山东大学 软件 学院

**密码学原理与实践** 课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：202200201095 | 姓名：杨伟康 | | 班级：22网安 |
| 实验编号：实验一 | | | |
| 实验题目：对称加密算法实验 | | | |
| 实验学时：8 | | 实验日期： 10.19 | |
| 实验目的与内容：  **目的**：理解开源密码库实现的基本架构，熟悉对称算法的加解密函数封装与调用，并能能够利用开源设计接口进行二次封装，并实现一个界面友好，功能正确的采用对称算法的文件加解密工具。锻炼接口分析能力和工具设计的工程能力。  **内容**：下载OPENSSL开源软件源码，编译链接形成相应操作系统下的函数库，通过阅读网站和软件说明，至少了解其中DES和AES算法的函数接口，在原始加密函数（ECB\CBC\CFB\OFB）基础上进行封装，实现一组能直接加密解密任意长度消息的加解密函数接口，并在此基础上实现一个采用口令密钥演化方法针对任意文件口令加密加密和解密命令行程序。该程序可以对操作系统中的任意大小和类型的文件用口令加密得到密文文件，并能够通过口令解密得到明文文件。 | | | |
| 硬件环境：  *机型，CPU，内存等主要参数。*  *若用虚拟机，则宿主机与虚拟机参数分别列出。*  处理器 11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz 2.69 GHz  机带 RAM 16.0 GB (15.9 GB 可用)  设备 ID 9DCDB7C5-EBBA-4C1F-845F-94B127DFAE09  产品 ID 00342-30515-76480-AAOEM  系统类型 64 位操作系统, 基于 x64 的处理器  笔和触控 没有可用于此显示器的笔或触控输入 | | | |
| 软件开发环境：  *若用虚拟机，请写明虚拟机软件及版本，以及宿主机操作系统版本。*  Visual studio2022+OPENSSL3.3.2 | | | |
| 实验步骤与内容： 实验的思路  1. **环境准备**：下载OPENSSL源码，在操作系统上编译安装。 2. **算法学习**：阅读密码学的实验指导书，了解OPENSSL中DES和AES算法的接口和使用方法。 3. **函数封装**：写一段代码，做到在ECB、CBC、CFB、OFB等模式下封装加密解密函数，处理填充问题以支持任意长度消息。 4. **口令密钥演化**：实现基于口令的密钥生成方法，用于加密和解密。 5. **程序实现**：编写命令行程序，实现文件加密和解密功能。 6. **测试与调试**：对工具进行多种方式的测试，记录调试过程，分析实验结果  主要数据结构的分析说明 **主要数据结构分析说明**  在实现基于OPENSSL开源软件的文件加解密工具过程中，我们需要对主要的数据结构进行分析和理解，以便更好地封装和使用对称加密算法（如DES和AES）的函数接口。以下是几个关键数据结构的分析说明：  **1. 密钥管理相关数据结构**   * **EVP\_KEY**（抽象表示，OPENSSL内部使用）   + **描述**：虽然OPENSSL没有直接提供一个名为EVP\_KEY的结构体，但EVP（高级加密标准）接口用于管理密钥，通常通过EVP\_PKEY结构体来实现密钥的存储和管理。   + **用途**：存储对称或非对称密钥的元数据，包括密钥类型、长度、算法等。   + **相关函数**：EVP\_PKEY\_new(), EVP\_PKEY\_free(), EVP\_PKEY\_assign\_EVP\_PKEY(), 等。 * **对称密钥**（例如AES密钥）   + **描述**：在AES等对称加密算法中，密钥通常以字节数组的形式存储。   + **用途**：用于加密和解密过程中的密钥数据。   + **管理**：通过EVP\_CIPHER\_CTX上下文结构管理。   **2. 加密算法上下文数据结构**   * **EVP\_CIPHER\_CTX**   + **描述**：这是一个用于存储对称加密算法上下文的结构体，包含了加密或解密操作所需的全部状态信息。   + **用途**：初始化、配置、执行加密或解密操作。   + **相关函数**：EVP\_CIPHER\_CTX\_new(), EVP\_CIPHER\_CTX\_free(), EVP\_EncryptInit\_ex(), EVP\_DecryptInit\_ex(), EVP\_CipherUpdate(), EVP\_CipherFinal\_ex(), 等。   **3. 消息摘要和填充相关数据结构**   * **EVP\_MD\_CTX**   + **描述**：用于存储消息摘要（如哈希函数）上下文的结构体。   + **用途**：虽然主要用于消息摘要计算，但在某些情况下可能与加密填充模式相关。   + **相关函数**：EVP\_MD\_CTX\_new(), EVP\_MD\_CTX\_free(), EVP\_DigestInit\_ex(), EVP\_DigestUpdate(), EVP\_DigestFinal\_ex(), 等。   **4. 文件操作相关数据结构**   * **FILE**   + **描述**：标准C库中的文件结构体，用于表示打开的文件。   + **用途**：读写文件数据。   + **相关函数**：fopen(), fread(), fwrite(), fclose(), fseek(), ftell(), 等。   **5. 口令密钥演化（Password-Based Key Derivation Function, PBKDF）相关数据结构**   * **EVP\_PBE\_KEYGEN**（抽象表示，通过EVP接口调用）   + **描述**：虽然OPENSSL没有直接提供一个结构体用于PBKDF，但EVP接口提供了PBKDF2等密码演化函数。   + **用途**：从用户提供的口令生成加密密钥。   + **相关函数**：PKCS5\_PBKDF2\_HMAC(), EVP\_BytesToKey()（虽然EVP\_BytesToKey较旧且不推荐使用，但在某些示例代码中仍可见）。   **6. 错误处理相关数据结构**   * **ERR\_STATE**（抽象表示，OPENSSL内部使用）   + **描述**：OPENSSL内部使用的错误处理结构体，用于记录错误信息。   + **用途**：通过ERR\_get\_error(), ERR\_error\_string()等函数获取和打印错误信息。   *主要算法代码的分析说明*  *项目管理文件的说明*  *实验投入的实际学时数* 调试排错过程的记录 第一个任务写出能直接写出加密解密任意长度的消息的接口，按照实验指导书，边进行加解密函数的熟悉边完成就可以。  过程中我想要输出一下密文，尝试直接打印密文（ciphertext）时遇到了问题，因为密文通常包含不可打印的字符（即非ASCII字符）。这些字符在打印到控制台时可能显示为乱码或特殊符号。  于是输出转化为十六进制字符    使密文输出变规整  在写加密程序的时候，在网上搜索口令派生密钥的安全函数，看到pbkdf2可以使用，    在学习了网上关于pbkdf2的代码之后，发现#include<openssl/pbkdf2.h>不能引用，  派生函数也无法使用  在我冥思苦想是不是我的openssl配置错误的时候，我突然想到版本更新3.x之后会不会被弃用了，网上搜索得知，果然新版openssl已经不用了  继续寻找  Openssl的3.x能用的函数，在阅读新发的实验指南引用了openssl/kdf.h  具体看openssl使用指南第四章，于是我的口令派生密钥就写好了    将文件以二进制形式存储再加密，本来用的是BIO即openssl自带的来读入，但是导致老是不能成功读取，或者不能成功写入加密文件，网上资料又是很少关于这个的使用    于是放弃使用了，直接使用c++标准库中的rb和wb来进行二进制读写  修改代码如下   关键代码 口令派生密钥部分  int derive\_key\_from\_password(const char\* password, unsigned char\* derived\_key, size\_t key\_len) {  EVP\_KDF\* kdf = EVP\_KDF\_fetch(NULL, "PBKDF2", NULL);  if (!kdf) return -1;  EVP\_KDF\_CTX\* ctx = EVP\_KDF\_CTX\_new(kdf);  if (!ctx) { EVP\_KDF\_free(kdf); return -1; }  unsigned char salt[] = "NaCl";  unsigned int iterations = 80000;  OSSL\_PARAM params[] = {  OSSL\_PARAM\_construct\_octet\_string("pass", password, strlen(password)),  OSSL\_PARAM\_construct\_octet\_string("salt", salt, strlen(salt)),  OSSL\_PARAM\_construct\_uint("iter", &iterations),  OSSL\_PARAM\_construct\_utf8\_string("digest", "SHA256", 0),  OSSL\_PARAM\_construct\_end()  };  if (EVP\_KDF\_derive(ctx, derived\_key, key\_len, params) != 1) {  EVP\_KDF\_CTX\_free(ctx);  EVP\_KDF\_free(kdf);  return -1;  }  EVP\_KDF\_CTX\_free(ctx);  EVP\_KDF\_free(kdf);  return 0;  }  加密部分  int encrypt\_file(const char\* input\_filename, const char\* output\_filename, unsigned char\* key) {  FILE\* input\_file = fopen(input\_filename, "rb");  if (!input\_file) {  perror("Failed to open input file");  return -1;  }  FILE\* output\_file = fopen(output\_filename, "wb");  if (!output\_file) {  fclose(input\_file);  perror("Failed to open output file");  return -1;  }  unsigned char iv[IV\_SIZE];  if (!RAND\_bytes(iv, IV\_SIZE)) {  fclose(input\_file);  fclose(output\_file);  perror("Failed to generate IV");  return -1;  }  // 写入 IV 到输出文件  if (fwrite(iv, 1, IV\_SIZE, output\_file) != IV\_SIZE) {  fclose(input\_file);  fclose(output\_file);  perror("Failed to write IV to output file");  return -1;  }  EVP\_CIPHER\_CTX\* ctx = EVP\_CIPHER\_CTX\_new();  if (!ctx) {  fclose(input\_file);  fclose(output\_file);  perror("EVP\_CIPHER\_CTX\_new failed");  return -1;  }  if (EVP\_EncryptInit\_ex(ctx, EVP\_aes\_256\_cbc(), NULL, key, iv) != 1) {  EVP\_CIPHER\_CTX\_free(ctx);  fclose(input\_file);  fclose(output\_file);  perror("EVP\_EncryptInit\_ex failed");  return -1;  }  unsigned char in\_buffer[BUFFER\_SIZE];  int bytes\_read;  while ((bytes\_read = fread(in\_buffer, 1, BUFFER\_SIZE, input\_file)) > 0) {  int out\_len;  unsigned char out\_buffer[BUFFER\_SIZE + 16];  if (EVP\_EncryptUpdate(ctx, out\_buffer, &out\_len, in\_buffer, bytes\_read) != 1) {  EVP\_CIPHER\_CTX\_free(ctx);  fclose(input\_file);  fclose(output\_file);  perror("EVP\_EncryptUpdate failed");  return -1;  }  if (fwrite(out\_buffer, 1, out\_len, output\_file) != out\_len) {  EVP\_CIPHER\_CTX\_free(ctx);  fclose(input\_file);  fclose(output\_file);  perror("Failed to write encrypted data to output file");  return -1;  }  }  int final\_len;  unsigned char final\_buffer[16];  if (EVP\_EncryptFinal\_ex(ctx, final\_buffer, &final\_len) != 1) {  EVP\_CIPHER\_CTX\_free(ctx);  fclose(input\_file);  fclose(output\_file);  perror("EVP\_EncryptFinal\_ex failed");  return -1;  }  if (fwrite(final\_buffer, 1, final\_len, output\_file) != final\_len) {  EVP\_CIPHER\_CTX\_free(ctx);  fclose(input\_file);  fclose(output\_file);  perror("Failed to write final block to output file");  return -1;  }  EVP\_CIPHER\_CTX\_free(ctx);  fclose(input\_file);  fclose(output\_file);  return 0;  }  解密部分  int decrypt\_file(const char\* input\_filename, const char\* output\_filename, unsigned char\* key) {  FILE\* input\_file = fopen(input\_filename, "rb");  if (!input\_file) {  perror("Failed to open input file");  return -1;  }  FILE\* output\_file = fopen(output\_filename, "wb");  if (!output\_file) {  fclose(input\_file);  perror("Failed to open output file");  return -1;  }  unsigned char iv[IV\_SIZE];  if (fread(iv, 1, IV\_SIZE, input\_file) != IV\_SIZE) {  fclose(input\_file);  fclose(output\_file);  perror("Failed to read IV from input file");  return -1;  }  EVP\_CIPHER\_CTX\* ctx = EVP\_CIPHER\_CTX\_new();  