《现代密码学》实验报告

实验名称：SHA-256的实现	实验时间：2024年12月9日
学生姓名：黄集瑞	学号：22336090
学生班级：22保密管理	

一、实验目的

> 通过实现数据的 SHA-256 加密过程，可以帮助我们理解哈希函数的工作原理，包括消息预处理、消息分块、初始化向量设置、压缩函数的迭代运算等具体步骤；掌握 SHA-256 算法中涉及的逻辑操作（如位移、置换、模加运算）同时，通过该实验，有利于提高我们对复杂算法的分析与实现能力，提升编程实践能力和问题解决能力，为后续研究更高级的密码学算法奠定基础。

二、实验内容

> - 用C++实现SHA-256的加密算法
> - 输入如下：
>
>   二进制流输入需要加密的消息
> - 输出如下：
>
>   二进制流输出通过SHA-256算法加密后的密文（**注意要按照大端序进行输出**）

三、实验原理

> 我认为对于本次实验而言，如何处理好对消息的读入最为关键，因为其关系着我们如何进行每一轮`chunk`的处理，以及哈希值的计算；其次，我们需要理解SHA-256对不同长度的消息是如何进行预处理的；最后，就是实现SHA-256的主体逻辑。
>
> - 输入以及输出逻辑：
>
>   对于本次实验而言，因为不同长度消息的输入会对应着不同的消息预处理以及`chunk`的初始化，所以输入部分需要特别处理；并且同上次实验，样例数据的内存仍会超过题目所限制的运行内存，所以我们**不能直接读入所有数据**，而需要采取**分组读取**的方法。而对于输出，我们只需要注意，最后要将其修改为**大端序**即可。
> - 消息预处理部分：
>
>   对于消息预处理而言，不管输入的消息多大，我们都需要先补充一个`0x80`字节，然后按照要求进行补充`0x00`字节，最后需要剩余64位也即8个字节的位置来填充消息本身的长度。（由于我们仅仅是留下8个字节的长度来保存长度，所以消息长度一般是不能超过2^{64}）
> - SHA-256的主体逻辑：
>
>   SHA-256算法的主体逻辑无非就是对于固定长度的盘块通过一个`map`映射函数从而得到更新后的hash值，只需要记住每处理一个盘块就会更新一次哈希值；而每一次盘块处理都会有64次循环，这样总体逻辑就成功实现了。

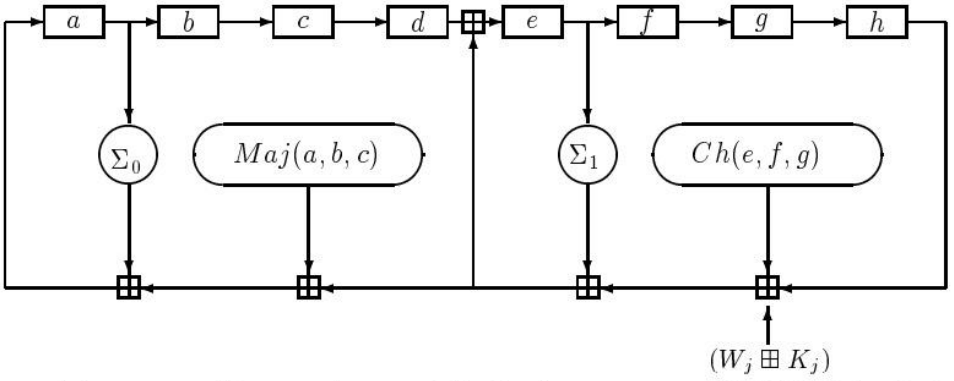


Figure 1: j^{th} internal step of the SHA-256 compression function. C

四、实验步骤

- 输入以及输出逻辑

  本次的输入为需要加密的消息的二进制数据，由于算法的主体循环都是通过对512位的`chunk`进行处理，所以我们便按照一个盘块大小来进行读取数据，具体代码如下：

```
// 读入数据，由于数据过大会占用内存，所以切合实际情况，每次读入512位数据，最后再进行填充
uint8_t data[64]={0};
bool flag = true;
uint8_t bytesRead = 0;
while(flag)
{
    in.read((char *)data, 64);
    // 读取的是字节长度
    bytesRead = in.gcount();
    // 说明读取的数据不足512位，需要进行填充（也意味着我们的读取已经结束）
    if(bytesRead < 64)
    {
        len += bytesRead;
        flag = false;
        sha256_final(data, bytesRead, len); // 修改后
    }
    len += bytesRead;
    sha256(data);
}
```

