第 3 章　DES 算法

3.1　DES 算法描述

DES（Data Encryption Standard，数据加密标准）是美国 IBM 公司研制的对称密码算法。1977 年，美国国家标准与技术研究院（NIST）将 DES 确定为联邦资料处理标准（FIPS），并授权在非密级政府通信中使用。

DES 诞生之后便得到了广泛的关注和研究，因其巧妙的设计原理，起初具有较高的安全性，但是随着计算技术的发展和密码分析技术水平的提高，DES 密钥空间较小的缺点逐渐被暴露出来。1997 年，56 比特的 DES 被攻破，DES 也因此被证明不再安全。借鉴 DES 的设计思想，3-DES、AES 等新算法被先后提出，目前这些新算法仍然被广泛应用。

3.1.1　算法结构

DES 是一种分组密码，其设计遵循分组密码的两个重要原则：混淆和扩散。混淆是使密文与密钥之间的统计关系尽可能复杂化；扩散则是让每个明文位尽可能多地作用到密文位中。

DES 的明文分组、密文分组及密钥分组大小均为 64 比特。在实际运算中，对密钥每隔 7 个比特设置一个奇偶校验位，即密钥的第 8 位、第 16 位、第 24 位、第 32 位、第 40 位、第 48 位、第 56 位和第 64 位为奇偶校验位，不参与运算。因此，实际参与运算的密钥长度只有 56 比特。

DES 是采用 Feistel 结构的分组密码算法，其特征是加密和解密算法高度相似，且由多轮相同的轮函数组成。DES 密码算法主要包括密钥编排算法及加/解密算法，其中，密钥编排算法负责根据 64 比特初始密钥生成每轮加/解密所需的 48 比特轮密钥；加/解密过程分为初始置换、16 次轮加密及逆初始置换。DES 加/解密算法流程如图 3-1 所示。

（1）对 64 比特明文块进行初始置换（Initial Permutation，IP），产生 32 比特的左明文 L_0 和右明文 R_0。

（2）轮（Round）加密：执行轮函数 f，输入 48 比特轮密钥 K_{i+1}，以及上一轮生成的 L_i、R_i，输出 L_{i+1} 与 R_{i+1}。

（3）重复（2）共 16 次，得到 L_{16} 与 R_{16}。

（4）将 L_{16} 与 R_{16} 拼接起来，对组成的 64 比特块进行逆初始置换（IP^{-1}），得到 64 比特密文。

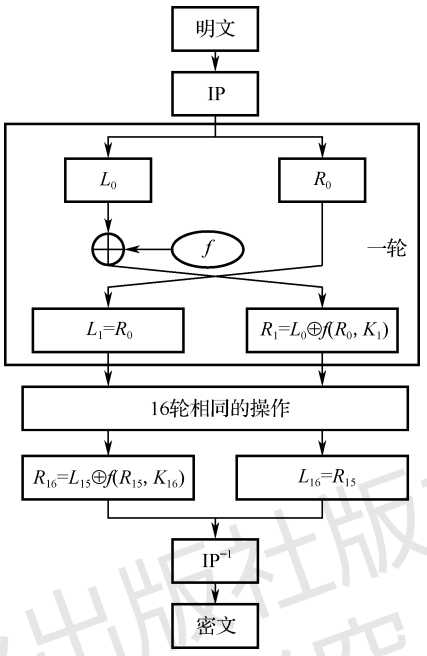


图 3-1　DES 加/解密算法流程

由于 DES 的巧妙设计，其加密与解密过程唯一的区别是颠倒了轮密钥的使用顺序。假设初始密钥为 K，在加密时由密钥编排算法扩展为轮密钥 $K_1,K_2,K_3,\cdots,K_{16}$，则解密时轮密钥顺序应为 $K_{16},K_{15},K_{14},\cdots,K_1$。

3.1.2　核心部件

1．初始置换和逆初始置换

初始置换（IP）将输入数据块按位重新排序，并把结果拆分成两部分。其输入为 64 比特明文数据块，输出为 32 比特的左半部分 L_0 和 32 比特的右半部分 R_0。

逆初始置换（IP^{-1}）是 IP 的逆过程，也将输入数据块按位重新排序。其输入为 L_{16} 和 R_{16} 合并成的 64 比特数据块，输出是 64 比特密文数据块。

上述两个置换都是位置换，可以看作简单地调换各个位的顺序。其置换规则由表 IP 以及 IP^{-1} 给出，表中元素 $IP_{i,j}=k$ 的含义是：置换后第 i 行与第 j 列的交点位置存放原来第 k 个比特。例如，若 $IP_{1,2}=50$，则意味着输出块中第 1 行与第 2 列的交点位应该存放输入块的第 50 个比特。代码示例如下。

```
//DES 初始置换函数 IP
//输入：64 比特明文块 message_piece
```

```
//输出：32 比特左、右部分 l、r
int shift_size;                  //存放替代位的地址
unsigned char shift_byte;   //存放移位
unsigned char initial_permutation[8];                    //存放初始置换结果
memset(initial_permutation, 0, 8);

for (i=0; i<64; i++) {
      shift_size = initial_message_permutation[i];       //取 IP 表
      shift_byte = 0x80 >> ((shift_size - 1)%8);         //计算替代比特在字节内的位置
      shift_byte&= message_piece[(shift_size - 1)/8];    //取出对应比特
      shift_byte<<= ((shift_size - 1)%8);
      initial_permutation[i/8] |= (shift_byte>> i%8);    //存放结果
}

unsigned char l[4], r[4];                //左明文 l，右明文 r
for (i=0; i<4; i++) {
      l[i] = initial_permutation[i];
      r[i] = initial_permutation[i+4];    //l、r 各取结果 32 比特
}
```

2．f 函数

在第 i 轮中，f 函数的输入为第 $i-1$ 轮输出的 R_{i-1}，以及当前轮的轮密钥 K_i；输出为 32 比特处理结果 $f(R_{i-1},K_i)$。之后，将 $f(R_{i-1},K_i)$ 与 L_{i-1} 进行异或，生成第 $i+1$ 轮的输入 R_i。f 函数内部又可以按序分成 3 个部分：E 扩展置换、S 盒替换以及 P 盒置换，如图 3-2 所示。