if (!ctx) {  fclose(input\_file);  fclose(output\_file);  perror("EVP\_CIPHER\_CTX\_new failed");  return -1;  }  if (EVP\_DecryptInit\_ex(ctx, EVP\_aes\_256\_cbc(), NULL, key, iv) != 1) {  EVP\_CIPHER\_CTX\_free(ctx);  fclose(input\_file);  fclose(output\_file);  perror("EVP\_DecryptInit\_ex failed");  return -1;  }  unsigned char in\_buffer[BUFFER\_SIZE + 16]; // +16 是为了处理可能的填充  int bytes\_read;  while ((bytes\_read = fread(in\_buffer, 1, BUFFER\_SIZE, input\_file)) > 0) {  int out\_len;  unsigned char out\_buffer[BUFFER\_SIZE];  if (EVP\_DecryptUpdate(ctx, out\_buffer, &out\_len, in\_buffer, bytes\_read) != 1) {  EVP\_CIPHER\_CTX\_free(ctx);  fclose(input\_file);  fclose(output\_file);  perror("EVP\_DecryptUpdate failed");  return -1;  }  if (fwrite(out\_buffer, 1, out\_len, output\_file) != out\_len) {  EVP\_CIPHER\_CTX\_free(ctx);  fclose(input\_file);  fclose(output\_file);  perror("Failed to write decrypted data to output file");  return -1;  }  }  int final\_len;  unsigned char final\_buffer[BUFFER\_SIZE];  if (EVP\_DecryptFinal\_ex(ctx, final\_buffer, &final\_len) != 1) {  EVP\_CIPHER\_CTX\_free(ctx);  fclose(input\_file);  fclose(output\_file);  perror("EVP\_DecryptFinal\_ex failed");  return -1;  }  if (fwrite(final\_buffer, 1, final\_len, output\_file) != final\_len) {  EVP\_CIPHER\_CTX\_free(ctx);  fclose(input\_file);  fclose(output\_file);  perror("Failed to write final block to output file");  return -1;  }  EVP\_CIPHER\_CTX\_free(ctx);  fclose(input\_file);  fclose(output\_file);  return 0;  } 运行结果(文本格式，必要时抓屏)1.加密成不同类型文件 首先准备一个文件input.txt,内容为    在命令行调用加密程序，按照输入规则依次输入  Usage: %s <password> <input file> <output file>    加密成功  出现加密文件  无法打开或打开说乱码    此时再删除原文件  我们通过解密来再次得到原文件    解密成功    查看原文件  恢复成功   多种方式测试结果的记录 加密成enc格式的加密文件    删除原文件  再次解密    打开后可以看到原文件没有因为加密文件的类型变化而损坏   2.加密其他类型文件 创建一个ppt类型文件，作为例子  内部随意输入作为标记    进行加密    可以看到虽然加密文件的名称以及文件类型不同，但加密文件大小一致    删除原文件然后解密    得到pptx类型文件，我们查看内部    发现内部保存完好  其他类型的文件也是同理，这里不再重复操作 实验的综合分析关键代码分析 口令派生密钥部分：  使用PBKDF2算法，通过指定的盐值、迭代次数和哈希算法（SHA-256），将用户输入的口令转换为对称加密所需的密钥。  确保了密钥生成过程的安全性和可靠性。  加密部分：  实现了文件的逐块读取和加密，保证了任意大小文件的加密能力。  使用了随机生成的初始化向量（IV），并将其写入输出文件，以确保加密过程的安全性。  加密过程中使用了AES-256-CBC算法，并通过OpenSSL的EVP接口进行实现。  解密部分：  实现了文件的逐块读取和解密，保证了任意大小文件的解密能力。  从输入文件中读取IV，并使用其进行解密初始化。  解密过程中同样使用了AES-256-CBC算法，并通过OpenSSL的EVP接口进行实现。 实验结果分析 功能实现：  加密和解密功能均成功实现，能够正确地对任意大小和类型的文件进行加密和解密操作。  命令行程序运行稳定，用户输入口令、源文件路径和目标文件路径后，能够正确执行加密或解密操作。  性能评估：  加密和解密速度较快，能够满足一般文件加密的需求。  内存占用合理，没有出现内存泄漏等问题。  安全性评估：  使用了强密码算法（AES-256-CBC）和安全的密钥派生方法（PBKDF2），确保了加密过程的安全性。  随机生成的IV和安全的盐值使用，增加了加密的复杂性和安全性。  工程能力锻炼：  通过本次实验，加深了对OpenSSL开源密码库的理解和使用。  提高了接口分析能力和工具设计的工程能力。  学会了如何设计和实现一个功能正确、界面友好的命令行工具。 改进想法 增加异常处理：  在加密和解密过程中，可以增加更多的异常处理逻辑，以应对可能出现的各种错误情况。  例如，可以捕获文件读写错误、内存分配错误等，并给出相应的错误提示。  优化用户体验：  可以对命令行程序进行进一步的优化，增加更多的用户提示和交互功能。  例如，可以添加帮助信息、参数验证等功能，提高程序的易用性和友好性。  支持更多算法：  可以进一步扩展程序的功能，支持更多的对称加密算法和非对称加密算法。  这样可以为用户提供更多的选择，满足不同场景下的加密需求。  性能优化：  可以对加密和解密过程进行性能优化，提高程序的运行效率。  例如，可以使用多线程或异步I/O等技术，提高文件的读写速度和加密解密速度。  经过上面的分析，本次实验成功实现了基于OpenSSL开源密码库的文件加解密工具，并锻炼了接口分析能力和工具设计的工程能力。通过实验结果的综合分析，我们可以看到该工具在功能、性能和安全性等方面均表现出色，但仍存在一些可以改进的地方。希望这些建议能够对后续的实验和研究有所帮助。 | | | |
