DES 算法软件速度优化的实现

中国建设银行广西区分行　郑 焰

DES算法是银行过去常常使用的一种加解密算法，广泛地用于数据存取、交易保护和身份认证。尽管DES已经不再被认为是安全的加密算法，目前已经被逐步取代。但是由于目前硬件加密还无法覆盖小型机等大型系统，同时该算衍生出来的相关算法（如多重DES算法）至今仍被广泛应用，旧系统的数据转换仍然需要应用，因此DES类算法的软件优化在大型系统中仍然具备很高的借鉴价值。

众所周知，DES（Data Encryption Standard）算法是上世纪70年代由IBM公司发明的对称加密算法，经过NSA（美国国家安全局）的评估与修改，先后被NIST、ISO等权威机构作为工业标准发布，可以免费用于商业应用。关于DES算法的实现过程说明的文章很多，在本文中不再详述。

DES算法的基本过程包括以下几个部分：

初始置换

将输入的64位数据按位重新组合，并将重新组合后的数据分成左（Lo）右（Ro）两个部分，每个部分均32位。

密钥置换

从64位密钥中抽取56位(Ko)，每次密钥置换时，将K_{n-1}分为28bit的左右两部分，每部分分别循环左移1或2位，然后从移动后产生的56位秘钥中选出48位作为子密钥Kn。

扩展置换

将Rn切分为4位一组，将每组扩充至6位，最后组合成48位的数据R'n。

密钥异或

将R'n与子密钥Kn进行异或生成R''n。

S-Box置换

将R''n与输入指定的8个S-Box经过替换得到一个32位（8个4位分组）的输出R'''n。

P-Box置换

将R'''n输入到P-Box替换得到R''''n。

左右互换

将R''''n与Ln 异或得到Rn+1，将Rn赋值给Ln+1。

末置换

将L_{16}和R_{16}组合成64位数据后按照初始置换的逆过程重新按位组合得到最后的计算结果。

在实际应用中，存在着多种不同的DES的实现版本，最常见的就是取消初始置换和末置换，以及修改S-Box的内容，但无论如何以上几个基本过程在各个版本的程序中都可以清晰地看到轮廓。

通过对我们手头的DES程序代码进行分析，发现此版本对S-Box进行了部分修改并且裁减了S-Box的置换过程。由于优化后的DES程序必须与原来的程序在输入输出上完全保持一致，因此直接使用现成的高效标准的DES代码或者硬件设备是不可行的，只能对现有的代码进行优化。代码优化一般可以从数据结构、操作流程和语言风格几个方面着手，在本例中也是如此。

与绝大多数版本的程序一样，该版本的DES算法程序也是采用数组方式来实现位运算的。DES算法中包含了多种位运算，如位交换、扩充、压缩、异或等。应该说用数组方式实现位运算是最简单并且容易理解的方式。将参与计算的数据中的每一位分别对应到数组中的每一个元素，这样所有的位移操作都可以通过数组元素的交换完成，非常直观地实现了算法的描述过程，并且便于跟踪调试。这种方式虽然带来了实现上的便捷，但是增加了实际计算量。如在每次迭代过程中，扩展置换后的数组与子密钥的异或计算需要48次循环，P-Box置换后的左右部分交换计算则需要两个32次循环过程。

```
    void F(int n, char *ll, char *rr, char *LL, char*
RR)
    {
    int i;
    char buffer[64],tmp[64];

    /* 扩展置换 */
    for (i=0;i<48;i++) buffer[i]=rr[e_r[i]-1];
    /* 与子密钥异或 */
    for (i=0;i<48;i++) buffer[i]=(buffer[i]+K[n][i])&1;
```

```
    /* S-Box 置换 */
    s_box(buffer,tmp);
    /* P-Box 置换 */
    for (i=0;i<32;i++) buffer[i]=tmp[P[i]-1];
    /* 左右互换 */
    for (i=0;i<32;i++) RR[i]=(buffer[i]+ll[i])&1;
    for (i=0;i<32;i++) LL[i]=rr[i];
    }
```