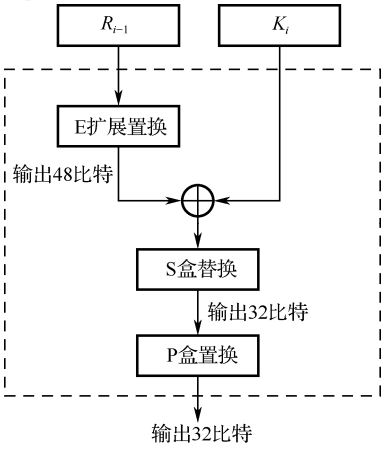


图 3-2　f 函数内部构造

1）E 扩展置换

E 扩展置换的输入为 32 比特的 R_{i-1}，输出为 48 比特数据块。其扩展数据块的目的有两个：一是使得数据块能够与轮密钥进行异或运算；二是在后续的 S 盒替换运算中进行压缩。

与 IP 类似，E 扩展置换的规则以表 E（见 3.4.1 节）的形式给出。表中元素 $E_{i,j}=k$ 的含义是：输出块中第 i 行与第 j 列的交点位置存放输入块的第 k 个比特。例如，若 $E_{1,1}=32$，则意味着输出块中第 1 行第 1 列存放输入块的第 32 个比特。可以注意到，表中的某些项是重复的。这是为了扩展数据长度，某些输入位会被复制到两个不同的输出位置，也因此增强了数据的扩散性。代码示例如下。

```
//DES E 扩展函数
//输入：32 比特右半部分密文 r
```

```
//输出：E 扩展结果 er
for (i=0; i<48; i++) {
    shift_size = message_expansion[i];      //取表 E
    shift_byte = 0x80 >> ((shift_size - 1)%8);
    shift_byte&= r[(shift_size - 1)/8];
    shift_byte<<= ((shift_size - 1)%8);     //计算替换元素位置
    er[i/8] |= (shift_byte>> i%8);          //存放结果
}
```

2）S 盒替换

S 盒替换的目的是对扩展后的数据块进行压缩，同时进行一次非线性替换，替换操作由 8 个不同的 S 盒查找表完成。S 盒替换的构造如图 3-3 所示。首先将 E 扩展置换结果与轮密钥 K_i 进行异或后的 48 比特数据块分为 8 组，每组 6 比特，分别送入对应的 S 盒。每个 S 盒有一张 4 行 16 列的查找表，由 6 比特输入作为索引，输出 4 比特结果。记 S 盒的 6 比特输入为 *abcdef*，第 0 比特和第 5 比特组合形成行号 *af*，第 1～4 比特组合形成列号 *bcde*，用于查找 4 比特结果。实际上，替换操作由 8 个不同的 S 盒查找表完成。最后再将 8 个 S 盒的 4 比特结果合并，形成 32 比特输出。需要注意的是，S 盒的行列号都是从 0 开始的。S 盒替换达成了数据混淆的目的，是整个 DES 算法中唯一的非线性部件，也是整个算法安全性的来源。

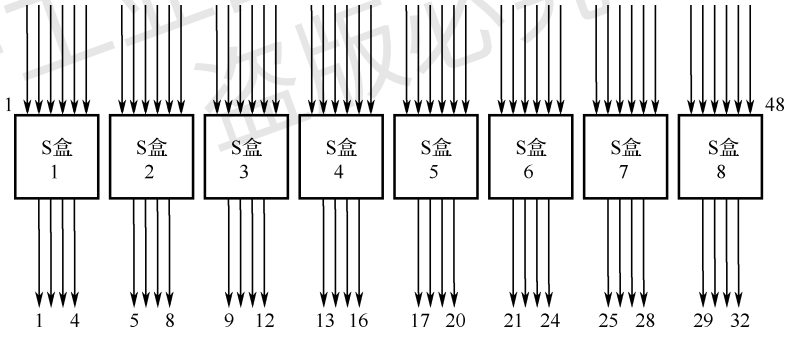


图 3-3　S 盒替换的构造

【例】以 S_1 为例，若输入为 001101，则行号为 01，即十进制的 1；列号为 0110，即十进制的 6。查找表 S_1 的第 1 行第 6 列，结果为 13，即二进制的 1011。因此，输入 001101 经过 S 盒替换后，变换为输出 1011。代码示例如下。

```
//DES   S 盒替换（第 1 字节）
//输入：32 比特 E 扩展置换结果 er
//输出：S 盒替换结果 ser（第 1 字节）
row = 0;                          //行号
row |= ((er[0] & 0x80) >> 6);     //取 S1 盒输入的最高位
row |= ((er[0] & 0x04) >> 2);     //取 S1 盒输入的最低位，组成行号
column = 0;                       //列号
```

```
column |= ((er[0] & 0x78) >> 3);                          //取列号
ser[0] |= ((unsigned char)S1[row×16+column] << 4);  //查找表 S1
row = 0;
row |= (er[0] & 0x02);                                    //取 S2 盒输入的最高位
row |= ((er[1] & 0x10) >> 4);                             //取 S2 盒输入的最低位，组成行号
column = 0;
column |= ((er[0] & 0x01) << 3);
column |= ((er[1] & 0xE0) >> 5);                          //取 S2 盒列号（分别在 2 字节）
ser[0] |= (unsigned char)S2[row×16+column];          //查找表 S2
```

3）P 盒置换

S 盒替换得到了一个 32 比特输出，接着还需对其进行一次 P 盒置换。P 盒置换与 IP 置换类似，都是按位置换。该置换为一一映射，即把输入的每一位映射到输出位，且任何一位都不能被映射两次，也不能被省略。置换规则以表 P 的形式给出，含义与 IP 置换、E 扩展置换类似，在此不做赘述。

3．密钥编排

DES 加密算法共对明文执行 16 次轮加密，需要 16 个轮密钥。由初始 64 比特用户密钥生成 16 个 48 比特轮密钥的过程称为密钥编排。密钥编排算法也需进行多轮迭代运算，其算法流程如下。

（1）假设 64 比特初始用户密钥为 K，对其进行选择置换 1（PC_1）得到 56 比特密钥 K'，其左、右两部分分别记为 C_0、D_0。

（2）第 i 轮时，分别将 C_{i-1}、D_{i-1} 循环左移，得到 C_i、D_i。左移的位数与轮次有关，在第 1、2、9、16 轮移动一位；否则移动两位。

（3）将 C_i、D_i 合并为 56 比特，进行选择置换 2（PC_2），生成 48 比特轮密钥 K_i。

（4）重复步骤（2）、（3）共 16 次，获得 16 个轮密钥。

由于密钥编排要进行置换和密钥长度的压缩，因此称其为压缩置换（Compression Permutation）。压缩置换的存在，使得每轮都使用不同的密钥位，这增加了 DES 的安全性。

