基于 Python 语言的 3DES 算法优化①

李爱宁，唐　勇，孙晓辉，刘昕彤

(河北工程技术高等专科学校 电气自动化系，沧州 061001)

摘　要：介绍了 3DES 加密算法的原理，描述了对 Python 语言自带模块 pyDES 模块中 3DES 算法的优化过程，分析了 3DES 算法编程实现过程中效率优化时所遇到的问题及解决方法；最后使用 Python 语言中的 profile 工具对优化后的算法进行了测试,数据表明优化后的 3DES 算法提高了原算法的加解密速度和安全性。

关键词：3DES 算法；Python；PyDES 模块；优化

Optimization of 3DES Cryptography Algorithm Based on Python

LI Ai-Ning, TANG Yong, SUN Xiao-Hui, LIU Xin-Tong

(Department of Electrical Automation, Hebei Engineering and Technical College, Cangzhou 061001, China)

Abstract: This paper discusses the theory of 3DES Cryptography algorithm, describes the optimization process of PyDES module, which is a pure python module that implements the DES and Triple-DES algorithms; and analyzes the problems and solutions encountered in the optimization process; Finally, the optimized algorithm was tested by profile which is a tool for python program testing, and the result shows that the optimized algorithm can Enhance the Security and speed of the original algorithm。

Key words: triple-DES algorithms; python program; PyDES module; optimization;

人类已进入信息化社会时代。数字化、信息化、网络化正在冲击、影响、改变我们社会生活的各个方面。信息安全越来越引起人们的关注，加密技术作为信息安全的核心，正发挥着重大的作用。加密算法分对称加密算法和非对称加密算法两种，其中对称密码算法具有加密、解密处理速度快、实时性强等优点。常用的加密算法有 DES、AES 等。

尽管 DES 已被证实是不安全的算法（主要是密钥太短），但三重 DES（3DES）增加了密钥长度，由 56 位增加到 112 或 168 位，有更高的安全性，而且在新一代因特网安全标准 IPSEC 协议集中已将 DES 作为加密标准。另一方面，基于 DES 算法的加/解密硬件目前已广泛应用于国内外卫星通信、网关服务器、机顶盒、视频传输以及其它大量的数据传输业务中。所以对 3DES 的研究仍有很大的现实意义。

1　3DES加密算法描述

DES 成为一个世界范围内的标准已经 20 多年了，很好地抗住了多年的密码分析，除最强有力的可能敌手外，对其它的攻击仍是安全的。DES 对 64 位的明文分组进行操作，通过一个初始置换，将明文分成左半部分和右半部分，然后进行 16 轮完全相同的运算，最后经过一个末置换便得到 64 位密文。每一轮的运算包含扩展置换、S 盒代换、P 盒置换和两次异或运算，另外每一轮中还有一个轮密钥（子密钥）。整体框图如图 1 所示。

3DES（即 Triple DES）是 DES 向 AES 过渡的加密算法（1999 年，NIST 将 3-DES 指定为过渡的加密标准），是 DES 的一个更安全的变形[1]。它以 DES 为基本模块，通过组合分组方法设计出分组加密算法，其具体实现如下：设 Ek(.)和 Dk(.)代表 DES 算法的加

① 基金项目:河北工专自然科学项目(XZ1003)
收稿时间:2010-11-11;收到修改稿时间:2010-12-23

密和解密过程，K 代表 DES 算法使用的密钥，P 代表明文，C 代表密文。则 3DES 算法的加解密过程可以表示为：

3DES 加密过程为：$C=E_{k3}(D_{k2}(E_{k1}(P)))$

3DES 解密过程为：$P=D_{k1}((E_{K2}(D_{k3}(C)))$

具体的加/解密过程如图 2 所示。K1、K2、K3 决定了算法的安全性。若三个密钥互不相同，本质上就相当于用一个长为 168 位的密钥进行加密。多年来，它在对付强力攻击时是比较安全的。若数据对安全性要求不那么高，K1 可以等于 K3。在这种情况下，密钥的有效长度为 112 位。

