数据安全第一次大作业实验报告

张书樵：Enigma 破解
张万豪：C 语言实现分组密码和 Hash 函数
罗子秋：SM3 口令破解

2023 年 4 月 20 日

1 Enigma 破解

1.1 背景

Enigma 机是二战时德军使用的加密装置。它拥有三个可以转动的转子，一个固定的反射器，插线板，键盘，灯板等组件。插线板能够交换两个字母，转子可以将一个字母映射为另一个字母。这都是通过电路实现的。使用者点击键盘上的按键输入明文字母后，机器中电池产生的电流将通过插线板、三个转子、反射器，然后折返，再次通过三个转子和插线板，将灯板上的密文字母对应的灯泡点亮。每次按下按键时，转子会产生旋转，并且在到达特定位置时会产生进位。因此，每次加密时导通的电路会发生变化，使得 Enigma 机的加密过程难以被破解。

在二战后期，德军使用的 Enigma 机允许操作者从多个转子中选择 3 个，每个转子有 26 种初始位置，并且插线板上可以交换 10 组字母。这产生了一个巨大的密钥空间，看上去使 Enigma 机坚不可摧。然而，波兰数学家 Rejewski 在 30 年代就发现了 Enigma 机操作流程上存在的弱点，并针对其设计了破解方法。在二战爆发后，相关的技术被共享给了盟军的情报机构，但德军也加强了 Enigma 密码的安全性。英国数学家和计算机科学家 Turing 进一步地设计了已知明文攻击的方法，成功破解了 Enigma 机，为世界反法西斯战争的胜利做出了重要的贡献。本次作业便复现了 Rejewski 和 Turing 破解 Enigma 机使用的方法。

1.2 算法原理

1.2.1 Rejewski 的破解方法

在上世纪三十年代，Enigma 机的操作流程是：每天所有人都会使用相同的日密钥，但日密钥不用于加密信息；在发送一条信息时，操作者需要先随机生成三个字母的信息密钥。操作者会首先使用日密钥，将信息密钥输入两次，产生的 6 个字母的密文作为开头。随后，操作者将机器的转子调整为信息密钥所对应的转子位置，并开始加密信息。信息密钥输入两次的原因是为了避免信息传输中因为干扰出现错误，收信方如果发现通过日密钥解码出的前 6 个字母明文不是重复两次的格式，则可以发现问题。但是，重复是加密的敌人，Rejewski 敏锐地观察到了这一点。第一个字母和第四个字母是同一个明文加密出来的，并且输入第一个字母时，转子的设置都是日密钥，是完全相同的。于是，在某一天中收到的所有由 Enigma 机加密的密文，都会满足类似这样的规则：

- 如果第 1 个字母为 A，那么第 4 个字母为 E；
- 如果第 1 个字母为 B，那么第 4 个字母为 L；
- 如果第 1 个字母为 C，那么第 4 个字母为 C；
- $\cdots$

A	B	C	D	E	F	G	H	I	J	K	L	M	N	$\cdots$
E	L	C	O	N	W	D	I	A	P	K	S	Z	H	$\cdots$

表 1: 密文的第一个字母和第四个字母的对应表格（部分）

将结果列出，如表1所示。Rejewski 发现，表中的字母存在一些“链”，例如，上一行的字母 A 对应下一行的字母 E，上一行的字母 E 对应下一行的字母 N，以此类推，最后又回到字母 A。这样就产生了字母链 AENHI。同时，也可以找到其它的字母链：BLSJP，C，DOFWVG，K，MZURTY，Q，X。如果机器的初始状态相同，那么得到的字母链自然也是相同的。更重要的是，Rejewski 发现，插线板的存在只会改变字母链中的字母，但不会影响各个字母链的长度。[1] 例如，在上面的场景中，额外将 A 和 C 之间加入一条接线，那么，原先的 AENHI 链会变为 CENHI，C 链会变为 A 链；但两条链的长度是不变的。因此，字母链的长度特征是插线板作用下的不变

量，它只和转子的初始位置有关，由此可以将插线板的作用和转子的初始位置解耦合。波兰的破译团队用了一年的时间进行预计算，得到了所有转子初始位置与字母链的长度的对应关系。此后，在收到新的密文后，破译者可以查表，将字母链的长度与预计算的结果进行比较，得到可能的转子的初始位置。在成功恢复出转子初始位置后，破译者可以对密文进行解密，之后进一步地分析插线板的状态。具体的算法原理见算法1。

1.2.2 Turing 的破解方法

在波兰沦陷之前，Rejewski 的方法被共享给了英国和法国人。英国情报部门认为德军很快会发现重复输入信息密钥的漏洞，并改进 Enigma 的操作流程，因此 Turing 接到的任务是发明一个不依赖于这个漏洞的破解方法。