3.2　DES 快速实现方法

随着计算机算力的提升以及密码分析技术的发展，目前 DES 算法已被证明不安全。从 2000 年开始，由于 3-DES 和 AES 等分组密码的提出，基本上已不再有关于 DES 算法实现的研究。即便如此，从 DES 诞生起也出现了大量关于 DES 实现技术的研究，下面挑选其中的一部分进行介绍。

3.2.1 基于 AVX 的 DES 快速实现

基于第 2 章介绍过的 SIMD 指令（详见 2.3 节），可利用单指令多数据流来加速某些算法，减少实现开销。Gueron 等人将这种思想应用到了 DES 中，并将该方法称为“多块（Multi-Block）DES”。该方法将输入调整成适合使用 SIMD 指令的形式，可以高效地并行处理多条消息，但需要花费额外的变换操作。以下做简单的介绍。

1．变换操作

先考虑单条输入：假设单条输入消息为 M，有一指针 p 保存其地址。在加密 M 时，可以直接读取内存中连续的 64 比特块，并分别对其进行加密。

再考虑两条输入：假设两条消息为 M_1 和 M_2，其地址分别由指针 p_1 和 p_2 保存，B_1、B_2 分别是 M_1、M_2 的两个 64 比特块。在 CBC（Cipher Block Chaining）模式下，若希望在 128 比特寄存器（xmm）上使用 SIMD 指令对其并行处理，则需要将 B_1、B_2 放入同一个 xmm 寄存器，假设为 xmm2。在一般情况下，只能通过两个指针分别从内存中读取 B_1 和 B_2，并分别存入两个寄存器 xmm0 和 xmm1 的低半部分。然后将 xmm0、xmm1 的低 64 比特内容合并到 xmm2 中。代码样例如下。

```
vmovdqu (%rsi), %xmm0
vmovdqu (%rdi), %xmm1
vpshufd $0x4e, %xmm1, %xmm1
vpblendd $0x0c, %xmm0, %xmm1, %xmm2
```

更高效的方法是通过指针将两个块读入 xmm1、xmm2，然后用类似的软件流将这些寄存器的内容合并到两个 xmm 寄存器中。代码样例如下。

```
vmovdqu (%rsi), %xmm0
vmovdqu (%rdi), %xmm1
vpunpcklqdq %xmm0, %xmm1, %xmm2
vpunpckhqdq %xmm0, %xmm1, %xmm3
```

传统的 DES 算法实际上是独立处理这两个比特块的。记 $B_1=[b_1,a_1]$，$B_2=[b_2,a_2]$，其中 a_1、a_2、b_1、b_2 都是 32 比特的半块。Gueron 等人提出的高效实现方法将这些半块放入两个寄存器 xmm0、xmm1 中，记 xmm0=$[0,0,b_2,b_1]$ 及 xmm1=$[0,0,a_2,a_1]$。代码样例如下。

```
vmovdqu (%rsi), %xmm0
vmovdqu (%rdi), %xmm1
vpunpckldq %xmm0, %xmm1, %xmm2
vpunpckhdq %xmm0, %xmm1, %xmm3
```

2．SIMD 处理

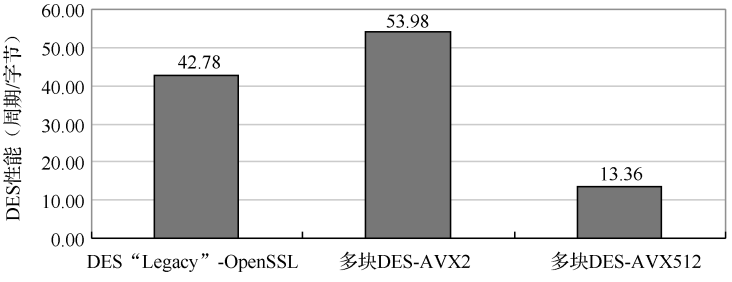
当输入组织成几个寄存器中半块的形式时，便可以使用 SIMD 指令对其进行操作。用 SIMD 执行 32 比特半块数据的移位、逻辑与及异或指令，可以轻松实现初始置换和逆初始置换。f 函数由 E 扩展置换、S 盒替换和 P 盒置换组成。其中，E 扩展置换可以用类似的 SIMD 指令完成，而 S 盒替换和 P 盒置换都需要根据对应的表格设置常量。S 盒将每块中的每 6 比特输入转换成对应的 4 比特输出，为了防止串行访问 S 盒表，要将表的一部分加载到寄存器中，并且将不同块的几个元素并行地排列。最后，将操作结果寄存器使用掩码寄存器混合，以确定 S 盒替换是否使用 S 盒表的正确部分。以此类推，迭代 S 盒表的每一部分就可以得到最终结果，并将其在结果寄存器中进行合并。

这项技术可以在 Intel 的 AVX512 指令集上完成。AVX512 能够在一个寄存器中一次性处理 8 个半块，在一字节中存储 S 盒的 6 比特输入，最终将 8 个不同块的 48 比特输入排列在一起。在 AVX512 寄存器中，这个排列过程可以加载绝大部分的 S 盒表。在 16 次这样的排列之后，整个 S 盒表就加载完成了。这个过程可以使用多个寄存器并行实现，每次加载 S 盒时进行相同的排列。完成 S 盒操作后，用高级 SIMD 指令准备 P 盒的常数，对所有半块并行执行 P 盒置换。

该技术经过调整后也可向下兼容 Haswell 的 AVX2 指令集，但使用 AVX2 将导致并行性降低，在处理 S 盒时的迭代次数增加，成本更大。这些额外成本导致在 CBC 模式下使用多块模式反而得不偿失。

对于单个输入的数据流，可以在 ECB（Electronic Code Book）模式下实现类似的技术。ECB 模式不需要变换操作，但需要额外的指令将块拆分为两半，并在处理结束后将其合并。但由于这个额外操作与 CBC 模式下的变换操作相比要简单得多，因此 ECB 模式下的多块实现将带来更好的性能提升。

与 OpenSSL 上的实现相比，该技术在 AVX2 上的实现性能为 OpenSSL 上的 79%；但借助 AVX512 指令及更大的寄存器，采用该技术可实现 3.2 倍性能提升。基于 SIMD 的 DES 快速实现如图 3-4 所示。



