$

且

$$\Pr[\mathrm{Binding}_{\mathcal{A},\mathrm{Com}}(n) = 1] \leq \mathrm{negl}(n).$$

很容易从随机预言机 H 构造一个安全的承诺方案。要承诺消息 m，发送方选择均匀的 $r \in \{0,1\}^n$ 并输出 $\mathrm{com} := H(m \| r)$。（在随机预言机模型中，不需要 Gen 和 params，因为 H 实际上充当了该方案的公共参数。）直观上，隐藏性源于攻击者查询 $H(x \| r)$ 的概率可忽略（因为 r 是均匀 n 比特字符串）；如果它从未进行这种形式的查询，那么 $H(m \| r)$ 就不会泄露关于 m 的任何信息。绑定性源于 H 是抗碰撞的这一事实。

可以在**没有**随机预言机的情况下构造承诺方案（事实上，可以从单向函数构造），但细节超出了本书的范围。

---

[1] 可以证明，这（本质上）是最坏的情况，当 H 偏离随机且 $\{x_i\}$ 是均匀选择时，碰撞发生的概率更高。