Turing 发现，每天早上 6 点左右，德军都会发送天气预告，其中包含固定的德语单词 Wetterbericht。这个已知明文单词在信息中的位置会略有变化，但 Enigma 机有一个重要的特点：字母加密之后不会是其自身。因此，通过将 Wetterbericht 在密文中进行比较，就可以得到其可能出现的合法位置，这就相当于获得了一段明文和对应的密文。

将一段已知的明文和密文列出来，也可以得到一张表，如表2所示。同样的，这张表上也存在字母圈。例如，第 5 列 B 加密为 J，第 8 列 J 加密为 X，第 12 列 X 加密为 B。根据 Enigma 机的原理，明文字母在输入进转子前，会经过插线板；密文字母在从转子输出后，也会经过插线板。这种加密状态具体如图1所示。

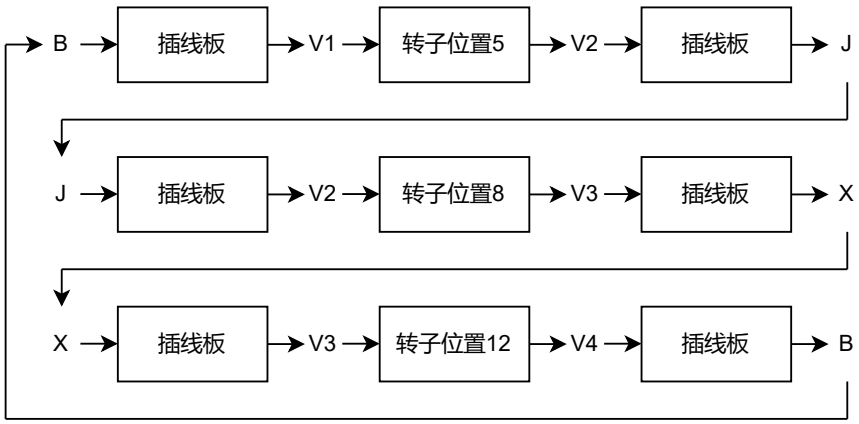

图 1: 字母圈对应的加密过程

假设 B 经过插线板后变为字母 V1，而 J 是由字母 V2 经过插线板变换得到的。由插线板的性质，字母 J 再次经过插线板后，会再次变为字母 V2。以此类推，我们也可以得知，V4 与 V1 应当是相等的，虽然我们还无法确定 V1 是哪个字母。因此，由这个由字母 B，J，X 组成的圈可以推导出以下结论：

> 存在字母 V1，使得其经过转子位置 5、转子位置 8、转子位置 12 的加密后，得到的结果是其自身。

而插线板的贡献，则完全被抵消掉了。因此，像 Rejewski 一样，Turing 也成功地将暴力破解密钥的搜索空间减小到了一个可以接受的范围。接下来要做的事情，便是枚举每一种转子的初始位置，验证是否存在满足要求的字母 V1。并且，这种循环的字母圈可能不止一种。只有当猜测的初始位置满足所有的字母圈约束，它才可能是正确的解。具体的算法原理见算法2。

1.3 实际样例的攻击过程和结果

通过 Python 语言编程实现了 Rejewski 和 Turing 的破解方法。Enigma 机的参数，包括转子和反射器的内部连接基于题目的要求设计。[1] 实际样例的攻击是固定 Ring Setting 为 D-E-S，恢复转子顺序和 Initial Position。程序运行在 Apple M1 Pro 的 CPU 上，由于计算量不大，没有设计多进程并行计算加速。

1.3.1 Rejewski 的破解方法

程序首先根据密文的第一个字母和第四个字母，第二个字母和第五个字母，第三个字母和第六个字母的对应表格，找到循环圈。随后，程序遍历所有的转子顺序和 Initial Position，为不同状态的 Enigma 机生成对应的字母循环圈。程序自动根据循环圈中的字母数量，比对结果，并将比对成功的 Enigma 机设定输出。程序运行耗时 27 秒，得到的解为：转子顺序为 II-III-I，Initial Position 为 A-A-A。程序输出了唯一解，且与题目中给出的初始设置符合。

[1] 经过比对，题目设定的机器型号为 Enigma I/Enigma M3。

1.3.2 Turing 的破解方法

程序根据已知的明文和密文，搜寻字母圈。随后，枚举转子的初始状态，寻找是否有状态能够满足所有的字母圈规则。程序运行耗时 74 秒，得到的解为：转子顺序为 II- III-I，Initial Position 为 A-A-A。程序输出了唯一解，且与题目中给出的初始设置符合。