  可以看到我初始化了一个`uint8_t data[64]`的数组来作为读取的`chunk`，然后每次都读取512位的数据，若读取的数据小于512位则会进行额外的处理。其中，我采用了`gcount()`的函数来读取输入数据的字节长度，以便于让其参加后面的运算。

  而另外一个需要注意的地方就是本次的输出需要实现**大端序**的输出，所以这里我编写了一个将二进制转为大端序的函数，具体代码如下：

```
// 将小端字节序转换为大端字节序
uint32_t toBigEndian(uint32_t value) {
    return ((value & 0x000000FF) << 24) |
           ((value & 0x0000FF00) << 8)  |
           ((value & 0x00FF0000) >> 8)  |
           ((value & 0xFF000000) >> 24);
}
```

该代码将位于后端的数据移到了最前面，而将最前面的数据转移到了最后面。最后，将最终的hash值进行输出即可。

- 消息预处理

对于我的代码而言，由于每次都是直接读入512位的数据，那么当读入的数据小于512位的时候就表明此时的消息已经是最后一部分了。而最后一部分的消息便需要附加填充比特并且再附加64位的长度值，得到最终的盘块，而我们进行添加附加填充比特的公式是：$l+1+k\equiv 448 \mod 512$，剩下的64位正好就是可以填充我们的长度消息，但是我们需要注意填充仍有两种不同的情况：

- 当读取的块长度本身小于448时，我们通过以上公式可以附加填充消息：首先，先在消息后面添加一个`0x80`，然后算出要填充`k`个`0x00`，最后补充上长度消息即可。
- 当读取的块长度本身大于等于448的时候，当我们添加上`0x80`时便没有足够的空间来添加此时的消息长度。为此，我们的解决方法是先照常补充上`0x80`然后在后面的字节中一直补充`0x00`，这样可以先构成一个512的盘块并进行一次哈希值的运算；最后，我们将新的盘块全部填充`0x00`，只在最后的64位中填充长度消息，然后再进行最终的hash值运算即可。（这样我们将一个盘块分为了两个盘块来计算）具体代码如下：

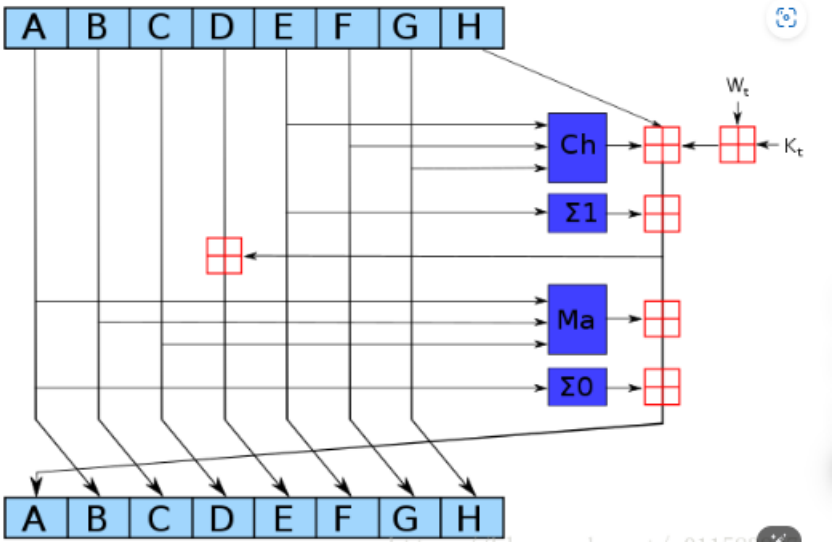
```
// 这个函数专门用于处理最后一个数据块
void sha256_final(uint8_t *data, uint8_t bytesRead, uint64_t len)
{
    // 进行填充
    if (bytesRead < 56)
    {
        data[bytesRead] = 0x80;
        for (int i = bytesRead + 1; i < 56; i++)
        {
            data[i] = 0x00;
        }
    }
    else // 此时同一个盘块无法再存储数据的长度，所以需要进行特殊的处理
    {
        data[bytesRead] = 0x80;
        for (int i = bytesRead + 1; i < 64; i++)
        {
            data[i] = 0x00;
        }
        // 以上的填充构成一个盘块
        // 先处理一遍哈希值
        sha256(data);
        memset(data, 0, 56); // 清空数据，仅剩下最后的8个字节来保存长度
    }
    // 保存数据长度
    uint64_t length = len * 8;
    for(int i = 0; i < 8; i++)
    {
```

```
            data[56 + i] = (length >> (56 - i * 8)) & 0xff;
        }
        // 处理最后一个盘块
        sha256(data);
    }
```