| 结论分析与体会：  **结论分析与体会**  **实验中遇到的问题和解决的方法**   1. **密文输出问题**：    * **问题**：直接打印密文时，由于密文包含非ASCII字符，导致控制台显示乱码或特殊符号。    * **解决方法**：将密文转换为十六进制字符串进行输出。具体实现可以通过遍历密文字节数组，将每个字节转换为对应的两位十六进制数，然后拼接成字符串输出。 2. **口令派生密钥问题**：    * **问题**：在新版OpenSSL（3.x）中，pbkdf2相关的头文件和函数被弃用，导致无法引用和使用。    * **解决方法**：查阅新版OpenSSL的文档，发现可以使用openssl/kdf.h中的函数进行口令派生密钥的生成。具体使用了KDF\_derive函数，并通过设置合适的参数（如KDF类型、盐值、迭代次数等）来实现。 3. **文件二进制读写问题**：    * **问题**：使用OpenSSL自带的BIO进行文件读写时，经常遇到读取或写入失败的问题，且网上相关资料较少。    * **解决方法**：放弃使用BIO，转而使用C++标准库中的rb（二进制读取）和wb（二进制写入）模式进行文件的读写操作。这种方式更加直接且易于控制，成功解决了文件读写的问题。   **实验结果达到设计目标的程度**  实验结果成功达到了设计目标。通过封装对称算法（ECB\CBC\CFB\OFB）的加解密函数，实现了能够加密解密任意长度消息的接口。同时，利用口令派生密钥的方法，实现了一个命令行程序，该程序可以对操作系统中的任意大小和类型的文件进行加密和解密。加密后的文件可以通过相同的口令进行解密，还原为明文文件。  **还可以进行哪些改进**   1. **增强安全性**：    * 可以增加对盐值的随机性和复杂性的要求，以提高口令派生密钥的安全性。    * 可以考虑使用更安全的加密算法和模式，如GCM模式，以提供更强的加密和认证功能。 2. **优化用户界面**：    * 可以设计更加友好的用户界面，如使用图形化界面（GUI）替代命令行界面（CLI），以提高用户体验。    * 可以增加错误提示和日志记录功能，以便用户更好地了解程序的运行状态和错误信息。 3. **提高性能**：    * 可以对加密和解密过程进行优化，如使用多线程或并行计算来提高处理速度。    * 可以对内存使用进行优化，以减少程序的内存占用。   **实验得到哪些收获和启发**   1. **加深了对加密算法的理解**：    * 通过实现对称算法的加解密函数接口，加深了对加密算法原理和实现方式的理解。    * 了解了口令派生密钥的重要性和实现方法，提高了对密码学安全性的认识。 2. **提高了编程能力**：    * 通过解决实验中遇到的问题，提高了编程能力和解决问题的能力。    * 学会了使用C++标准库进行文件的二进制读写操作，增强了编程技能。 3. **培养了工程实践能力**：    * 通过设计和实现一个功能正确的文件加解密工具，培养了工程实践能力和系统设计能力。    * 学会了如何根据需求进行功能设计、接口封装和模块划分，提高了软件开发的综合能力。 | | | |

山东大学 软件 学院

**密码学原理与实践** 课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：202200201095 | 姓名：杨伟康 | | 班级：22网安 |
| 实验编号：实验二 | | | |
| 实验题目：**公钥加密算法实验** | | | |
| 实验学时：8 | | 实验日期： 11.22 | |
| 实验目的与内容：  **目的**：理解开源密码库实现的基本架构，熟悉公钥和对称算法的加解密函数封装与调用，并能能够利用开源设计接口进行二次封装，并实现一个功能正确的采用公钥密码和对称密码算法相结合的文件加解密签名验证工具。锻炼接口分析能力和工具设计的工程能力。  **内容**：下载OPENSSL开源软件源码，编译链接形成相应操作系统下的函数库，通过阅读网站和软件说明，至少了解其中RSA算法以及DES和AES算法的函数接口，设计基于公钥和对称钥密码算法相结合的文件加解密方案。要求：能够生成用户的RSA密钥对，采用对称算法生成对称加密的密钥并用公钥加密，然后用对称密钥加密文件内容；采用公钥解密对称密钥，并用对称密钥解密文件。文件加密时要求生成签名，解密文件时要求能够验证签名。 | | | |
| 硬件环境：  *机型，CPU，内存等主要参数。*  *若用虚拟机，则宿主机与虚拟机参数分别列出。*  处理器 11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz 2.69 GHz  机带 RAM 16.0 GB (15.9 GB 可用)  设备 ID 9DCDB7C5-EBBA-4C1F-845F-94B127DFAE09  产品 ID 00342-30515-76480-AAOEM  系统类型 64 位操作系统, 基于 x64 的处理器  笔和触控 没有可用于此显示器的笔或触控输入 | | | |
| 软件开发环境：  *若用虚拟机，请写明虚拟机软件及版本，以及宿主机操作系统版本。*  Visual studio2022+OPENSSL3.3.2 | | | |
| 实验步骤与内容： 实验思路 实验的核心目标是设计一个文件加解密工具，该工具结合公钥密码（RSA）和对称密码（AES）算法。具体步骤如下：   1. **生成密钥对**：    * 生成RSA密钥对（公钥和私钥）。    * 生成的密钥对将用于后续的文件加密和解密。 2. **加密对称密钥**：    * 生成一个对称密钥（如AES密钥）。    * 使用RSA公钥加密该对称密钥。 3. **加密文件**：    * 使用AES算法和生成的对称密钥加密文件内容。    * 将加密后的对称密钥、初始化向量（IV）以及加密后的文件内容写入到输出文件中。 4. **签名生成与验证**：    * 在加密文件时生成签名。    * 在解密文件时验证签名。 5. **解密文件**：    * 从加密文件中读取加密的对称密钥、IV和文件内容。    * 使用RSA私钥解密对称密钥。    * 使用解密后的对称密钥和IV解密文件内容。    * 验证签名以确保文件未被篡改。  实验模型描述  1. **密钥生成模块**：    * 使用OpenSSL的EVP接口生成RSA密钥对。   **c**   |  |  | | --- | --- | |  |  | |  | // Generate RSA key pair and save to files  int generate\_rsa\_keypair(const char\* public\_key\_file, const char\* private\_key\_file) {  EVP\_PKEY\_CTX\* ctx = EVP\_PKEY\_CTX\_new\_id(EVP\_PKEY\_RSA, NULL);  if (!ctx || EVP\_PKEY\_keygen\_init(ctx) <= 0) handle\_errors();  if (EVP\_PKEY\_CTX\_set\_rsa\_keygen\_bits(ctx, RSA\_KEY\_BITS) <= 0) handle\_errors();  EVP\_PKEY\* pkey = NULL;  if (EVP\_PKEY\_keygen(ctx, &pkey) <= 0) handle\_errors();  FILE\* pub\_file = fopen(public\_key\_file, "wb");  FILE\* priv\_file = fopen(private\_key\_file, "wb");  if (!pub\_file || !priv\_file) handle\_errors();  if (!PEM\_write\_PUBKEY(pub\_file, pkey) || !PEM\_write\_PrivateKey(priv\_file, pkey, NULL, NULL, 0, NULL, NULL)) handle\_errors();  fclose(pub\_file);  fclose(priv\_file);  EVP\_PKEY\_free(pkey);  EVP\_PKEY\_CTX\_free(ctx);  return 0; // Success  } |  1. **对称密钥加密模块**：    * 使用RSA公钥加密对称密钥。   **c**   |  |  | | --- | --- | |  |  | |  | // Encrypt symmetric key using RSA public key  int rsa\_encrypt\_key(const char\* public\_key\_file, unsigned char\* key, size\_t key\_len, unsigned char\* encrypted\_key, size\_t\* encrypted\_key\_len) {  FILE\* pub\_file = fopen(public\_key\_file, "rb");  if (!pub\_file) return -1;  EVP\_PKEY\* public\_key = PEM\_read\_PUBKEY(pub\_file, NULL, NULL, NULL);  fclose(pub\_file);  if (!public\_key) return -1;  EVP\_PKEY\_CTX\* ctx = EVP\_PKEY\_CTX\_new(public\_key, NULL);  if (!ctx || EVP\_PKEY\_encrypt\_init(ctx) <= 0) return -1;  if (EVP\_PKEY\_encrypt(ctx, NULL, encrypted\_key\_len, key, key\_len) <= 0) return -1;  if (EVP\_PKEY\_encrypt(ctx, encrypted\_key, encrypted\_key\_len, key, key\_len) <= 0) return -1;  EVP\_PKEY\_free(public\_key);  EVP\_PKEY\_CTX\_free(ctx);  return 0;  } |  1. **文件加密模块**：    * 使用AES-CBC模式加密文件内容。    * 将IV、加密后的对称密钥和加密后的文件内容写入到输出文件中。   **c**   |  |  | | --- | --- | |  |  | |  | // Encrypt file  int encrypt\_file(const char\* input\_filename, const char\* output\_filename, unsigned char\* key, const char\* public\_key\_file) {  FILE\* input\_file = fopen(input\_filename, "rb");  if (!input\_file) return -1;  FILE\* output\_file = fopen(output\_filename, "wb");  if (!output\_file) {  fclose(input\_file);  return -1;  }  unsigned char iv[IV\_SIZE];  if (!RAND\_bytes(iv, IV\_SIZE)) return -1;  if (fwrite(iv, 1, IV\_SIZE, output\_file) != IV\_SIZE) return -1;  size\_t encrypted\_key\_len;  unsigned char encrypted\_key[RSA\_KEY\_BITS / 8];  if (rsa\_encrypt\_key(public\_key\_file, key, KEY\_LENGTH, encrypted\_key, &encrypted\_key\_len) != 0) return -1;  if (fwrite(&encrypted\_key\_len, sizeof(size\_t), 1, output\_file) != 1 || fwrite(encrypted\_key, 1, encrypted\_key\_len, output\_file) != encrypted\_key\_len) return -1;  EVP\_CIPHER\_CTX\* ctx = EVP\_CIPHER\_CTX\_new();  if (!ctx) return -1;  if (EVP\_EncryptInit\_ex(ctx, EVP\_aes\_256\_cbc(), NULL, key, iv) != 1) return -1;  unsigned char in\_buffer[BUFFER\_SIZE];  int bytes\_read;  while ((bytes\_read = fread(in\_buffer, 1, BUFFER\_SIZE, input\_file)) > 0) {  int out\_len;  unsigned char out\_buffer[BUFFER\_SIZE + EVP\_MAX\_BLOCK\_LENGTH];  if (EVP\_EncryptUpdate(ctx, out\_buffer, &out\_len, in\_buffer, bytes\_read) != 1) return -1;  if (fwrite(out\_buffer, 1, out\_len, output\_file) != out\_len) return -1;  }  int final\_len;  unsigned char final\_buffer[EVP\_MAX\_BLOCK\_LENGTH];  if (EVP\_EncryptFinal\_ex(ctx, final\_buffer, &final\_len) != 1) return -1;  if (fwrite(final\_buffer, 1, final\_len, output\_file) != final\_len) return -1;  EVP\_CIPHER\_CTX\_free(ctx);  fclose(input\_file);  fclose(output\_file);  return 0;  } |  1. **签名生成与验证模块**：    * 随机生成HMAC的对称密钥，单独存储，并解密时使用密钥验证。   **c**   |  |  | | --- | --- | |  |  | |  | int generate\_hmac(const char\* input\_filename, const char\* hmac\_filename, unsigned char\* hmac\_key, size\_t hmac\_key\_len) {  FILE\* input\_file = fopen(input\_filename, "rb");  if (!input\_file) return -1;  FILE\* hmac\_file = fopen(hmac\_filename, "wb");  if (!hmac\_file) {  fclose(input\_file);  return -1;  }  // Optional: Write HMAC key to a file  FILE\* key\_file = fopen("hmac\_key.bin", "wb");  if (key\_file) {  fwrite(hmac\_key, 1, hmac\_key\_len, key\_file);  fclose(key\_file);  }  unsigned char buffer[BUFFER\_SIZE];  unsigned char hmac\_value[EVP\_MAX\_MD\_SIZE];  unsigned int hmac\_len = 0;  HMAC(EVP\_sha256(), hmac\_key, hmac\_key\_len, NULL, 0, hmac\_value, &hmac\_len); // Initialize HMAC  size\_t bytes\_read;  while ((bytes\_read = fread(buffer, 1, BUFFER\_SIZE, input\_file)) > 0) {  HMAC(EVP\_sha256(), hmac\_key, hmac\_key\_len, buffer, bytes\_read, hmac\_value, &hmac\_len);  }  fwrite(hmac\_value, 1, hmac\_len, hmac\_file);  fclose(input\_file);  fclose(hmac\_file);  return 0;  } |  1. **文件解密模块**：    * 从加密文件中读取IV、加密后的对称密钥和加密后的文件内容。    * 使用RSA私钥解密对称密钥。    * 使用解密后的对称密钥和IV解密文件内容。    * 验证签名。   这里展示解密文件的源代码，实现了上面功能，不再分开展示  **c**   |  |  | | --- | --- | |  |  | |  | #include <openssl/evp.h>  #include <openssl/pem.h>  #include <openssl/rsa.h>  #include <openssl/err.h>  #include <openssl/hmac.