1.4 代码文档

全部代码见 `enigma` 文件夹。其中，`enigma.py` 中定义了 Enigma 机模拟器和转子行为的实现，`rejewski.py` 中实现了 Rejewski 的破解方法，`turing.py` 中实现了 Turing 的破解方法。

2 C 语言实现分组密码和 Hash 函数

2.1 分组密码 AES 实现

高级加密标准（Advanced Encryption Standard，AES），又称 Rijndael 加密法，是美国联邦政府采用的一种分组加密标准。这个标准用来替代原先的 DES，已经被多方分析且广为全世界所使用。

AES 作为一种分组加密算法，在加密时，会首先将数据分成大小相等的组，然后对每组数据分别加密。在 AES 标准规范中，每组数据的大小是 16 字节，而加密所用密钥可以是 128 位、192 位、256 位。不同长度密钥加密流程相同，但是加密所需的轮数不同，128 位的密钥加密需要 10 轮，192 位密钥加密需要 12 轮，256 位密钥加密需要 14 轮。下面以 128 位密钥加密为例，介绍 AES-128 算法的实现。

AES 算法流程如图2所示，当密钥长度为 128 位时，Nr = 10，即会执行 10 次轮函数，前 9 次轮函数操作都是一样的，第 10 次轮函数中没有列混淆操作。AES 的核心就是实现轮函数中的所有操作：字节替换、行移位、列混淆和轮密钥加，以及实现密钥扩展。以下功能都在 AES.c 中实现。

2.1.1 密钥扩展

由函数 int keyExpansion(const uint8_t *key, uint32_t keyLen, AesKey *aesKey) 进行密钥扩展实现

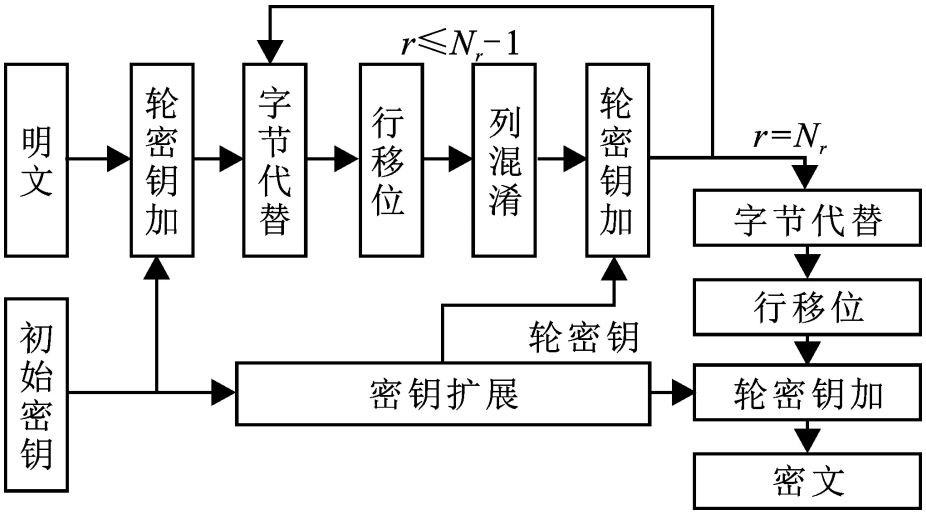

图 2: AES 算法加密流程

输入的密钥为 128 位 (也就是 4 个字)，扩展的密钥 w 将为 44 个字。其中前四个字 w[0-3] 就是原始密钥，剩下的 w[4-43] 会按照以下规则迭代获得：若 i%4=0, 则 $w[i] = w[i-4] \oplus g(w[i-1])$, 否则 $w[i] = w[i-4] \oplus w[i-1]$; 其中 $g(w[i-1])$ 函数包括三步操作：

1) 将 $w[i-1]$ 循环左移 8Bit；

2) 然后分别对每个字节做 S 盒置换；

3) 再与 32 比特的常量 $rcon[i/4-1]$ 进行异或，其中 $rcon$ 是一个长度为 10 的一维数组；

需要注意的是，256 位密钥扩展实现和 128 位、192 位稍有不同，具体可以参考 AES 标准文档 [2]，本函数的具体实现采用文档中算法 Figure 11. Pseudo Code for Key Expansion. 对 128 位、192 位、256 位做了统一实现。