	OpenSSL DES	AVX2 多块DES	AVX512 多块DES
加速	1×	0.79×	3.2×

图 3-4　基于 SIMD 的 DES 快速实现

该技术的思想同样适用于其他密码算法，但对于使用更复杂运算（如乘法或大 S 盒）的密码算法，则需要更多的指令来实现复杂运算，因此效率较低。

3.2.2 64 位平台上的 DES 快速实现

并非所有处理器都支持 SIMD 指令，此时需要其他的方法来提升算法的效率。Biham 等介绍了一种新的方法，将 64 位处理器当作 64 个并行的 1 位处理器，使其在软件实现中达到类似 SIMD 的效果。下面介绍了两种方案，分别是非标准表示和标准表示的 DES 快速实现。

DES 快速实现的总体思路是在并行加密 64 个字时，一次加密 64 个字的各 1 位，而非一次加密一个 64 位字。为此，必须将每一比特分开存放在不同的字中，在计算时分别存取。因此，要重新考虑 DES 所需的各个操作。异或操作是按位计算的，不受影响，P 盒置换和 E 扩展置换操作实现时不用改变字的存放顺序，只要改变寄存器的命名顺序，就可以对需要的比特直接寻址，因此不需要额外指令。S 盒比较复杂，为了实现 S 盒表的查找，需要从不同的字中分别提取 6 比特，拼凑成索引；查找到结果后再将 4 比特分别存放，非常低效。因此需要针对 S 盒查找进行优化。若使用与、或、非、异或等操作表示 S 盒的门电路，则采用一些优化手段可以减少 S 盒的总门数。

记一个 S 盒的 6 比特输入为 $abcdef$，先计算 de 的 16 种函数（除了结果恒为 0、1 的常函数），再存入 14 个寄存器中，这里需要两个求非运算以及 10 个额外操作（0、1、d、f、$\overline{d}$ 和 $\overline{f}$ 是已知的）。在每个 S 盒中，该步骤只需要预先做一次，之后计算输出位时就可以直接使用这些函数，所以 S 盒的每行只需要 6 个操作来计算，再用 6 个操作来组合计算结果。因此，计算 S 盒的每个输出位需要 30 个操作；每个 S 盒总共需要 $12+4\times30=132$ 个操作。由于 S 盒表的某些部分是相似的（相同或互补），操作可以进一步优化。最终，平均每个 S 盒只需要 100 个操作。4 个 2 比特输入（bc 或 af）的值可以形成组合，如不同行的相同 2 比特的组合 $((bc=00),(bc=01),(bc=10),(bc=11))$，或者是 4 行的值的组合。这 4 个值的组合可以表示为（第 1 种情况）

$$\left[\underline{f_{00}}\oplus c\cdot\underline{(f_{00}\oplus f_{01})}\right]\oplus b\left[\underline{(f_{00}\oplus f_{10})}\oplus c\cdot\underline{(f_{00}\oplus f_{01}\oplus f_{10}\oplus f_{11})}\right]$$

其中，带下画线的函数值为经过预计算的已知常数。f_{bc} 是保存在上述寄存器中的 16 个值之一，$f_{bc}=S(abcdef)$（其中 de 是输入的实际值，af 是假定值）。中间步骤计算的是项组合的值（如 f_{00}、$f_{00}\oplus f_{01}$、$f_{00}\oplus f_{10}$、$f_{00}\oplus f_{01}\oplus f_{10}\oplus f_{11}$），而非项本身。表 3-1 和表 3-2 描述了在 300MHz 的 Alpha 处理器上，每轮以及整个 DES 的最大门数量。因此，预期速度应该达到 $300\times\frac{2^{20}}{4}=75\,\text{Mbps}$，该实现方法比 64 位 Alpha 计算机上最快的 DES 实现快了约 5 倍。

表 3-1　非标准表示法 DES 在 Alpha 上每轮指令的数量

操　　作	指　　令
E 扩展置换	0
密钥编排	48
P 盒置换	0
与左半部分异或	32
S 盒替换	$8\times100=800$（平均值）
加载+写回	$8\times(6+6\text{load}+4\text{load}+4\text{store})=160$
每轮总数	1040

表 3-2　非标准表示法 DES 在 Alpha 上指令的总数

操　　作	总　　数	平均每块的值
IP、IP^{-1}	0	0
16 轮	$16\times1040=16640$	260
表示转换	2500	40

在标准和非标准表示之间的转化也可以在约 1250 条指令中完成。在加密之前和之后两次执行此操作需要大约 2500 条指令，每个加密块大约需要 40 条指令。

这种实现可以用于两种不同的情景：①标准表示的 DES 加/解密过程，可以通过使用一些指令转化表示，以与其他 DES 实现有效兼容；②磁盘集群、大型通信数据包等大块数据的加/解密。此时，标准表示法并不重要，不需要进行标准/非标准表示的转换，因此这种实现更快。

这种实现方法实际上可以应用于任何密码，但实现效率取决于许多因素，如原始密码的效率、处理器的字长以及密码操作的复杂性。对于操作简单（如不涉及乘法）、S 盒规模小（它们的门复杂度会更小）或使用更小的寄存器的算法，该实现方法则更实用。

接下来我们讨论 64 位处理器上的标准表示法 DES 实现。与 32 位处理器不同，在 64 位处理器上，扩展到 48 比特的密文右半部分 R_i 可以完整地存储在一个字中。此外，如果将 S 盒输入的 6 个分别存放的比特存放在一整个字节中，我们就可以通过单字节引用直接访问 S 盒表。同理，我们可以将查找表应用到初始置换和逆初始置换中（这里的查找表从单字节映射到 64 比特结果），并对各种表查找的结果进行异或处理。

在每轮中，我们将 R_i（表示为 8 字节，但每字节中只使用 6 比特）和轮密钥（表示方法相同）进行异或，然后用 8 个查找表来实现 S 盒替换，结果也要异或。此时，S 盒的 64 比特结果中已经包括了 P 盒置换和 E 扩展置换操作。但是，在使用这种表示法时，逆初始置换中应该省略左、右半块中的重复位。

由于同一轮中 8 个 S 盒可以并行处理，因此流水线不会阻塞；但如 Feal、Khufu 等其他密码中，每个操作的输入都取决于前一个操作的输出，所以可能导致阻塞。尤其在

处理器算力提升后（如可同时计算的指令数量增加），那些无法并行处理的算法的阻塞问题会更加严重（因为处理器的算力无法得到发挥），而这也是本算法的重大优势。此外，下一轮的输入是由上一轮的结果决定的。这虽然不会减慢处理速度，但还有优化的空间。因为每轮的输入只与上一轮的 6 个 S 盒有关，所以可以继续优化代码，提高算法的并行效率。同时，本算法的所有表和变量合计约 4 字节，可以轻松载入到缓存中，因此算法是内存友好的。

