单表替换密码及其统计分析实验报告

MonoAlphabeticCipher

作者：张佳佳
学号：PB14210032
所在院系：信息科学与技术学院信息安全系
指导老师：李卫海
完成时间：2016/12/30

摘要:单表替换密码作为古典密码中十分经典的一种，曾经有着广泛的应用，本文就单表替换密码的原理使用C语言实现了单表替换密码的源程序，同时就单表替换密码的漏洞之一------字母统计频率较为固定实现了单表替换密码的统计分析和破解。

目录：

1. 成员及分工
2. 关键技术
3. 实验结果及其分析
4. 性能分析
5. 创新点
6. 参考文献

成员及分工：

成员数目：1
成员姓名：张佳佳
成员学号：PB14210032
本次实验由本人独立完成，未以组队的形式合作完成，其中不免有疏漏之处，望见谅。

关键技术：

开发条件：

1. 开发环境： `Ubuntu 16.04`
2. 开发工具： `gcc` 、 `make` 、等开发工具
3. 测试环境： `linux` (详)， `windows` (略)

项目结构：

- Encrypt
  ---encrypt
- Decrypt
  ---decrypt
- Attack
  ---attack
- publicFuncLib
  ---publib
  其中 `Encrypt` `Decrypt` `Attack` 基本属于包装层，主要函数及功能实现在共用函数库publib中，其中依赖关系为：
  `encypt` : encrypt.o publib.o
  `decrypt` : decrypt.o publib.o
  `attack` : attack.o decrypt.o publib.o

加/解密功能技术实现:

加解密功能主要依赖于单表替换，故抽取取替换功能作为函数 `Substitution` 作为公共函数库publib的一个模块，当需要加密时，调用替换模块，当需要解密时，先用公共函数库publib里的 `GetDecryptPwd()` 函数逆转加密密码，之后再调用替换模块反向替代即可解密得到原文。代码如下：

```
void encrypt(char *IFname, char *OFname, char *passwordFile)
{
    /*check and read the password*/

    char (*pwd)[NUM_OF_LETTERS];
    pwd = CheckAndReadPwd(passwordFile);

    /*process of substitution*/

    Substitution(IFname, OFname, pwd);
}
```

```
void decrypt(char *IFname, char *OFname, char *passwordfile)
{
    /*get decrypt password from encrypt password*/

    char (*decryptPwd)[NUM_OF_LETTERS];
    decryptPwd = GetDecryptPwd(passwordfile);

    /*substitution using decrypt password to get the plaintext*/

    Substitution(IFname, OFname, decryptPwd);
}
```

通过将主要功能模块提取出来，加解密模块的代码得到了极大的简化，且更加的利于理解。
加解密过程首先是要将文档中的密码提取出来，存储为二维数组，表示替代关系的一一对应，大致示意图如下：

A	B	C	...	Z
A'	B'	C'	...	Z'

二维数组双映射关系的数据结构使得后期的处理更加的简便，且通过将二维数组转换成**hash表**，替换操作的运行时间可以得到极大的降低，这是典型的以空间换时间的做法;并且支持所有char型可打印字符范围内的加密和解密，且可扩展性更好。

频率统计分析攻击技术实现实现：

- **频率统计过程所使用的数据结构：**

```
struct Dict
{
    char Key[NUM_OF_LETTERS];
    double Value[NUM_OF_LETTERS];
};
```

使用的是键-值结构体的字典模式，便于统计过程中的字符对应，而且由于统计过程中不断的需要反向查找字符对应出现的下标，以确定在Value的那个位置加一统计次数，若采用传统的方法如：

```
int GetIndex(char *array, char ch)
{
    int i;
    for (i = 0; i < NUM_OF_LETTERS; i++)
        if (ch == array[i])
            return i;
    return -1;
}
```

此种传统的查找下标的方法虽然也可以胜任，但显然平均查找长度为$26\times1/2=13(O(n/2))$, 即每统计一个字符，平均需要13次比较，频率统计的下标查找作为最主要也是运行最多的步骤，显然这样开销较大;与加解密过程相同，此处利用char型数据属于整型数据族的特点，可以以字符作为键值建立**hash表**,在增加不大的空间冗余的情况下，获得O(1)的查找时间，极大的降低了统计频率过程的时间复杂度。