2.1.2 轮密钥加

由函数 int addRoundKey(uint8_t state[][4], const uint32_t *key) 实现

对于一个 16 字节的数据块，AES 首先会将它按列存储到 4x4 的数据矩阵中，即使用 $(uint8\_t)state[4][4]$ 中存储，轮密钥加函数的功能是将数据矩阵与秘钥矩阵中每个元素进行异或操作，再存储到数据矩阵中。

2.1.3 字节替换

由函数 int subBytes(uint8_t state[][4]) 实现

字节替换是一个非线性的替换步骤，我们定义了一个有 256 字节的 S 表数组，字节替换的操作就是通过查询 S 表替换掉数据矩阵中的每个值，即对于数据矩阵中的每个元素 state[i][j] ($0 \leq i < 4, 0 \leq j < 4$)，令 state[i][j] = S[state[i][j]];

2.1.4 行移位

由函数 int shiftRows(uint8_t state[][4]) 实现

行移位的操作如图3所示，在加密时，保持数据矩阵的第一行不变，第二行向左移动 8Bit(一个字节)、第三行向左移动 16Bit(2 个字节)、第四行向左移动 24Bit(3 个字节)。行移位是 AES 的扩散层，其目的是将单个位上的变换扩散到整个状态，从而达到雪崩效应。

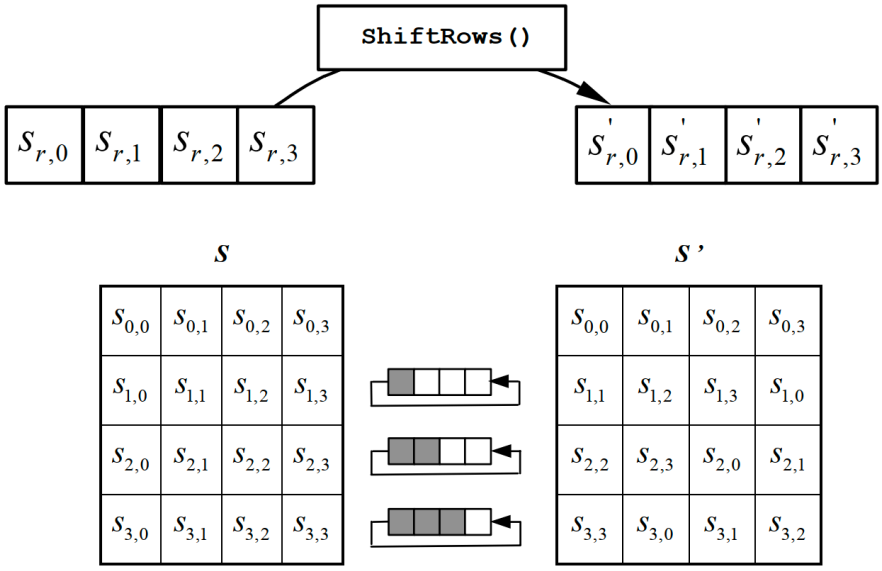

图 3: 行移位

2.1.5 列混淆

由函数 int mixColumns(uint8_t state[][4]) 和 uint8_t GMul(uint8_t u, uint8_t v) 实现

列混淆的操作是将数据矩阵左乘一个固定的矩阵 (1)，得到混淆后的矩阵，这个过程也可以简化成 (2) 的表达式，其中矩阵的乘法和加法并不是通常意义上的乘法和加法，而是定义在伽罗华域上的二元运算，加法 ($\oplus$) 是异或操作，而其中的乘法 ($*$) 由 uint8_t GMul(uint8_t u, uint8_t v) 实现。

$$\begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \tag{1}$$

$$\begin{cases} s'_{0,c} = (2 * s_{0,c}) \oplus (3 * s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\ s'_{1,c} = s_{0,c} \oplus (2 * s_{1,c}) \oplus (3 * s_{2,c}) \oplus s_{3,c} \\ s'_{2,c} = s_{0,c} \oplus s_{1,c} \oplus (2 * s_{2,c}) \oplus (3 * s_{3,c}) \\ s'_{3,c} = (3 * s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (2 * s_{3,c}) \end{cases} \quad 0 \le c < 4 \tag{2}$$

2.2 分组密码 SM4 实现

SM4 是由我国国家密码管理局发布的一个国密算法，其标准文档可以从国家标准全文公开系统获得。SM4 密码算法同 AES 类似，是一个分组算法。该算法的分组长度为 128 Bit（16 字节），密钥长度为 128 Bit（16 字节）。加密算法与密钥扩展算法均采用非线性迭代结构, 运算轮数均为 32 轮。本文实现的是 SM4 的 ECB 模式。其算法流程比较简单，下面介绍 SM4 的加密算法步骤。