这里我写了一个`sha256_final`函数来进行最后一个盘块的判断，其中若其字节长度`bytesRead`小于56（即小于448位）那么就按照第一种情况正常填充；否则，就按照第二种情况进行两次盘块的处理即可，这样就解决了长度扩展的问题。

- SHA-256的主体逻辑

  对于SHA-256的主体逻辑，网上有非常多参考资料，大体的过程为：



对于n个盘块，我们需要对hash值进行n次处理，每一次处理都是使用了一个映射`map`函数，其内容如下：$Map(H_{i-1}) = H_i$，而映射函数其中包含了64次加密循环，每次循环的过程都如上图所示，处理完　之后便更新了一次hash值。而对于里面的参数W_t共有64个，是由输入的盘块构造出来的，构造过程如 下图所示：

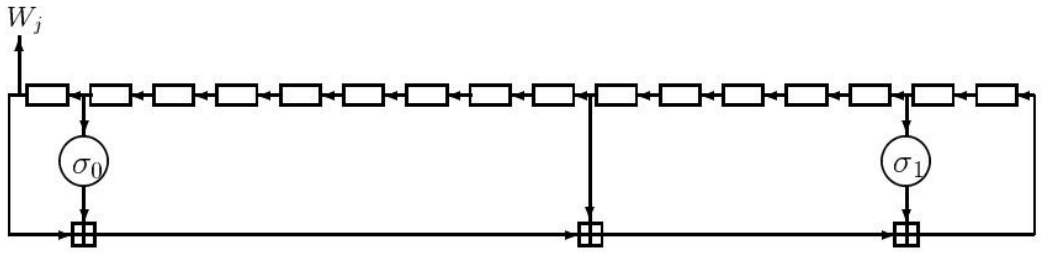


Figure 2: SHA-256 message schedule　知乎 @Datacruiser

而对于K_t则是对应的初始化常量，通过这些步骤，sha-256的主体逻辑就完成了。具体代码如下：

```
// 调用该中间过程，而该过程一定是对512位数据进行处理，所以不需要记录长度变量
void sha256(uint8_t *data)
{
    uint32_t W[64] = {0}; // 用于存储扩展后的数据
    uint32_t a, b, c, d, e, f, g, h; // 用于存储中间变量
    uint32_t T1, T2; // 用于存储临时变量
    // 初始化中间变量
    a = H[0];
    b = H[1];
    c = H[2];
```

```
    d = H[3];
    e = H[4];
    f = H[5];
    g = H[6];
    h = H[7];
    // 对数据进行扩展
    for(int i = 0; i < 16; i++)
    {
        W[i] = (data[i * 4] << 24) | (data[i * 4 + 1] << 16) | (data[i * 4 + 2] << 8) | data[i * 4 + 3];
    }
    for(int i = 16; i < 64; i++)
    {
        W[i] = SIG1(W[i - 2]) + W[i - 7] + SIG0(W[i - 15]) + W[i - 16];
    }
    // 进行压缩
    for(int i = 0; i < 64; i++)
    {
        T1 = h + EP1(e) + Ch(e, f, g) + K[i] + W[i];
        T2 = EP0(a) + Ma(a, b, c);
        h = g;
        g = f;
        f = e;
        e = d + T1;
        d = c;
        c = b;
        b = a;
        a = T1 + T2;
    }
    // 更新哈希值
    H[0] += a;
    H[1] += b;
    H[2] += c;
    H[3] += d;
    H[4] += e;
    H[5] += f;
    H[6] += g;
    H[7] += h;
}
```