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <openssl/applink.c>  #define BUFFER\_SIZE 8192  #define KEY\_LENGTH 32 // AES-256 requires 32 bytes  #define IV\_SIZE 16  #define RSA\_KEY\_BITS 2048  #define ENCRYPTED\_KEY\_LENGTH (RSA\_KEY\_BITS / 8) // 256 bytes for RSA-2048  #define HMAC\_LENGTH EVP\_MAX\_MD\_SIZE // HMAC length  #define HMAC\_FILENAME "hmac\_data.bin" // HMAC 文件名  #define HMAC\_KEY\_FILENAME "hmac\_key.bin" // HMAC 密钥文件名  void handle\_errors() {  ERR\_print\_errors\_fp(stderr);  abort();  }  // Decrypt symmetric key using RSA private key  int rsa\_decrypt\_key(const char\* private\_key\_file, unsigned char\* encrypted\_key, size\_t encrypted\_key\_len, unsigned char\* decrypted\_key, size\_t\* decrypted\_key\_len) {  FILE\* priv\_file = fopen(private\_key\_file, "rb");  if (!priv\_file) {  fprintf(stderr, "Failed to open private key file\n");  return -1;  }  EVP\_PKEY\* private\_key = PEM\_read\_PrivateKey(priv\_file, NULL, NULL, NULL);  fclose(priv\_file);  if (!private\_key) {  fprintf(stderr, "Failed to read private key\n");  handle\_errors();  return -1;  }  EVP\_PKEY\_CTX\* ctx = EVP\_PKEY\_CTX\_new(private\_key, NULL);  if (!ctx) {  fprintf(stderr, "Failed to create PKEY context\n");  handle\_errors();  EVP\_PKEY\_free(private\_key);  return -1;  }  if (EVP\_PKEY\_decrypt\_init(ctx) <= 0) {  fprintf(stderr, "Failed to initialize decryption context\n");  handle\_errors();  EVP\_PKEY\_CTX\_free(ctx);  EVP\_PKEY\_free(private\_key);  return -1;  }  // First call to EVP\_PKEY\_decrypt() to determine the buffer length  size\_t out\_len = \*decrypted\_key\_len;  if (EVP\_PKEY\_decrypt(ctx, NULL, &out\_len, encrypted\_key, encrypted\_key\_len) <= 0) {  fprintf(stderr, "Failed to determine buffer length\n");  handle\_errors();  EVP\_PKEY\_CTX\_free(ctx);  EVP\_PKEY\_free(private\_key);  return -1;  }  \*decrypted\_key\_len = out\_len; // Set the output length  if (EVP\_PKEY\_decrypt(ctx, decrypted\_key, decrypted\_key\_len, encrypted\_key, encrypted\_key\_len) <= 0) {  fprintf(stderr, "Failed to decrypt key\n");  handle\_errors();  EVP\_PKEY\_CTX\_free(ctx);  EVP\_PKEY\_free(private\_key);  return -1;  }  EVP\_PKEY\_CTX\_free(ctx);  EVP\_PKEY\_free(private\_key);  return 0;  }  // Function to print key in hex  void print\_key\_hex(unsigned char\* key, size\_t key\_len) {  for (size\_t i = 0; i < key\_len; ++i) {  printf("%02x", key[i]);  }  printf("\n");  }  // Read HMAC key from file  int read\_hmac\_key(const char\* filename, unsigned char\* hmac\_key, size\_t key\_len) {  FILE\* file = fopen(filename, "rb");  if (!file) {  fprintf(stderr, "Failed to open HMAC key file\n");  return -1;  }  if (fread(hmac\_key, 1, key\_len, file) != key\_len) {  fprintf(stderr, "Failed to read HMAC key\n");  fclose(file);  return -1;  }  fclose(file);  // Print the HMAC key in hex format  printf("HMAC Key: ");  print\_key\_hex(hmac\_key, key\_len);  return 0;  }  // Verify HMAC  int verify\_hmac(const char\* input\_filename, unsigned char\* hmac\_key, size\_t hmac\_key\_len, const unsigned char\* decrypted\_data, size\_t data\_len) {  unsigned char calculated\_hmac[HMAC\_LENGTH];  unsigned int calculated\_hmac\_len = 0;  // Calculate HMAC on the decrypted data  HMAC(EVP\_sha256(), hmac\_key, hmac\_key\_len, decrypted\_data, data\_len, calculated\_hmac, &calculated\_hmac\_len);  // Read the stored HMAC  FILE\* hmac\_file = fopen(HMAC\_FILENAME, "rb");  if (!hmac\_file) {  fprintf(stderr, "Failed to open HMAC file\n");  return -1;  }  unsigned char stored\_hmac[HMAC\_LENGTH];  size\_t stored\_hmac\_len = fread(stored\_hmac, 1, HMAC\_LENGTH, hmac\_file);  