同 AES 类似, SM4 算法也需要一个 S 盒，实现为一个长度为 256 的一维数组。然后根据用户输入的 128 位密钥进行轮密钥扩展，需要将其将其分为 4 组，之后对应的与系统参数 FK 进行异或运算，将产生的结果进行 32 轮迭代生成 32 个轮子密钥，供后面 32 轮迭代循环使用。这里通过 SM4.c 中的 int sm4_set_key(const uint8_t *key , sm4_ctx * const ctx) 实现，扩展的轮密钥存入到 ctx->rk 中。

然后将将 128Bit 的明文分成 4 个 32bit 的字 X1,X2,X3,X4，对其执行 32 次迭代运算，每次迭代运算的操作可以表达为式 (3) 所示。具体 F 函

数实现为 SM4.c 的 static inline void SM4_F(uint32_t * const blks, const uint32_t *rkg) 函数。其中 rk_i i 。

$$X_{i+4} = F(X_i, X_{i+1}, X_{i+2}, X_{i+3}, rk_i), 0 \leq i < 32 \tag{3}$$

最后经过一轮反序变换得到密文，实现即 SM4.c 中 void sm4_encrypt(const uint8_t *in, uint8_t *out, const sm4_ctx *ctx) 执行完 for 循环后的操作。

2.3 Hash 函数 SHA-256, SHA-512 实现

SHA-2，名称来自于安全散列算法 2（英语：Secure Hash Algorithm 2）的缩写，一种密码散列函数算法标准，由美国国家安全局研发，由美国国家标准与技术研究院（NIST）在 2001 年发布。属于 SHA 算法之一，是 SHA-1 的后继者。SHA-2 家族分为六个不同的算法标准：SHA-224、SHA-256、SHA-384、SHA-512、SHA-512/224、SHA-512/256。本文使用 C 语言实现了 SHA-256 和 SHA-512，

SHA-512 的消息扩展算法和压缩函数都比 SHA-256 更复杂，SHA-512 的消息扩展算法使用了 80 个常数，而 SHA-256 只使用了 64 个常数，但是二者的计算 Hash 值的流程相同，下面以 SHA-256 为例介绍具体实现，代码在 SHA2.c 的 int sha256(unsigned char *message) 函数中。

2.3.1 消息预处理

SHA-256 首先需要对输入的信息进行填充，在消息的末尾进行 Bit 填充，首先填充一个 1，然后填充若干个 0，使得消息长度对 512 取模的余数为 448，需要注意的是即使在消息未填充的时候已经满足消息长度对 512 取模的余数为 448，仍需要填充 1 位 1，所以填充消息的长度最短为 1Bit，最长为 512Bit。

完成消息填充之后，再将原始数据的长度补到已经进行了填充操作的消息后面，这部分占 64Bit，而且是采用大端存储，这就保证了完成填充和长度附加之后的消息长度是 512Bit 的倍数。

2.3.2 初始化向量

SHA256 使用一个长度为 8 的 32Bit 初始化 Hash 值向量，和一个长度为 64 的 32Bit 常量数组:

uint32_t hv_primes[8]= { ... };
const uint32_t kv_primes[64]= { ... };

2.3.3 循环运算

这里是 SHA2 算法的核心，将填充后的消息分成每块 512Bit 的 M 个数据块，对每个数据块执行以下的操作。

stpe1：首先生成 64 个运算字单元 uint32_t words[64]; 其中前 16 个字 words[0-15] 直接使用当前数据块的 16 个字（512Bits/32=16words），后面的 48 个字 words[16-63] 采用以下代码迭代生成，其中 RTSHIFT(x,n) 表示对 x 右移 n 位，RTROT(a,x,n) 表示对 a 字节的数据 x 循环右移动 n 位。

```
for (int i=16; i<64; ++i) {
    s0= RTROT(4,words[i-15],7) ^ \
        RTROT(4,words[i-15],18) ^ \
        RTSHIFT(words[i-15],3);
    s1= RTROT(4,words[i-2],17) ^ \
        RTROT(4,words[i-2],19) ^ \
        RTSHIFT(words[i-2],10);
    words[i]=words[i-16]+s0+words[i-7]+s1;
}
```

step2: 然后定义 $A...H$ 为初始化的 Hash 向量 $h[0...7]$，执行 64 轮的加密循环。每次循环的操作如图4所示。具体的代码实现可以参考 SHA2.c 的 int sha256(unsigned char *message) 函数。

step3：执行完 64 轮循环之后需要更新 Hash 向量，分别将 $h[0...7]$ 加上 a, b, c, d, e, f, g, h 即可。

2.3.4 输出最后的摘要信息

完成 SHA256 加密后，使用 16 进制依次拼接输出 8 个 Hash 向量，即为最后的 Hash 值。

2.4 Hash 函数 SHA3-256, SHA3-512 实现

SHA-3 (Secure Hash Algorithm 3) 是美国国家标准技术研究所 (NIST) 于 2015 年发布的一种密码杂凑算法。它接受任意长度的消息作为输入，输