表 3-3 和表 3-4 描述了此实现所需的操作数量以及 Alpha 处理器上的指令数量。Biham 等人的文献提供了 300MHz Alpha 处理器上的 C 实现，加密速度达到了 46Mbps。在同一个处理器上，Eric Young 的 libdes 算法（单个 DES）的速度运行是 28Mbps。可见，该实现比当时已知的最快实现快了一倍。

表 3-3　快速标准 DES 在 Alpha 上每轮指令的操作数量

操　　作	操 作 类 型	指　令　数
密钥异或	1 load + XOR	2
EPS	8 次查表	8×3 = 24(extbl, add, lookup)
S 盒与 L 异或	8 XOR	8

表 3-4　快速标准 DES 在 Alpha 上指令的总数

操　　作	操 作 类 型	指　令　数
初始置换	5 times(3 XORs, 2 shifts, 1 AND)	5×6 = 30
E 扩展置换	初始扩展	26
16 轮	每一轮用 34 条指令	16×34 = 544
逆扩展置换	扩展置换的逆操作	4
IP^{-1}	最终置换	30

3.3　3-DES 在 GPU 上的高速实现

由于 DES 密钥长度为 64 比特，因此已经被更安全的 3-DES 或 AES 所取代。3-DES 加强了 DES 的安全性，可以被看作 AES 和 DES 之间的一个过渡形式。实际上，3-DES 只是连续进行 3 次 DES 操作。理论上，它消耗的 CPU 时间大约是 DES 的 3 倍，但是，它的安全性远远超过 DES 的 3 倍并具有较高的实际安全性。因此，Yeh 等人通过在 GPU 上实现了 3-DES 加/解密方案，有效提高了其性能。

3.3.1　3-DES 结构设计

由于 DES 是 3-DES 操作的主要核心，因此首先分析 DES 的结构。在 DES 中，密钥编排、IP 、IP^{-1}、轮函数等操作都是由 CPU 完成的。因此，这些操作花费了大量的 CPU 时间和计算资源。Yeh 等人将 CPU 的一些操作分配给 GPU，加快整个系统的运算速度，减少 CPU 资源的消耗。

3-DES 的加密可以分为三部分（见图 3-5）：首先用密钥 1 对明文进行 DES 加密；然后用密钥 2 进行 DES 解密；最后用密钥 3 进行 DES 加密，完成整个加密操作。3-DES 解密是对 3-DES 加密的逆向操作。可见，3-DES 需要花费至少 3 倍于 DES 的 CPU 时间。

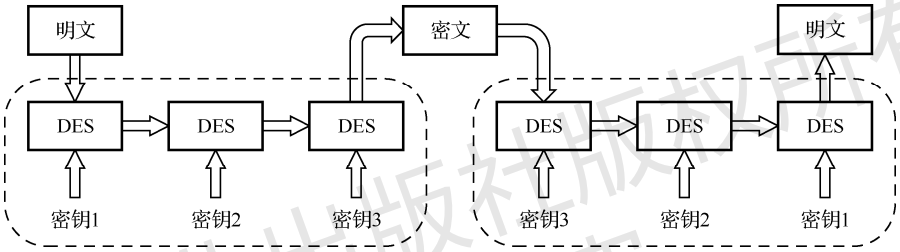


图 3-5　3-DES 结构

在 CUDA 编程中，实现 3-DES 的基本想法是将加密和解密操作放在 GPU 上，同时保留传统的 3-DES 结构。然而，这在 CUDA 编程中可能会面临以下一些问题。

（1）并行算法不容易构建。

由于 DES 程序是顺序执行的，因此使其内部流程并行化很困难。虽然 AES 使用了大量矩阵解决这个问题，但 DES 没有好的矩阵解决方案。

为了解决这个问题，Yeh 等人修改了传统的 DES 算法以适应 CUDA 程序。假设一次大约输入 $64\times N$ 比特的明文，其方法是在 GPU 上使用 N 个线程，创建具有并行性的 DES。

（2）如何利用 GPU 内存优化系统。

当尝试一次输入 $64\times N$ 比特的明文时，系统效率会迅速下降，主要原因是未合理分配 GPU 内存。

传统的 3-DES 加密包括两个 DES 加密和一个 DES 解密。当把 3-DES 加密和解密的 C 代码改写成 CUDA 代码时，减少数据移动的频率非常重要。Yeh 等人的文献中的初始函数调用为

```
__global__DES_encryption()
__global__DES_decryption()
```

其中，“__global__”意味着这个函数调用将由 GPU 核心执行。它们需要大量的数

据移动来完成一次 3-DES 加密。若将 3 个操作合并为一个操作，可以将 6 个数据移动操作减少为两个数据移动操作。因此，可建立针对 CUDA 编程的 3-DES。新的 3-DES 加密流程如图 3-6 所示。图 3-6 仅展示了 3-DES 在 GPU 上的加密流程，解密的流程与加密的流程类似。