- **频率统计分析获取密码的方案：**

显然频率分析最直观的方案是单字符比较，若采用多字符比较，则方案较为复杂，且需要构造多字符比较的字典。鉴于课本仅给了单字符的频率字典，故此采用单字符比较。
由此我们先分析单字符比较情形下如何获取最优解，为此需要借用下信息论的知识。
我们已经知道，课本已经给出了26个字符超大量的统计频率，因为统计数量十分的大，不妨认为等同于26字符的真实分布概率，由此我们可以得到一个概率表：

A	B	C	...	Z
p1	p2	p3	...	p26

同时理论上我们可以根据实验频率统计得到许多频率表：

A'	B'	C'	...	Z'
q1	q2	q3	...	q26

为得到最优解，我们首先需要对这许多的频率表与上面的分布概率表做一个比较，不难想到，在信息论当中，**相对熵**表示的就是当真实分布为p时，假定分布为q时的无效性，是两个随机分布的距离。显然，通过求概率分布和频率表的相对熵可以很好的得出这个频率分布与真实分布的贴合程度，相对熵越大，则贴合程度越小，相对熵越小，则贴合程度越大;依据此原理，我们只需要找到相对熵最小的那个频率分布，则其肯定是单字符比较的最优解;若要求不止一个解，则可以取相对熵最小的几个。
到此为止，我们已经解决了已知频率表求最优解的问题，那么我们该如何生成频率表，考虑所有可能出现的频率表，总共有26！，这些表是对于频率的一个排列，显然取这么大的频率表求解最优解是理论可行的，但是运算代价太高，为此我们需要先对频率表进行一个筛选。
依然藉由信息论的知识，由相对熵的凹凸性，当一个分布越平缓的接近于真实分布时，相对熵的值越小，这就提醒我们，不需要对那些频率分布差别较大的字符做交换排列生成频率表，由此我们可以将我们的排列限制在一定的范围，假如我们先将频率表按从大到小排列，则我们的交换将会在一个较小的范围内;
例如：

1	2	3	...	26
q1	q2	q3	...	q26

交换为：

2	*1*	3	...	26
q1	q2	q3	...	q26

这显然是符合我们的日常经验的，然而若全部做完这种交换的排列，数目也是极大的，有2^26 个，我们需要对其更进一步的简化。
再次利用相对熵的凸凹性，为使频率表更平滑，显然我们不需要做完这些所有的交换，只需要交换那些间隔差值比较小的，显然这也是符合我们认知的，因为频率差别越小，越容易发生错误，通过交换，很可能得到的结果更接近最优解，由此我们也彻底地解决了频率表的问题;具体的实现方案为：
1. 找出频率间隔最小的两个频率，做互换，生成新表
2. 找出频率间隔次小的两个频率，做互换，生成新表
3. ...
本次实验中，我共做了四轮，生成了16个频率表，并依据相对熵的大小从中抽取了10个作为最优解，也是本次攻击给出的最优的10组密码

实验结果及分析：

实验结果:单就密码而言，对长文本，密码中出现概率最高的字符正确率较高，出现概率较低的字符正确率略低，故对整个文本而言，出错率不高。对短文本，破解出的的密码出错率偏高，但全文出错率要好一点。
分析：由上面的分析可知，由于此解为单字符的最优解，故长文本会有较高的正确率，但是对于短文本而言，由于单字符比较本质上的缺陷(无法利用字符间的联系)，所以密码出错率较高，但是由于此法优先确定高频率字符，在全文出错率方面有一定的弥补作用。

性能分析：

由上面的分析可知，对本实验出现最多的操作 `字符替换` 和 `频率统计` 方面，均采用了**hash表**的结构，采用空间换时间的方式，极大的提升了程序的性能，百万词以内的加解密和攻击，基本在几秒以内即可完成(在ubuntu上测试)。

创新点：

1. 引用信息论里相对熵的观点，从理论上深刻剖析，极大简化了密码的评价系统，减少了大量不必要的计算量。
2. 程序关键部分采用了hash表的结构，从性能上有了极大的提升。

参考文献：

1. 课本 密码编码学与网络安全---原理与实践 [美] William Stallings著 李莉 杜瑞颖等译
2. 课本 信息论基础 [美] Thomas M.Cover Joy A.Thomas著 阮吉寿 张华译