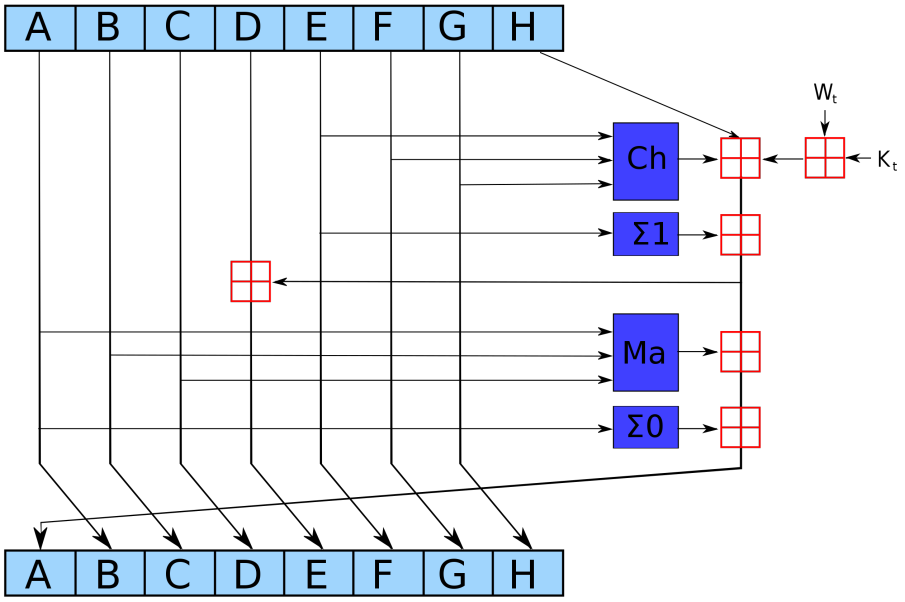

图 4: SHA256 第 t 次加密循环

出一个固定长度的消息摘要。SHA2 目前使用良好，SHA3 对 SHA2 更多是一种补充和备用，现阶段并不是取代作用。SHA3-256 和 SHA3-512 算法流程一样，仅仅是处理数据大小和输出摘要大小不同，下面以 SHA3-256 为例介绍其实现。具体实现可以参考 SHA3.c 中的 void sha3_256(unsigned char* message, int message_len) 函数

2.4.1 消息填充

在消息后加入 1 、最少个数的 0 和 1 （即加入：100...001 ），使填充后的消息长度是 512 的整数倍。然后将消息分成 n 段，每段的长度为 512Bit。

2.4.2 数据处理流程

对每个数据块执行 24 轮操作，参照 SHA3 标准文档 3.2 小节，每轮操作包括 5 个步骤：theta()、rho()、pi()、chi()、iota(round)，分别实现了这几个函数。

void theta() 包括以下三个子步骤：(1) 对于每一列，计算其对应的 C

值：将该列的 5 个字节异或起来；(2) 对于每一列，计算其对应的 D 值：将上一列的 C 值和下一列的 C 值进行异或运算，再将结果向左循环移位 1 个字节；(3) 对于每个字节，将其与对应的 D 值进行异或运算，更新 A 数组中的值。

void rho() 将 A 数组中的每个元素向左循环移位，实现通过对 A 状态矩阵中的某些元素进行旋转操作，从而打破对称性

void pi() 将每行元素向左循环移位，使得第 i 行的元素移动了 i 个位置，同时将第一列元素移动到了最后一列。

void chi() 以 5x5 的状态矩阵 A 为输入，将其转换为一个新的矩阵 B。

在 chi() 函数中，对于每个状态矩阵中的行，对该行的每个元素进行操作。对于每个元素，chi() 会将其值设置为该元素和该元素的相邻元素的异或结果，其中相邻元素的定义如下：（1）第一个相邻元素：该元素同一行的下一个元素（2）第二个相邻元素：该元素同一行的前一个元素（3）第三个相邻元素：该元素在下一行、同一列的元素这样，每个元素的值都会受到其自身和其周围元素的影响，从而实现数据的混淆。最终，chi() 函数输出一个新的状态矩阵 B。

int rc(int t) 对一个固定的常量进行异或操作，常量的值不同取决于算法的轮数。

最后将容器中剩余的数据输出为 16 进制的字符串即为最后的摘要

2.5 算法正确性及性能测试

2.5.1 测试流程

首先执行 generate_test_files.py 生成大小分别为 16K 和 4M 的待测试文件对于分组加密算法 AES 和 SM4，我们采用如下步骤进行正确性测试：