数组实现方式的迭代过程

如果数据结构采用整型而不是数组类型实现的话,以上的循环操作可以简化成几个简单的赋值语句,这样将极大地提高计算速度。而用整型数据结构实现的难点在于置换中的位交换操作和数据的表示方式。前者可以通过逻辑或、逻辑与、位移操作实现,虽然在编码上的控制要比数组方式复杂一些,但是并没有增加计算量。在64位机器上,所有数据都可以通过一个长整型变量表示,在32位机上则要复杂一些,尤其在不支持long数据类型的编译环境下,必须通过两个32位的变量表示64位和48位的数据,对于超过32位的数据的计算也需要通过两个变量计算完成。

```
    void F(int n, unsigned long *l, unsigned long *r,
unsigned long *L, unsigned long* R)
    {
    unsigned long  buf[2], tmp2;

    /* 扩展置换 */
    EXPAND_32to48(*r, buf);
    /* 与子密钥异或 */
    buf[0]^=arr_key[(n)][0];
    buf[1]^=arr_key[(n)][1];
    /* S-Box 置换以及 P-Box 置换 */
    sp_box(buf, &tmp1);
    /* 左右互换 */
    *R = tmp2 ^ *l;
    *L = *r;
    }
```

整型实现方式的迭代过程

现在对以上两段代码进行分析比较。在第一段代码中除置换以外的操作共有112次循环操作,并且每个循环操作中还包括了数组寻址操作,而在第二段代码中的相同功能部分只有4次赋值操作,并且只有两次数组寻址操作。改成整型数据结构实现后,迭代过程中的计算量大幅降低。

实际运行结果证明,在改变了数据结构之后的运算速度是改造前的3～5倍。对于没有裁减过S-Box置换的标准DES算法,改造后的运算速度是改造前的10倍。

在完成数据结构改造之后,每次迭代过程中的计算量都集中在扩展置换、S-Box置换和P-Box置换中。如果能够降低这几个置换操作的计算量,整个算法的计算速度势必有进一步提高。

32 1 2 3 4 5 4 5 6 7 8 9 8 9 10 11 12 13 12 13 14 15 1617,

16 17 18 19 20 21 20 21 22 23 24 25 24 25 26 27 28 29 28 29 30 31 32 1

以上是扩展置换映射表,即将Rn的第32位赋值给R'n的第1位,将Rn的第1位赋值给R'n的第2位,以此类推。映射表中的下划线很清晰地反映了扩展置换中的规律。很显然,扩展置换完全不必依赖循环实现。以下为扩展置换的代码片断。

```
    void EXPAND_32to48(unsigned long in, unsigned long
out[2])
    {
    out[0]  =  out[1] = 0;
    out[0] = (out[0]<<1) + ((in)&1);
    out[0] = (out[0]<<5) + ((in>>27)&0x1f);
    out[0] = (out[0]<<6) + ((in>>23)&0x3f);
    out[0] = (out[0]<<6) + ((in>>19)&0x3f);
    out[0] = (out[0]<<6) + ((in>>15)&0x3f);
    out[1] = (out[1]<<6) + ((in>>11)&0x3f);
    out[1] = (out[1]<<6) + ((in>>7 )&0x3f);
    out[1] = (out[1]<<6) + ((in>>3 )&0x3f);
    out[1] = (out[1]<<5) + (in&0x1f);
    out[1] = (out[1]<<1) + ((in>>31)&1);
    }
```

S-Box置换的过程是将48位数据分成8组进行分组置换,每组6个字节,这6个字节计算后可得到一个下标值,根据下标值可以从该组对应的S-Box中查询到置换后的4字节的数值。S-Box由8个数组构成,每个数组中都由64个0～15之间的整数组成。例如第n组的值为$D_1D_2D_3D_4D_5D_6$,计算下标值的公式为$off = D_1D_6 \times 8 + D_2D_3D_4D_5$,而$D_1D_2D_3D_4D_5D_6$最后置换得到的结果应该是Sn[off]。P-Box置换则是将32位的数据重新按位排列。