其中的主体逻辑就是实现了上述所阐述的内容，而由于该操作一定是对512位的盘块进行操作，所以不需 要判断长度之类的，可以直接完成这个主体逻辑。

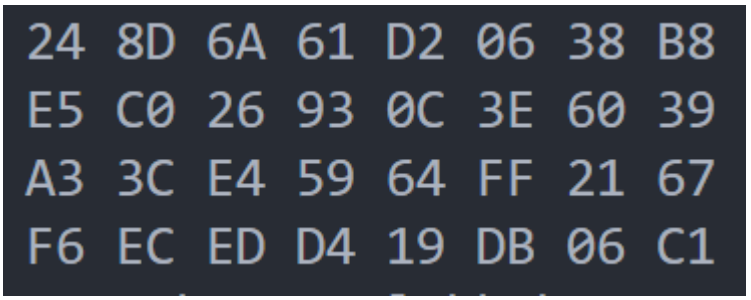
至此SHA-256算法就完成实现了。

五、实验结果

输入如下所示：

```
61 62 63 64 62 63 64 65 63 64 65 66 64 65 66 67
65 66 67 68 66 67 68 69 67 68 69 6A 68 69 6A 6B
69 6A 6B 6C 6A 6B 6C 6D 6B 6C 6D 6E 6C 6D 6E 6F
6D 6E 6F 70 6E 6F 70 71
```

输出为：



验证过后，该答案是正确的。

六、实验提升

在完成了以上逻辑之后，通过与其他同学的沟通，发现大家都有实现缓冲区的功能，而对于sha-256的实现也都是对缓冲区进行操作；并且当有了缓冲区之后，便可以一次读入更多的数据，所以代码的输入逻辑后面提升为如下所示：

```
    uint8_t buffer[1024*24] = {0}; // 大缓冲区，一次可以读入更多字节
    uint8_t chunk[64] = {0};     // 小缓冲区，用于存储每次处理的 512 位（64 字节）数据
    size_t bytesRead = 0;
    size_t bufferIndex = 0;
    uint64_t total_len = 0; // 用于存储总读取的字节长度
    bool flag = true;

    while (flag)
    {
        // 如果缓冲区已经全部处理完，则重新读取数据到缓冲区
        if (bufferIndex == bytesRead)
        {
            in.read((char *)buffer, sizeof(buffer));
            bytesRead = in.gcount();
            bufferIndex = 0;
            if(bytesRead == 0)
            {
                flag = false;
                sha256_final(chunk, 0, 0); // 调用 SHA-256 尾部处理函数
            }
            // 如果读到的数据不足 1024 字节，说明文件已经读到尾部
            if (bytesRead < sizeof(buffer))
            {
                flag = false;
            }
        }

        // 从缓冲区提取 64 字节数据块到 chunk
        while (bufferIndex + 64 <= bytesRead)
        {
            std::memcpy(chunk, buffer + bufferIndex, 64);
            sha256(chunk); // 对每块 64 字节的数据调用 SHA-256 处理函数
            bufferIndex += 64;
            total_len += 64;
        }

        // 处理文件尾部不足 64 字节的数据
        if (!flag && bufferIndex < bytesRead)
```

```
        {
            size_t remaining = bytesRead - bufferIndex;
            std::memcpy(chunk, buffer + bufferIndex, remaining);
            total_len += remaining;
            sha256_final(chunk, remaining, total_len); // 调用 SHA-256 尾部处
理函数
        }
    }
```

通过以上的修改，一次可以读入更多的数据，并且时间效率也有提升。但是，效率的提升也不是随着大缓冲区大小的增加而跟着增加的。经过尝试之后，发现此时的缓冲区选取能够得到最好的提升效果，至于原因就不得而知了。

七、实验总结

通过本次实验，我不仅掌握了 SHA-256 算法的实现细节，还深刻理解了其背后的设计思想。实验进一步强化了我对课堂知识的理解，也为我后续学习更高级的哈希算法（如 SHA-3）、对称加密与非对称加密等奠定了坚实的基础。同时，在未来的算法实现中，我也会再接再厉不断提升自己的编程能力。