1) 使用 gcc 生成可执行程序

2) 使用 python 的 subprocess 调用生成的可执行程序，并将密钥、待加密文件作为参数传入，生成加密后的文件

3) 使用 python 调用标准库函数对待加密文件进行加密。加密时每次读取待加密文件的 16 个字节，并使用标准库加密获取加密结果，然后同步读取可执行程序生成的加密文件中 16 字节，将二者进行对比，直到对待加密文件中的所有内容完成加密。

4) 如果中间某块加密内容不同则说明可执行文件加密错误，打印错误并返回；否则说明整个文件加密正确，算法实现正确性测试成功

对于 SHA2 和 SHA3 测试与对分组加密算法测试类似，具体步骤如下：

1) 使用 gcc 生成可执行程序

2) 使用 python 的 subprocess 调用生成的可执行程序，并将待处理文件作为参数传入，生成消息摘要

3) 使用 python 调用标准库函数对待处理文件进行 Hash 值计算。将可执行程序生成的摘要与标准库生成的进行对比，查看算法实现是否正确

同时，在分组加密、计算 Hash 值开始的前后，分别使用 clock() 函数计算开始和结束的时间点，并以秒为单位打印处理所需时间，以此来测试我们算法实现的性能。

2.5.2 测试结果

对于 AES 算法，我们分别测试密钥长度为 128、192 和 256 位，数据分组长度为 128 位情况下对大小为 16K，4M 的文件加密的正确性和性能

对 SM4 算法，我们测试密钥长度 128 位，数据分组长度 128 位情况下对大小为 16K，4M 的文件加密的正确性和性能

对 SHA256、SHA512、SHA3-256 和 SHA3-512 对大小为 16K，4M 的文件计算 Hash 值的正确性和性能

经过测试，算法对两个文档加密结果均与标准库一致，正确性测试成功，当然如果想要测试小长度的测试用例，也可以新建一个文件并写入待处理的数据，然后在 test.py 中执行想要测试的算法，此处因为已经对 16K 和 4M 的数据进行了测试，正确性得到了保证，所以不再赘述。test.py 程序执行过程中会输出几个加密算法执行的时间，性能测试的实验结果如表3所示。

表 3: 性能测试实验结果

	16K 数据加密时间 (s)	4M 数据加密时间 (s)
AES-128	0.053886	9.436504
AES-192	0.046349	11.447040
AES-256	0.053694	13.500836
SM4	0.001878	0.150440
SHA256	0.000425	0.053942
SHA512	0.000234	0.027403
SHA3-256	0.026404	0.047198
SHA3-512	6.342261	11.991306

2.6 代码文档

所有代码都存放在在 task2 目录下,具体操作可以参考目录中的 README.md 文件, 其中

* **AES 算法实现**：AES.c, AES.h, aes_main.c
* **SM4 算法实现**：SM4.c, SM4.h, sm4_main.c
* **SHA256，SHA512 算法实现**：SHA2.c
* **SHA3-256，SHA3-512 算法实现**：SHA3.c
* **正确性和性能测试脚本**：Makefile, test.py , generage_test_file.py

3 SM3 口令破解

3.1 SM3 算法实现

SM3 密码杂凑算法是中国国家密码管理局 2010 年公布的中国商用密码杂凑算法标准。具体算法标准原始文本参见参考文献 [1]。该算法于 2012 年发布为密码行业标准 (GM/T 0004-2012), 2016 年发布为国家密码杂凑算法标准 (GB/T 32905-2016)。

SM3 适用于商用密码应用中的数字签名和验证，是在 SHA-256 基础上改进实现的一种算法，其安全性和 SHA-256 相当。SM3 和 MD5 的迭代过

程类似，也采用 Merkle-Damgard 结构。消息分组长度为 512 位，摘要值长度为 256 位。

整个算法的执行过程可以概括成四个步骤：消息填充、消息扩展、迭代压缩、输出结果。

本次作业采用 python 来实现这四个步骤，首先使用 str2bin() 将字符串转化为比特串，然后调用 msgfill() 模块进行填充和扩展，随后用 iterationfunction() 模块对填充后的消息进行迭代压缩，最后输出相应的结果，其示例如下图所示。