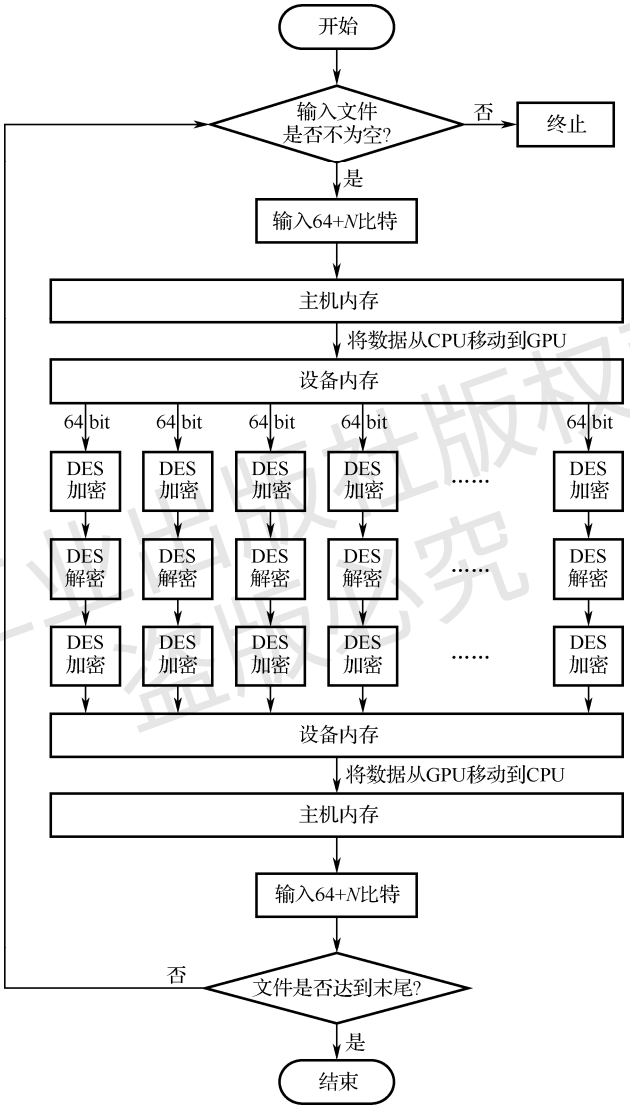


图 3-6　新的 3-DES 加密流程

3.3.2　在 GPU 上实现 3-DES

DES 加密和解密的第 1 步是将明文或密文从 CPU 转移到 GPU 上。首先，CPU 发出

指令，在显存中分配相同大小的数据块。然后，CPU 通过 DMA 将数据从内存发送到显存。显卡收到数据后，CPU 可以调用 GPU 实现并行计算。

当 GPU 接收数据和指令时，需要将数据发送给线程。在 DES 的实现中，一个明文分组是 64 比特。使用长度为 8 的无符号数组来存储 64 比特明文，将所有 $64\times N$ 比特数据存储在名为 block 的无符号数组中。首先，GPU 读取 block 数组中的所有元素，并将它们发送到不同的线程，用 $J=(\text{blockIdx}.x\times\text{blockDim}.x+\text{threadIdx}.x)$ 来表示每个线程。其中，J 表示线程 ID；blockIdx 表示一个 Grid 中 Block 的数量；blockDim 表示一个 Block 中线程的数量；threadIdx 表示 Block 中对应的线程 ID。然后，将明文分为两部分，分别命名为 Left 和 Right。Left 和 Right 都为 32 比特，分别由 4 个 unsigned char 元素组成。

在标准文档的各种表中，比特是从左到右排列的（从 1 开始）。在用序列表示时，使用行优先顺序，如图 3-7 所示。

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

图 3-7　行优先位序列

奔腾处理器使用字节（行）顺序读取内存，得到如图 3-8 所示的位序列。序列分为上下两部分，每部分需要 32 比特的寄存器来存储，因此交换这两部分相当于交换了这两个寄存器的标号。

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

图 3-8　在奔腾处理器上的输入/输出位序列

DES 的初始置换结构非常简单，可以用一系列的位交换来实现，使用的算法称为 Hoey 算法。算法开始时，也隐含了上半部分和下半部分的互换。

DES 还需要 16 次轮函数来完成加密和解密。一般通过 DES 函数的 3 次调用来完成 3-DES 加密和解密，这需要在 CPU、GPU 和内存之间移动数据，可能会浪费大量的 CPU 时间，因此减少内存访问的方法是连续做 64 次轮函数，从而程序只需要进行两次内存访问就可以完成一次 3-DES 加密或解密。

重写轮函数的关键是解决内存访问问题。由于常量内存的速度通常比全局内存快，而且不会占用共享内存的空间，因此可使用 unsigned long 类型的常量内存来存储 S 盒和 P 盒。

3.3.3 3-DES 加密的实现

设计一个 3-DES 加密程序，要考虑各种类型的文件。例如，在 PowerPoint 或 Word 等文件中，使用'\0'作为其结束符号。在这种情况下，只需要确定输入的字是什么。但在音乐或电影文件等流媒体数据中，'\0'不是结束符号，所以需要调整程序来适应所有的文件类型。

此外，一次输入的是 $64 \times N$ 比特明文，而不是 64 比特。这可能会在确定结束符号的位置时出现错误。例如，如果一次输入 1000 个字的明文，而结束符号'\0'是第 1250 个字，这时必须做两次循环才能输入所有的明文。但当两次循环后，却读到了明文的第 2000 个字，而该字是编译器无法识别的空字，所以循环并不会停止。

综合上述两个因素，Yeh 等人的文献中给出了伪代码。

```
Algorithm- DES Encryption (plaintext, ciphertext);
Input:plaintext(64 * N bits plaintext).
Output:ciphertext(64 * N bits ciphertext).
// N means how many threads we use in device.
begin
declare integerplaintext[64 * N], temptext[64 * N],ciphertext[64 * N];
declare integerkey1[64], key2[64], key3[64], key[192];
declare integerplaintextsize =64 * N;
while(not reach the end of file)
{
input_EN(plaintext,temptext); //put plaintext tinto temp buffer
key =key_schedule (key1, key2, key3); // generate subkeys
copy temptext form host memory to device memory
__global__3-DES_encryption (temptext, key);
copy temptext form device memory to host memory
swap (ciphertext,temptext);
output_EN (ciphertext);
}
end
```

在上面的伪代码中，只有名为__global__3-DES_encryption 的函数会在 GPU 中执行。除了__global__3-DES_encryption 函数，只有 key_schedule 函数可以被并行化。这里的 key_schedule 沿用了传统 3-DES 的 key_schedule 设计，没有做任何修改。所以，key_schedule 函数的计算量很少，不值得在 CUDA 中执行，其内存访问时间大于计算时间。

另一个值得一提的是输入函数。由于希望代码支持所有的文件类型，因此需要设计新的结束符号判断方法。Yeh 等人的文献中的方法是为所有文件类型设计一个新的结束

符号。如果一次输入 1000 个字，相当于 8000 比特的明文，详细设计如下。

（1）设计新的结束符号。结束符号是一个独特的词，只有程序能识别，所以不能选择常见的词作为结束符号。文章选择"/nend/ab"作为结束符号。

（2）每隔 1000 个字在明文中添加结束符号。程序每隔 1000 个字检查明文是否结束。Yeh 等人用 fread()来确定是否已经读入了所有的明文。如果 fread()返回值为 8000，就意味着输入了 8000 比特或 1000 个字。同时，这意味着该文件可能没有结束，应该加上结束符号，然后继续寻找明文的结尾，所以一次应该准确地输出 1008 个字（1000 个字+8 个字结束符号）。

（3）找出明文的结尾。当 fread()返回的值小于 8000 时，这意味着已经找到了明文的结尾。由于这里对 1000 个字进行了一次处理，因此需要填充明文来达到 1000 个字。结束符号"/nend/ab"只有 8 个字。在结束符号之后，需要用"\0"来填补空缺，以达到 1008 个字。