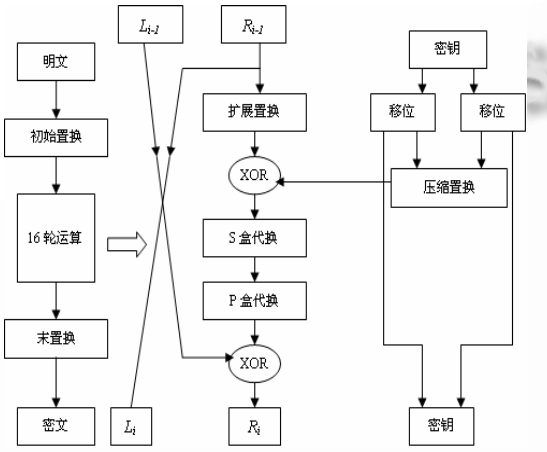

图 1 DES 加密算法框图

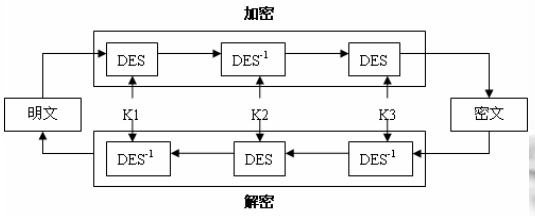

图 2 3DES 加/解密过程

2 Python语言简介

Python[2]是一种功能强大而完善的通用型程序设计语言。已经具有十多年的发展历史，成熟且稳定。Python 具有简洁的语法、丰富的扩展模块，可以使用 C/C++、java 等语言进行扩展，使用它可以大幅度缩短开发周期，节约开发成本。

Python 支持现有的各种主流操作系统，如 Microsoft Windows、Solaris、Mac OS、Linux 等，甚至包括 Palm OS 这样的嵌入式环境。它的源程序和二进制代码可以免费获得。

Python 为开发人员提供了丰富的模块，通过这些模块，我们可快速开发出功能强大的程序。Python 中用于加密的模块为 pyDES 模块[3]，该模块用来提供 DES、Triple-DES 的加密算法。

3 基于Python的3DES算法优化过程

3.1 密钥的处理

有密码学知识可知 DES 采用 64 位密钥技术，实际只有 56 位有效，8 位用来校验。但是，在实际应用中，用来生成 DES 算法密钥的原始密钥的位数是不确定的，因此应首先对原始密钥进行预处理，即用 SHA-1 算法生成原始密钥的摘要，这里的 SHA-1 使用 Python 自带的 hash lib 库中的 sha1 函数，该函数生成 40 个字节的字符串。这样，无论多长的密钥经过这一步处理后，都变成了固定长度的字符串。然后，字符串中的每个字符用低 7 位的 ASCII 值的二进制表示，形成长度为 280 的二进制串，将该 280 的二进制串分成 5 段，每段的长度为 56，每段逐一进行异或操作，得到 56 长度的二进制串，最后对这长度为 56 的二进制串进行置换操作，得到最终的预处理密钥。其过程可用上图 3 表示。

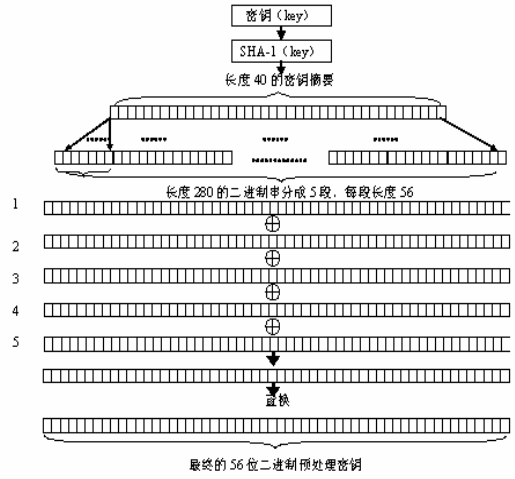

图 3 密钥处理过程

在最后的置换步骤中，从时间效率及安全性上考虑，对原来的 DES 密钥压缩置换表进行改造，如下表 1 和表 2 所示。

密钥经过这样预处理后，破译者即使得到原始密钥，在没有得到加解密模块或不知道密钥处理的方法之前，亦即不能得到真正有效的密钥，因而也很难对

密文进行破解。

表 1　压缩置换原表

0x0039	0x0031	0x0029	0x0021	0x0019	0x0011	0x0009	0x0001
0x003A	0x0032	0x002A	0x0022	0x001A	0x0012	0x000A	0x0002
0x003B	0x0033	0x002B	0x0023	0x001B	0x0013	0x000B	0x0003
0x003C	0x0034	0x002C	0x0024	0x003F	0x0037	0x002F	0x0027
0x001F	0x0017	0x000F	0x0007	0x003E	0x0036	0x002E	0x0026
0x001E	0x0016	0x000E	0x0006	0x003D	0x0035	0x002D	0x0025
0x001D	0x0015	0x000D	0x0005	0x001C	0x0014	0x000C	0x0004