fclose(hmac\_file);  // Compare HMACs  if (stored\_hmac\_len != calculated\_hmac\_len || memcmp(stored\_hmac, calculated\_hmac, stored\_hmac\_len) != 0) {  fprintf(stderr, "HMAC verification failed\n");  return -1; // HMACs do not match  }  return 0; // HMAC verified successfully  }  // Decrypt file  int decrypt\_file(const char\* encrypted\_filename, const char\* output\_filename, const char\* private\_key\_file) {  FILE\* encrypted\_file = fopen(encrypted\_filename, "rb");  if (!encrypted\_file) {  fprintf(stderr, "Failed to open encrypted file\n");  return -1;  }  FILE\* output\_file = fopen(output\_filename, "wb");  if (!output\_file) {  fprintf(stderr, "Failed to open output file\n");  fclose(encrypted\_file);  return -1;  }  // Read the IV  unsigned char iv[IV\_SIZE];  if (fread(iv, 1, IV\_SIZE, encrypted\_file) != IV\_SIZE) {  fprintf(stderr, "Failed to read IV\n");  fclose(encrypted\_file);  fclose(output\_file);  return -1;  }  // Read the encrypted key length and the encrypted key  size\_t encrypted\_key\_len;  if (fread(&encrypted\_key\_len, sizeof(size\_t), 1, encrypted\_file) != 1) {  fprintf(stderr, "Failed to read encrypted key length\n");  fclose(encrypted\_file);  fclose(output\_file);  return -1;  }  unsigned char encrypted\_key[ENCRYPTED\_KEY\_LENGTH];  if (fread(encrypted\_key, 1, encrypted\_key\_len, encrypted\_file) != encrypted\_key\_len) {  fprintf(stderr, "Failed to read encrypted key\n");  fclose(encrypted\_file);  fclose(output\_file);  return -1;  }  // Decrypt the symmetric key  unsigned char decrypted\_key[KEY\_LENGTH];  size\_t decrypted\_key\_len = KEY\_LENGTH;  if (rsa\_decrypt\_key(private\_key\_file, encrypted\_key, encrypted\_key\_len, decrypted\_key, &decrypted\_key\_len) != 0) {  fclose(encrypted\_file);  fclose(output\_file);  return -1;  }  // Print the decrypted symmetric key in hex format  printf("Decrypted Symmetric Key: ");  print\_key\_hex(decrypted\_key, decrypted\_key\_len);  // Read HMAC key  unsigned char hmac\_key[KEY\_LENGTH]; // Adjust size as needed  if (read\_hmac\_key(HMAC\_KEY\_FILENAME, hmac\_key, KEY\_LENGTH) != 0) {  fclose(encrypted\_file);  fclose(output\_file);  return -1;  }  // Initialize the AES decryption context  EVP\_CIPHER\_CTX\* ctx = EVP\_CIPHER\_CTX\_new();  if (!ctx) {  fprintf(stderr, "Failed to create decryption context\n");  fclose(encrypted\_file);  fclose(output\_file);  return -1;  }  if (EVP\_DecryptInit\_ex(ctx, EVP\_aes\_256\_cbc(), NULL, decrypted\_key, iv) != 1) {  fprintf(stderr, "Failed to initialize decryption context\n");  EVP\_CIPHER\_CTX\_free(ctx);  fclose(encrypted\_file);  fclose(output\_file);  return -1;  }  unsigned char in\_buffer[BUFFER\_SIZE];  unsigned char out\_buffer[BUFFER\_SIZE + EVP\_MAX\_BLOCK\_LENGTH];  int bytes\_read, out\_len;  size\_t total\_decrypted\_len = 0;  // Decrypt the file and store the plaintext  while ((bytes\_read = fread(in\_buffer, 1, BUFFER\_SIZE, encrypted\_file)) > 0) {  if (EVP\_DecryptUpdate(ctx, out\_buffer, &out\_len, in\_buffer, bytes\_read) != 1) {  fprintf(stderr, "Decryption failed\n");  EVP\_CIPHER\_CTX\_free(ctx);  fclose(encrypted\_file);  fclose(output\_file);  return -1;  }  fwrite(out\_buffer, 1, out\_len, output\_file);  total\_decrypted\_len += out\_len;  }  if (EVP\_DecryptFinal\_ex(ctx, out\_buffer, &out\_len) != 1) {  fprintf(stderr, "Final decryption step failed\n");  EVP\_CIPHER\_CTX\_free(ctx);  fclose(encrypted\_file);  fclose(output\_file);  return -1;  }  fwrite(out\_buffer, 1, out\_len, output\_file);  total\_decrypted\_len += out\_len;  EVP\_CIPHER\_CTX\_free(ctx);  fclose(encrypted\_file);  fclose(output\_file);  printf("Decryption