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。 | | | |