表 2　修改后的压缩置换表

0x0032	0x002B	0x0024	0x001D	0x0016	0x000F	0x0008	0x0001
0x0033	0x002C	0x0025	0x001E	0x0017	0x0010	0x0009	0x0002
0x0034	0x002D	0x0026	0x001F	0x0018	0x0011	0x000A	0x0003
0x0035	0x002E	0x0027	0x0020	0x0038	0x0031	0x002A	0x0023
0x001C	0x0015	0x000E	0x0007	0x0037	0x0030	0x0029	0x0022
0x001B	0x0014	0x000D	0x0006	0x0036	0x002F	0x0028	0x0021
0x001A	0x0013	0x000C	0x0005	0x0019	0x0012	0x000B	0x0004

3.2 S-盒的处理

针对 Python 语言特性为了提高时间运行效率，将加密算法中 S-盒的数据全部预先转化成二进制串存储，这样运算时可以直接从中取出数据即可，省去了转化的过程，从而在一定程度上提高了加密的运行速度。

```
_S_BOX_=(
        (                    #S-1
('1110','0100','1101','0001','0010','1111','1011','1000','0011','1010','0110','1100','0101','1001','0000','0111'),
('0000','1111','0111','0100','1110','0010','1101','0001','1001','0110','1100','1011','1001','0101','0011','1000'),
('0100','0001','1110','1000','1101','0110','0010','1011','1111','1100','1001','0111','0011','1010','0101','0000'),
('1111','1100','1000','0010','0100','1001','0001','0111','0101','1011','0011','1110','1010','0000','0110','1101')
 ),

 (                           #S-2
('1111','0001','1000','1110','0110','1011','0011','0100','1001','0111','0010','1101','1100','0000','0101','1010'),
('0011','1101','0100','0111','1111','0010','1000','1110','1100','0000','0001','1010','0110','1001','1011','0101'),
('0000','1110','0111','1011','1010','0100','1101','0001','0101','1000','1100','0110','1001','0011','0010','1111'),
('1101','1000','1010','0001','0011','1111','0100','0010','1011','0110','0111','1100','0000','0101','1110','1001')
 ),

                                    .....

 (                           #S-8
('1101','0010','1000','0100','0110','1111','1011','0001','1010','1001','0011','1110','0101','0000','1100','0111'),
('0001','1111','1101','1000','1010','0011','0111','0100','1100','0101','0110','1011','0000','1110','1001','0010'),
('0111','1011','0100','0001','1001','1100','1110','0010','0000','0110','1010','1101','1111','0011','0101','1000'),
('0010','0001','1110','0111','0100','1010','1000','1101','1111','1100','1001','0000','0011','0101','0110','1011')
 )
  )
```

图 4　变换后 S 盒结构

(3) 3DES 算法编程实现过程中代码效率优化

针对被加密消息的大小，用 Python 自带的 profile 工具对 Python 语言自带的 pyDES 模块中 3DES 算法中各函数的运行时间作比较分析如下：