八、思考题

1. SHA-1 的初始常数是很有规律的 `0x67452301, 0xEFCDAB89, 0x98BADCFE, 0x10325476, 0xC3D2E1F0`，SHA-256 的初始常数和轮常数分别来自于前几个素数的平方根/立方根的小数部分。为什么要这么设计？

   答：SHA-256 的初始常数和轮常数的设计，旨在：

   1. 避免人为干预和潜在后门，增强算法的公正性和安全性；
   2. 增强随机性和扩散性，确保输出的高度敏感性；
   3. 提高对差分攻击和线性攻击的抵抗能力；
   4. 保持算法的简洁性和可验证性。

   相比 SHA-1 的简单模式化常数，SHA-256 的设计更先进，进一步提升了安全性，确保其在当前安全需求下的可靠性。

2. 有一类 hash 算法被称为“Non-cryptographic hash function”，例如 Java (OpenJDK) 中 `hashCode` 的实现是非常简单的迭代计算 `h = 31 * h + x`，其他的此类算法包括 FNV、MurmurHash、xxHash 等。它们与 MD5、SHA-1、SHA-256 等 hash 算法的区别是什么？它们有哪些主要用途？为什么会比 SHA-256 等等更适合用于这些场景？

   答：

   Non-cryptographic hash functions

   - 目标是 **快速性** 和 **分布均匀性**。
   - 它们不需要具备密码学安全性（如抗碰撞性或不可逆性），而是关注如何快速生成具有良好分布特性的哈希值。

   Cryptographic hash functions

   - 目标是 **安全性**。
   - 必须满足抗碰撞性（难以找到两个不同的输入产生相同的输出）、抗篡改性（输入轻微变化会导致输出剧烈变化）、以及抗逆向性（从输出推导输入几乎不可能）。

由于以上hash算法的目的不同，所以**Non-cryptographic hash functions**更加适用于一些不太敏感的领域，比如数据检索、错误检测、数据库索引和文件或数据比较。因为不需要保证安全性，所以避免了那些复杂的加密操作，它们的工作速度通常比后者更快，并且也更加容易实现，而且由于其一般生成较短的hash值，它们的存储需求也更加的低。

3. 如果你需要设计一个用户系统，你也许知道数据库中不应该以明文存储用户的密码（遗憾的是，我国的知名计算机技术社区 CSDN 就犯过这样的错误），而应该存储密码的 hash。但是，在这种情况下直接使用 SHA-256 等等仍然是不推荐的。有哪些 hash 算法更适合用来处理密码？它们与 SHA-256 等等有什么区别？

   答：

   - **bcrypt**：bcrypt是一种设计的非常成熟的密码哈希函数，它内置了盐值和密钥拉伸功能。bcrypt算法在每次哈希时都会使用一个成本因子（work factor），这个因子可以调整以增加哈希的计算时间，从而提高安全性。
   - **PBKDF2（Password-Based Key Derivation Function 2）**：PBKDF2是一个密码基密钥派生函数，它使用一个密码和一个盐值，通过重复应用一个加密哈希函数（如SHA-1或SHA-256）来派生密钥。PBKDF2允许设置迭代次数，从而增加破解的难度。
   - **Argon2**：Argon2是密码哈希竞赛（Password Hashing Competition）的获胜者，它结合了内存哈希函数的特性，使得攻击者难以使用GPU进行暴力破解。Argon2有三种变体，分别针对不同的应用场景。其高度灵活，支持调整**内存使用**、**计算时间**和**并行度**，对抗硬件加速攻击效果较好。
   - **scrypt**：scrypt是一个内存密集型的密码哈希函数，它旨在使暴力破解攻击变得昂贵和耗时。scrypt允许设置内存和CPU的成本参数，以适应不同的安全需求。

     与SHA-256等通用哈希函数相比，这些密码哈希函数的主要区别在于它们提供了额外的安全特性，如盐值和密钥拉伸，这些特性通过增加计算复杂度、内存需求以及盐值使得它们更适合用于密码存储。SHA-256等算法虽然安全，但它们计算速度快，没有内置盐值和密钥拉伸功能，因此更容易受到暴力破解攻击。