successful\n");  // Verify HMAC after decryption  unsigned char\* decrypted\_data = (unsigned char\*)malloc(total\_decrypted\_len);  if (!decrypted\_data) {  fprintf(stderr, "Memory allocation failed\n");  return -1;  }  // Reopen the output file to read the decrypted data  output\_file = fopen(output\_filename, "rb");  if (!output\_file) {  fprintf(stderr, "Failed to open output file for reading\n");  free(decrypted\_data);  return -1;  }  fread(decrypted\_data, 1, total\_decrypted\_len, output\_file);  fclose(output\_file);  // Verify HMAC using the decrypted data  if (verify\_hmac(encrypted\_filename, hmac\_key, KEY\_LENGTH, decrypted\_data, total\_decrypted\_len) != 0) {  free(decrypted\_data);  return -1;  }  free(decrypted\_data);  printf("HMAC verification successful\n");  return 0;  }  int main(int argc, char\* argv[]) {  if (argc != 4) {  fprintf(stderr, "Usage: %s <encrypted file> <output file> <private key file>\n", argv[0]);  exit(EXIT\_FAILURE);  }  const char\* encrypted\_file = argv[1];  const char\* output\_file = argv[2];  const char\* private\_key\_file = argv[3];  if (decrypt\_file(encrypted\_file, output\_file, private\_key\_file) != 0) {  perror("Decryption failed");  exit(EXIT\_FAILURE);  }  return 0;  } |  主要数据结构分析说明  * **EVP\_PKEY**：表示公钥或私钥。 * **EVP\_PKEY\_CTX**：用于公钥和私钥的操作上下文。 * **EVP\_CIPHER\_CTX**：用于对称加密操作的上下文。 * **FILE**：文件操作结构，用于文件的读写。  项目管理文件的说明 使用Visual Studio 2022管理项目，包括源文件、头文件和OpenSSL库文件的链接。确保项目配置正确，包含OpenSSL的头文件路径和库文件路径。 实验投入的实际学时数 预计实验投入时间为20学时，包括理论学习、代码编写、调试和实验报告撰写。 调试排错过程的记录  * **问题1**：在编译时遇到未定义的引用错误。   + **解决**：由于不同版本的Openssl，所使用的函数代码都不同，所以在调用函数以及引用库时，都得区别版本，确保OpenSSL库文件路径正确，并正确配置链接器选项。   + **比如：**   3.0版本以前,HMAC都是调用openssl/hmac.h中的函数，而在新版本，要使用信封中的函数openssl/evp.h   * **问题2**：加密后的文件无法正确解密。   + **解决**：检查IV和加密密钥的使用是否正确，确保在加密和解密过程中使用相同的IV和密钥。   解密时，要根据加密信息写入文件的顺序，比如 本次我的加密文件内部储存顺序为：iv，加密过的对称密钥，加密数据，HMAC.   * **问题3**：签名验证失败。   + **解决**：检查签名生成和验证过程中的参数是否正确，确保使用正确的HMAC对应的密钥生成签名并使用正确的密钥验证签名。   + **比如：**   这里我用十六进制将两个密钥打印出来，用于解密时对比密钥是否解密成功    但是解密时，明明两个密钥都成功解密出来，但是还是HMAC验证还是出错    经过原因的排查，是我验证的时候，我直接用加密信息计算的HMAC，  而最初我的HMAC是用明文计算的，所以我先计算出明文，再验证HMAC，根据HMAC的验证选择是否输出明文。  经过修改可以成功验证   运行结果  * **加密文件**：成功生成加密文件，文件内容被AES加密，对称密钥被RSA加密。   * **解密文件**：成功解密文件，恢复原始文件内容，签名验证通过。    **多种方式测试结果的记录**   * **测试1**：使用不同大小和类型的文件进行测试，确保加密和解密过程能够处理不同大小和类的文件。   1. 加密文本     进行加密    成功加密  因为加密成的是txt文件，所以能够打开但还是乱码    加密成其他类型也可以如：    加密成enc文件    此时解密    可以看到加解密的对称密钥一致    文件虽然是不同类型，也成功恢复   * 1. 加密ppt     内容为    还是上面的流程    恢复成功    但是可以看到我中间有一次失败了，  是因为我文件处于打开状态，程序无法成功写入   * **测试2**：使用不同的对称密钥和RSA密钥对进行测试，确保加密和解密过程的正确性。     生成不同的RSA密钥    再次测试仍然成功     * **测试3**：对加密文件进行篡改，验证签名验证过程的正确性。     篡改加密文件    后面加入数字  解密    可以看到HMAC验证失败 实验结果的分析综合 通过本次实验，成功实现了基于公钥和对称密码算法相结合的文件加解密签名验证工具。实验过程中，深入理解了OpenSSL库的使用，掌握了RSA和AES算法的实现和调用方法。通过调试和测试，验证了加密和解密过程的正确性，以及签名生成和验证的有效性。实验结果符合预期，达到了实验目的。 | | | |
| 结论分析与体会：  **实验结果达到设计目标的程度**  实验结果完全达到了设计目标。成功实现了一个基于公钥和对称密码算法相结合的文件加解密签名验证工具。该工具能够生成用户的RSA密钥对，采用对称算法（如AES）生成对称加密密钥，并用公钥加密该对称密钥。然后，使用对称密钥加密文件内容。解密时，先使用公钥解密对称密钥，再用对称密钥解密文件内容。同时，文件加密时生成签名，解密时验证签名，确保了数据的完整性和真实性。  **还可以进行哪些改进**   1. **增强错误处理**：    * 在加密和解密过程中增加更详细的错误处理逻辑，以便在出现问题时能够提供更具体的错误信息。 2. **支持更多算法**：    * 扩展工具以支持更多的对称和公钥加密算法，以满足不同场景下的需求。 3. **优化性能**：    * 对加密和解密过程进行优化，提高处理速度，特别是在处理大文件时。 4. **用户界面改进**：    * 提供一个更友好的用户界面，使用户能够更容易地使用该工具进行文件加解密和签名验证。   **实验得到哪些收获和启发**   1. **深入理解OpenSSL库**：    * 通过实验，深入了解了OpenSSL库的使用方法和功能，掌握了RSA和AES等加密算法的实现和调用方法。 2. **提高编程能力**：    * 在实现工具的过程中，提高了编程能力和问题解决能力，特别是在调试和测试阶段。 3. **理解密码学原理**：    * 通过实验，加深了对密码学原理的理解，特别是公钥和对称密码算法的结合使用。 4. **实践项目管理**：    * 使用Visual Studio 2022管理项目，学会了如何配置项目、链接库文件以及进行调试和测试。 5. **安全意识增强**：    * 通过实现加密和签名验证工具，增强了数据安全意识，了解了如何保护数据的机密性、完整性和真实性。 | | | |