（4）停止的条件。在找到明文的结尾后，下一轮 fread()返回值为 0。这意味着明文是空的，所以程序将打破循环，停止 3-DES 加密。

3.3.4　3-DES 解密的实现

3-DES 解密的实现与 3-DES 加密类似，但解密需要更多的判断操作，这意味着需要更多的 CPU 时间来决定何时停止操作。与加密不同，解密的输入是全部密文，需要先对密文进行解密，然后才能对其进行分析。

加密和解密的主要差异是判断条件。在 3-DES 加密中，系统可以使用 feof()来检测文件是否到达了终点。但在解密中，需要一次输入所有密文。如果只是解密和输出，明文就会出现多个结束符号。下面是实现的具体细节。

1．输入 1008 个字并解密

当程序启动时，将输入 1008 个字的密文，并将其发送到 GPU 进行解密。这个阶段将一直重复下去，直到没有密文输入。

2．解密后的分析

解密后，程序会分析明文的最后 8 个字。如果这 8 个字是"/nend/ab"，就意味着输入文件没有到达结尾。程序将输出没有"/nend/ab"的结果，并返回第 1 阶段。反之，如果这 8 个字不是"/nend/ab"，就意味着输入的文件已经到达了结尾，程序应该分析这 1008 个字以找出结束符号的位置。当程序找到结束符号的位置时，它将把没有"/nend/ab\0\0\0..."的结果输出为明文。完成了所有工作之后，程序跳出循环。

3.3.5 实现性能及分析

Yeh 等人在 OpenSSL 和显卡上测试了 DES、3-DES 实现，记录了它们的 CPU 时间，得到了实验数据。测试数据为用户时间，时间接口为 Linux 系统中的 time 函数。

结果共有两个部分，分别是 DES 在 CPU 上和 GPU 上实现的时间。如图 3-9 所示，在 4MB 的明文中，GPU 上实现 DES 的时间约为 OpenSSL 的 1/2。在 697MB 的明文中，在 GPU 实现上可以获得 3～4 倍的性能提升，这意味着在更大的明文规模中使用并行算法能获得更大的效益。

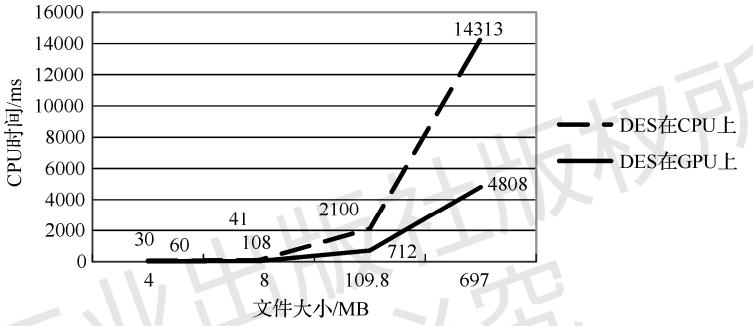


图 3-9 DES 性能对比

在 3-DES 的情况下，当处理大小接近 4MB 的文件时，DES 在 GPU 环境下的性能是 CPU 的 5 倍左右，如图 3-10 所示。此外，如果文件大小超过 700MB，它将获得超过 6 倍的性能。

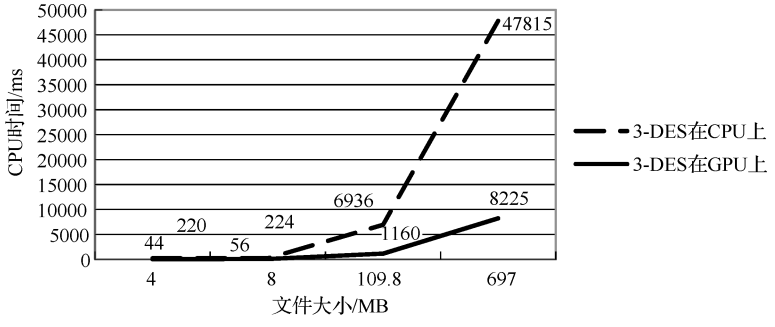


图 3-10 3-DES 性能对比

3.4　测试示例

3.4.1　参考常量

初始置换表：

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

逆初始置换表：

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

E 扩展置换表：

$$E=\begin{bmatrix}32&1&2&3&4&5\\4&5&6&7&8&9\\8&9&10&11&12&13\\12&13&14&15&16&17\\16&17&18&19&20&21\\20&21&22&23&24&25\\24&25&26&27&28&29\\28&29&30&31&32&1\end{bmatrix}$$

P 盒置换表：

$$P=\begin{bmatrix}16 & 7 & 20 & 21\\ 29 & 12 & 28 & 17\\ 1 & 15 & 23 & 26\\ 5 & 18 & 31 & 10\\ 2 & 8 & 24 & 14\\ 32 & 27 & 3 & 9\\ 19 & 13 & 30 & 6\\ 22 & 11 & 4 & 25\end{bmatrix}$$

S 盒表：

$$S_1=\begin{bmatrix}14 & 4 & 13 & 1 & 2 & 15 & 11 & 8 & 3 & 10 & 6 & 12 & 5 & 9 & 0 & 7\\ 0 & 15 & 7 & 4 & 14 & 2 & 13 & 1 & 10 & 6 & 12 & 11 & 9 & 5 & 3 & 8\\ 4 & 1 & 14 & 8 & 13 & 6 & 2 & 11 & 15 & 12 & 9 & 7 & 3 & 10 & 5 & 0\\ 15 & 12 & 8 & 2 & 4 & 9 & 1 & 7 & 5 & 11 & 3 & 14 & 10 & 0 & 6 & 13\end{bmatrix}$$

$$S_2=\begin{bmatrix}15 & 1 & 8 & 14 & 6 & 11 & 3 & 4 & 9 & 7 & 2 & 13 & 12 & 0 & 5 & 10\\ 3 & 13 & 4 & 7 & 15 & 2 & 8 & 14 & 12 & 0 & 1 & 10 & 6 & 9 & 11 & 5\\ 0 & 14 & 7 & 11 & 10 & 4 & 13 & 1 & 5 & 8 & 12 & 6 & 9 & 3 & 2 & 15\\ 13 & 8 & 10 & 1 & 3 & 15 & 4 & 2 & 11 & 6 & 7 & 12 & 0 & 5 & 14 & 9\end{bmatrix}$$

$$S_3=\begin{bmatrix}10 & 0 & 9 & 14 & 6 & 3 & 15 & 5 & 1 & 13 & 12 & 7 & 11 & 4 & 2 & 8\\ 13 & 7 & 0 & 9 & 3 & 4 & 6 & 10 & 2 & 8 & 5 & 14 & 12 & 11 & 15 & 1\\ 13 & 6 & 4 & 9 & 8 & 15 & 3 & 0 & 11 & 1 & 2 & 12 & 5 & 10 & 14 & 7\\ 1 & 10 & 13 & 0 & 6 & 9 & 8 & 7 & 4 & 15 & 14 & 3 & 11 & 5 & 2 & 12\end{bmatrix}$$