从以上可以看出，_xor，_sbox 和_expand 这几个函数占用了大多数的运行时间，尤其是_xor 函数随着明文串的增大所消耗的时间迅速增加。

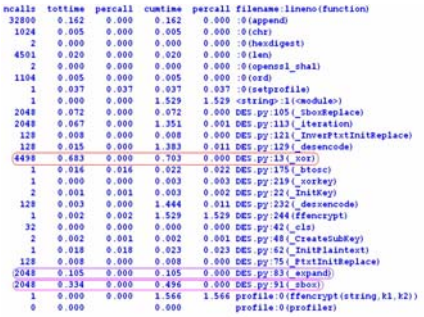

图 5　len(string)=1024 时运行时间

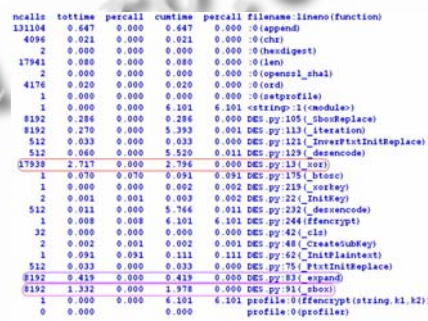

图 6　len(string)=4096 时运行时间

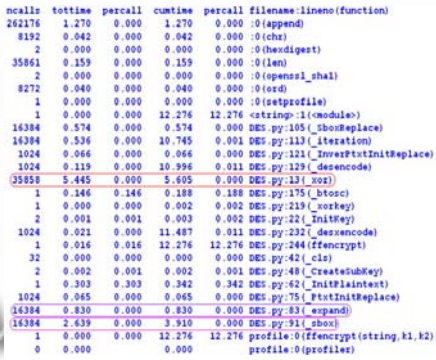

图 7　len(string)=8192 时运行时间

为此在以上分析的基础上，对_xor 和_sbox 这两个函数进行了改进：

```
#最初的 def _xor (FirstString, SecondString):
        …
xor=xor+str(int(FirstString[i])^ int(SecondString[i]))
        …
#改进后的
        def _xor (FirstString, SecondString):
        …
        if FirstString[i] == SecondString[i]:
            xor = xor + '0'
```

```
                else:
                    xor = xor + '1'
            ...
#最初的
    def _sbox (xer):
    …
    _row_ = (int (xer[i]) <<1) + int (xer [i+5])
_column_=(int(xer[i+1])<<3)+ (int(xer[i+2])<<2)+
(int(xer[i+3])<<1) +
    int(xer[i+4])
        row.append (_row_)
        column.append (_column_)
    …
  sbox= DEStable._S_BOX_[0][row[0]][column[0]] +
DEStable._S_BOX_[1][row[1]][column[1]] +
DEStable._S_BOX_[2][row[2]][column[2]]
DEStable._S_BOX_[3][row[3]][column[3]] +
DEStable._S_BOX_[4][row[4]][column[4]] +
DEStable._S_BOX_[5][row[5]][column[5]] +
DEStable._S_BOX_[6][row[6]][column[6]] +
DEStable._S_BOX_[7][row[7]][column[7]]
    …
#改进后的
    def _sbox (xer):
        i = 0，j = 0
        sbox = ''
        …
        _row_ = (int (xer[i]) <<1) + int (xer [i+5])
_column_=(int(xer[i+1])<<3)+ (int(xer[i+2])<<2) +
                (int(xer[i+3])<<1) +
            int(xer[i+4])
sbox=sbox+ DEStable._S_BOX_[j][_row_][_column_]
    …
```

下面对改进后的系统作性能测试。用 profile 工具对 8192 长度的字串作分析，如下图 8 所示。

与改进之前的 8192 长字串的测试输出相比较，可以看到，两个函数的调用次数没有改变，但是运行时间有了较大改善，尤其是_xor 函数改变的非常明显。改进后，我们看到 append 操作的次数显著减少，因而加快了运算速度。

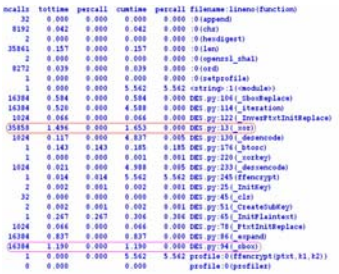

图 8 len(string)=8192 时运行时间

4 测试结果及结论

下面给出改进前和改进后的 3DES 测试结果：

测试环境：

硬件：AMD Sempron 2600+，768MB DDR400

软件：Windows XP SP2，Python 2.5

设定：K1=‘12345678’，K2=‘87654321’

测试用代码：

```
#测试 3DES
def perf_3des (string, k1, k2):
    print 'Test is running...'
    start=time.time()
    encrypt (string, k1, k2)
    end=time.time()
    perf=end-start
    print 'Test finished.\nThis time use %s
seconds'%perf
```