如果完全按照算法描述来实现,S-Box置换过程将数据分成8组之后,要经过位移以及逻辑与或操作后获得下标值,然后在一个大数组中查询到结果,以上步骤重复8次,将每次

的结果合并，然后按P-Box的映射关系重新编排后得到最终结果。这个过程似乎没有太大的改进空间。但是事实并非如此。

首先，计算下标值的公式实际上是从一个64个整数的集合到另一个64个整数的集合的一一映射，只要构造了这个映射关系的数组，那么计算下标值的位移、逻辑与、逻辑或等操作都可以简化成一个数组寻址操作。

其次，虽然P-Box置换的映射关系乍一看毫无规律可言，除了通过循环按位逻辑操作之外别无他法，但是我们可以看到P-Box置换只是交换了各个位的位置，并没有增加、丢失或者覆盖任何数据位，那么我们完全可以将P-Box置换与S-Box置换放在一个操作中完成。即不是将各个S-Box置换后的数据合并之后再进行P-Box置换，而是设法使得每组置换后的到的数据就是已经按照P-Box置换的映射关系调整了数据位的32位整数，当然在这32位中仅有4位是有效位，其余28位都为0。这样我们在完成分组置换后得到的是8个32位的整数，每个整数中均仅有4位是有效位，其余28位都为0，并且这8个整数的有效位都没有重叠。最后将8个32位的整数进行逻辑或操作即可得到S-Box置换与P-Box置换后的最终结果。这种改造带来的好处就是减少了分组置换后合并结果的位移和逻辑与或操作，以及为实现P-Box置换而进行的循环、位移和逻辑与或操作。要实现以上构想只需要将分组置换的映射数组事先进行P-Box置换，产生新的由32位整数组成而不是由0～15组成的映射数组。

在完成置换过程的改进之后，迭代过程内部的算法优化基本完成。虽然初始置换和末置换并不会对算法的安全造成太大的影响，但是由于在本例中必须与原有程序的输入输出保持一致，所以对原有程序的初始置换和末置换进行改进还是很有必要的。

所幸的是，本例中所采用的初始置换和末置换的转换矩阵是有规律的，先让我们看看初始置换的映射矩阵：

```
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
```

让我们把这个矩阵稍微变化一下：

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

很显然，初始置换的本质实际上就是一个矩阵变形。

```
t=(((a>>n)^b)&m);
b^=t;
a^=(t<<n);
```

以上过程可以完成a和b之间的位交换。A和b中的数据位按n位分组，a中的偶数组数据位与b中的奇数组数据位互换。M为掩码，凡是与a中要交换至b的数据位均设置为1，其他位设置为0。例如$a = a_1a_2a_3a_4$，$b = b_1b_2b_3b_4$，如果n为1，则m应该为0x5，交换后$a = a_1b_1a_3b_3$，$b = a_2b_2a_4b_4$；如果n为2，则m应该为0x3，交换后$a = a_1a_2b_1b_2$，$b = a_3a_4b_3b_4$。

利用以上过程，我们可以将初始数据分成两个32位的整数，经过数次在这两个整数之间进行位交换，并在必要时可辅以其他逻辑位移操作即可完成原来需要通过64次循环操作才能完成的初始置换。同样，由于末置换是初始置换的逆过程，因此也可以通过以上操作的逆操作完成。

运行结果表明，在完成操作流程的改造之后，DES算法的计算速度又提高了2～3倍。

最后，又对代码的风格进行了整理，包括删除冗余代码，充分利用寄存器变量，大量使用宏定义替代函数。事实证明这最后的工作不是毫无意义的，在原先的基础上，DES算法的计算速度再次提高了一倍。

更正

本刊2005年第七期第73页“信息安全事件定级方法研究”一文图1刊登有误，现刊出正确的图予以更正。并向作者及读者致歉。