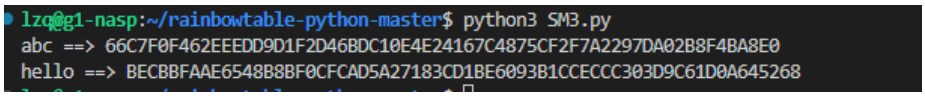

图 5: SM3 实现样例

3.2 彩虹表实现

对于彩虹表实现，关键是设计其 R 函数。在本次设计作业中，R 函数的具体流程为先对哈希值作字节转换处理，然后根据字节值、字节长度、哈希链长、密码长度等数值建立哈希值到字符集的映射关系，最终将哈希值映射到明文空间。哈希方法和密码函数定义在 RainbowTable 类的 hashV() 方法和 reduce() 方法中。

题目要求的是破解 5 位数和 6 位数的字母＋数字组合的密码（不区分字母大小写），本小节分别针对 5 位密码和 6 位密码生成彩虹表。对于 5 位密码，随机生成 100000 个密码，然后进行链长为 1000 的彩虹链生成；对于 6 位米麦，随机生成 200000 个密码，然后进行链长为 2100 的彩虹链生成；生成结果分别到 rain5.txt 和 rain6.txt 中。用 48 线程并行计算 10h，最终分别得到 7M 和 181M 的彩虹表文件。此外，我们还生成了覆盖 5 位密码全明文空间的完整彩虹表（链长为 5），文件大小为 4G，完整彩虹表链接如下：清华大学云盘下载链接。

3.3 破解结果

随机采样 100 个 5 位数密码和 100 个 6 位数密码进行破解，破解成功率为 0.83 和 0.67。6 位数密码破解率较低，可能是因为相比于 5 位数密码

及其彩虹表而言，6 位数彩虹表文件并没有随着明文空间等比例扩大，需要更大的彩虹表文件。

3.4 代码文档

全部代码见 rainbowtable-python-master 中，主程序在 rainbowGenerator.py 中，SM3 算法生成程序在 SM3.py 中。

参考文献

[1] M. Rejewski, “An application of the theory of permutations in breaking the enigma cipher,” *Applicationes mathematicae*, vol. 16, no. 4, pp. 543–559, 1980.

[2] M. J. Dworkin, E. B. Barker, J. R. Nechvatal, J. Foti, L. E. Bassham, E. Roback, and J. F. Dray Jr, “Advanced encryption standard (aes),” 2001.

算法 1: Rejewski 的破解方法

输入: 前 6 个密文字母的对应表格 T

输出: 解集 *Solutions*

```
1  预计算阶段
2  begin
       // 由波兰 Bomba 机模拟的 Enigma 机
3      E ← Enigma ();
       // 存储的数据库，用于之后查表
4      Database ← Map ();
5      foreach S ∈ PossibleSettings do
6          E.setPosition (S);
7          C ← FindChainsByEnigma (E);
8          Database.store (S,C);
9      end
10 end
11
12 破解过程
13 begin
14     Solutions ← Set ();
15     foreach S ∈ PossibleSettings do
16         Matched ← true;
17         for i ← 1 to 3 do
18             Chain ← FindChainsInTableRow (T[i],T[i+3]);
19             Expected ← Database.get (S+i);
20             if Chain ≠ Expected then
                   // 字母链没有通过检验
21                 Matched ← false;
22             end
23         end
24         if Matched = true then
               // 当前初始状态对应的字母链通过了检验
               // 将可能的解加入解集
25             Solutions.add (S);
26         end
27     end
28 end
```

序号	1	2	3	4	5	6	7	8	9
明文	A	S	T	W	B	K	D	J	X
密文	S	T	W	T	J	D	J	X	S

序号	10	11	12	13	14	15	16	17
明文	S	A	X	Q	F	C	J	T
密文	A	Q	B	J	Q	J	O	D

表 2: 明文与密文的对应表格

算法 2: Turing 的破解方法

输入: 明文与密文字母的对应表格 T

输出: 解集 *Solutions*

```
1  begin
       // 由英国 Bombe 机模拟的 Enigma 机
2      E ← Enigma ();
3      Solutions ← Set ();
4      Chains ← FindChainsInTable (T);
5      foreach S ∈ PossibleSettings do
6          E.setPosition (S);
7          Matched ← true;
8          foreach C ∈ Chains do
               // 检查是否满足字母链，并得到插线板规则的约束条件
9              Matched, Constraint ← ExamineChains (E,C);
10         end
11         if Matched = true then
               // 当前初始状态对应的字母链通过了检验
12             if SolveConstraint (Constraint) = true then
                   // 并且插线板规则没有冲突
13                 Solutions.add (S);
14             end
15         end
16     end
17 end
```