$$S_4=\begin{bmatrix}7 & 13 & 14 & 3 & 0 & 6 & 9 & 10 & 1 & 2 & 8 & 5 & 11 & 12 & 4 & 15\\ 13 & 8 & 11 & 5 & 6 & 15 & 0 & 3 & 4 & 7 & 2 & 12 & 1 & 10 & 14 & 9\\ 10 & 6 & 9 & 0 & 12 & 11 & 7 & 13 & 15 & 1 & 3 & 14 & 5 & 2 & 8 & 4\\ 3 & 15 & 0 & 6 & 10 & 1 & 13 & 8 & 9 & 4 & 5 & 11 & 12 & 7 & 2 & 14\end{bmatrix}$$

$$S_5=\begin{bmatrix}2 & 12 & 4 & 1 & 7 & 10 & 11 & 6 & 8 & 5 & 3 & 15 & 13 & 0 & 14 & 9\\ 14 & 11 & 2 & 12 & 4 & 7 & 13 & 1 & 5 & 0 & 15 & 10 & 3 & 9 & 8 & 6\\ 4 & 2 & 1 & 11 & 10 & 13 & 7 & 8 & 15 & 9 & 12 & 5 & 6 & 3 & 0 & 14\\ 11 & 8 & 12 & 7 & 1 & 14 & 2 & 13 & 6 & 15 & 0 & 9 & 10 & 4 & 5 & 3\end{bmatrix}$$

$$S_6=\begin{bmatrix}12 & 1 & 10 & 15 & 9 & 2 & 6 & 8 & 0 & 13 & 3 & 4 & 14 & 7 & 5 & 11\\ 10 & 15 & 4 & 2 & 7 & 12 & 9 & 5 & 6 & 1 & 13 & 14 & 0 & 11 & 3 & 8\\ 9 & 14 & 15 & 5 & 2 & 8 & 12 & 3 & 7 & 0 & 4 & 10 & 1 & 13 & 11 & 6\\ 4 & 3 & 2 & 12 & 9 & 5 & 15 & 10 & 11 & 14 & 1 & 7 & 6 & 0 & 8 & 13\end{bmatrix}$$

$$S_7=\begin{bmatrix}4 & 11 & 2 & 14 & 15 & 0 & 8 & 13 & 3 & 12 & 9 & 7 & 5 & 10 & 6 & 1\\ 13 & 0 & 11 & 7 & 4 & 9 & 1 & 10 & 14 & 3 & 5 & 12 & 2 & 15 & 8 & 6\\ 1 & 4 & 11 & 13 & 12 & 3 & 7 & 14 & 10 & 15 & 6 & 8 & 0 & 5 & 9 & 2\\ 6 & 11 & 13 & 8 & 1 & 4 & 10 & 7 & 9 & 5 & 0 & 15 & 14 & 2 & 3 & 12\end{bmatrix}$$

$$S_8=\begin{bmatrix}13&2&8&4&6&15&11&1&10&9&3&14&5&0&12&7\\1&15&13&8&10&3&7&4&12&5&6&11&0&14&9&2\\7&11&4&1&9&12&14&2&0&6&10&13&15&3&5&8\\2&1&14&7&4&10&8&13&15&12&9&0&3&5&6&11\end{bmatrix}$$

密钥置换 1 表：

$$\mathrm{PC}_1=\begin{bmatrix}57&49&41&33&25&17&9\\1&58&50&42&34&26&18\\10&2&59&51&43&35&27\\19&11&3&60&52&44&36\\63&55&47&39&31&23&15\\7&62&54&46&38&30&22\\14&6&61&53&45&37&29\\21&13&5&28&20&12&4\end{bmatrix}$$

密钥置换 2 表：

$$\mathrm{PC}_2=\begin{bmatrix}14&17&11&24&1&5\\3&28&15&6&21&10\\23&19&12&4&26&8\\16&7&27&20&13&2\\41&52&31&37&47&55\\30&40&51&45&33&48\\44&49&39&56&34&53\\46&42&50&36&29&32\end{bmatrix}$$

3.4.2　测试向量

本节中使用随机生成的密钥 K 进行加密。由于轮数较多，只给出初始置换结果、前两轮的加密结果以及最后的密文。

随机密钥 K：

10011001 11011011 10101000 01110001

11001000 01010110 11010011 01110000

输入文本：

example

明文 M：

01100101 01111000 01100001 01101101

01110000 01101100 01100101 00000000

初始置换

$L[0]$:

01111111 00010010 01101001 11001101

$R[0]$:

00000000 01111111 00101010 00000000

第 1 轮

轮密钥：

01010111 11111010 10001100 11100110

00100010 00000001

$L[1]$:

00000000 01111111 00101010 00000000

$R[1]$:

01000011 01100100 10010010 11101110

第 2 轮

轮密钥：

10111110 01110010 11110100 01001110

10000100 10001010

$L[2]$:

01000011 01100100 10010010 11101110

$R[2]$:

10101011 00000010 11010101 11100011

密文 C：

00111011 00011010 10001011 00011000

11100100 00001110 11101000 01001010

习题

3.1　Feistel 结构的安全基于哪些参数或设计？

3.2　DES 的 S 盒有什么作用？

3.3　对于第 6 个 S 盒，如果输入为 100111，那么输出是多少？如果输出是 0000，那么输入可能是哪些值？

3.4　DES 的 S 盒设计原理没有公开，S 盒设计应该遵循的一般原则是什么？如果由你来设计一个分组密码的 S 盒，你会用什么方法设计？

3.5　假设明文各位为 0，密钥各位为 1，求第 1 轮输出 L_1 和 R_1 的第 3 位。

3.6　假设 DES 初始密钥的十六进制形式为 d0 c2 b3 a4 57 68 91 79，请写出置换后的密钥。

3.7　DES 加密与解密之间有什么关系，请给出具体证明。

3.8　在 3.2 节介绍的 DES 快速实现方法中，在 AVX 上使用 SIMD 指令的好处是什么？在 64 位平台上不支持 SIMD，如何达到类似的效果？

3.9　简述 3-DES 的加/解密过程。

3.10　为什么要用 GPU 实现 3-DES？相比于 CPU 实现，GPU 实现方法的优势是什么？