表 3 所示为改进前和改进后的两次测试结果（取小数点后 4 位）,测试数据可以看出，与改进之前相比改进之后的性能提高了一倍多。

表 3 测试结果

		3DES (seconds) 第一次	3DES (seconds) 第二次
len(string)=256	1	0.671	0.344
	2	0.687	0.343
	3	0.671	0.328
	4	0.671	0.344
	5	0.672	0.344
	AVG	0.6744	0.3406
len(string)=1024	1	2.703	1.343
	2	2.719	1.344
	3	2.719	1.344
	4	2.719	1.343
	5	2.703	1.343
	AVG	2.7126	1.3434
len(string)=2048	1	5.422	2.687
	2	5.438	2.687
	3	5.437	2.688
	4	5.421	2.687
	5	5.437	2.688
	AVG	5.431	2.6874

改进后的 3DES 算法对密钥的生成过程进行了改

(下转第 173 页)

的 I/O 等待情况。

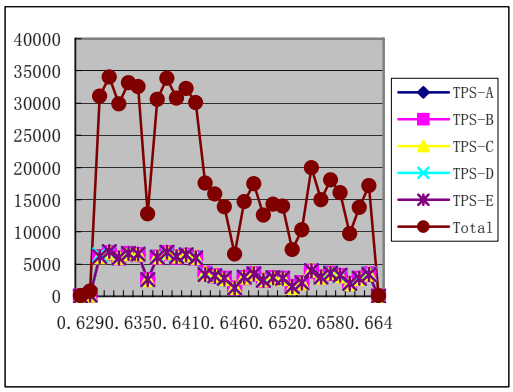

图 6 磁盘数据库累加系统测试结果

图 5 显示的 MDB 实时累加系统在整个数据处理过程中，波形比较平滑，无明显的 I/O 操作，系统的吞吐量也无明显的下降，I/O 瓶颈问题得以解决可以很大的提高系统的效率；而图 6 的 I/O 传输太过频繁，出现了大量的等待 I/O 操作，而且系统吞吐量有明显的下降趋势，这对处理每天上亿条话单的海量数据很不利，无法满足实时高效的要求。

由对比测试的结果可以看出，应用了内存数据库技术的 MDB 实时累加系统可以满足目前实时处理海量话单的要求，可以达到预期的效果。

4 结语

本文研究并详细分析了内存数据库在实时累加系统中的应用技术，以及为适应具体应用环境而作出的创新，并且从测试结果中也可以看出，内存数据库技术在解决海量数据处理及实时应用中有着很大的优势。

目前，电信行业已经进入 3G 时代，随着 3G 技术越来越成熟，移动 BOSS 系统要处理来的数据量必然激增。业务种类的增多、用户群的增长、通信技术的革新等等，都对 BOSS 系统的提出挑战，而本文对内存数据库面向应用的研究，在这种发展背景下有着一定的实用价值。

参考文献

1 王珊,肖艳芹,刘大为,覃雄派.内存数据库关键技术研究.计算机应,2007,27(10):2353–2357.

2 肖迎远.分布式实时数据库技术.北京:科学出版社,2009.94–103.

3 肖迎远,刘云生,廖国琼.主动实时内存数据库系统的数据交换策略及实现.计算机工程与应用,2004,40(29):11–14.

4 Hong DK, Chakravarthy S, Johnson T. Incorporating load factor into the scheduling of soft real-time transactions for main memory databases. Information Systems, 2000,25(4):309–322.

(上接第 187 页)

进，提高了算法的安全性和执行的效率，目前正在试用于我们正在研发的智能 Agent 安全机制[4-6]中。

参考文献

1 Stallings W. Cryptography and Network Security Principles and Practices.北京:电子工业出版社,2006.

2 Van Rossum G. An Introduction to Python. Network Theory Ltd 2003,4.

3 http://pydes.sourceforge.net.

4 Li AN, Zhao ZM. Research and design of security mechanism for multi-grade Agent system. The 11th IEEE International Conference on Communication Technology Proceedings. 2008.777-780.

5 李爱宁,赵泽茂.分布式网络中智能代理的安全迁移机制.计算机工程与科学,2009,31(1):101-103.

6 李爱宁,赵泽茂.基于 RBAC 模型的多等级移动 Agent 系统访问控制机制.计算机系统应用,2009,18(7):23-27.