创新创业实践报告

学生姓名 范六一 学号 202100460156

学院 网络空间安全 专业 网络空间安全

实验时间 2023 年 4 月 10 日

一. Project11

1. 加密算法

6.1 加密算法

设需要发送的消息为比特串 M，$klen$ 为 M 的比特长度。

为了对明文 M 进行加密，作为加密者的用户 A 应实现以下运算步骤：

A_1：用随机数发生器产生随机数 $k\in[1, n-1]$；

A_2：计算椭圆曲线点 $C_1=[k]G=(x_1, y_1)$，按 GB/T 32918.1—2016 中 4.2.9 和 4.2.5 给出的方法，将 C_1 的数据类型转换为比特串；

A_3：计算椭圆曲线点 $S=[h]P_B$，若 S 是无穷远点，则报错并退出；

A_4：计算椭圆曲线点 $[k]P_B=(x_2, y_2)$，按 GB/T 32918.1—2016 中 4.2.6 和 4.2.5 给出的方法，将坐标 x_2、y_2 的数据类型转换为比特串；

A_5：计算 $t=KDF(x_2\|y_2, klen)$，若 t 为全 0 比特串，则返回 A_1；

A_6：计算 $C_2=M\oplus t$；

A_7：计算 $C_3=Hash(x_2\|M\|y_2)$；

A_8：输出密文 $C=C_1\|C_3\|C_2$。

注：加密过程的示例参见附录 A。

2.解密算法

C为密文字符串,klen为密文中C2的长度

1. C1 = C里面获取，验证C1是否满足椭圆曲线。 ⇒ C2长度确定，可以获取C1内容。
2. S = [h]C1，S为无穷点，退出。
3. (x2, y2) = [d]C1
4. t = KDF (m2 ‖ y2, klen)
5. $\widetilde{M}$ = C2 + t
6. u = Hash(x2 ‖ $\widetilde{M}$ ‖ y2), u? == C3
7. $\widetilde{M}$为明文

3.获得公私钥

椭圆曲线方程: $y^2 = x^3 + ax + b \mod p$

1. 确认a、b、p，确认曲线。
2. 选择一个点P (x_g, y_g)为基点。
3. 对曲线做切线、x对称点运行。次数为d, 运算倍点为Q
4. d为私钥, Q为公钥

4. add()函数实现椭圆曲线上的点加

```
def add(x1, y1, x2, y2):
    if x1==x2 and y1==p-y2:
        return False
    if x1!=x2:
        lamda=((y2-y1)*invert(x2-x1, p))%p
    else:
        lamda=(((3*x1*x1+a)%p)*invert(2*y1, p))%p
    x3=(lamda*lamda-x1-x2)%p
    y3=(lamda*(x1-x3)-y1)%p
    return x3, y3
```

5. mul_add（）实现椭圆曲线上的点乘

```
def mul_add(x,y,k):
    k=bin(k)[2:]
    qx,qy=x,y
    for i in range(1,len(k)):
        qx,qy=add(qx, qy, qx, qy)
        if k[i]=='1':
            qx,qy=add(qx, qy, x, y)
    return (qx,qy)
```

6. 同时根据原理实现了加解密函数

其中生成随机参数时，参数设置为标准文档中的值，但若需要随机生成，可以通过修改注释。

```
def encrypt(m):
    plen=len(hex(p)[2:])
    m='0'*((4-(len(bin(int(m.encode().hex(),16))[2:])%4))%4)+bin(int(m.enco
    klen=len(m)
    while True:
        #k=randint(1,n)
        k=0x384F30353073AEECE7A1654330A96204D37982A3E15B2CB5
        while k==da:
            k=randint(1, n)
        x2,y2=mul_add(Pa[0],Pa[1],k)
        if(len(hex(p)[2:])*4==192):
            x2,y2='{:0192b}'.format(x2),'{:0192b}'.format(y2)
        else:
            x2, y2 = '{:0256b}'.format(x2), '{:0256b}'.format(y2)
        t=KDF(x2+y2, klen)
        if int(t,2)!=0:
            break
    x1,y1=mul_add(Gx, Gy,k)
    x1,y1=(plen-len(hex(x1)[2:]))*'0'+hex(x1)[2:],(plen-len(hex(y1)[2:]))*'
    c1=x1+y1
    c2=((klen//4)-len(hex(int(m,2)^int(t,2))[2:]))*'0'+hex(int(m,2)^int(t,2
    c3=Hash(hex(int(x2+m+y2,2))[2:].upper())
    return c1,c2,c3

def decrypt(c1,c2,c3):
    x1,y1=int(c1[:len(c1)//2],16),int(c1[len(c1)//2:],16)
    if pow(y1,2,p)!=(pow(x1,3,p)+a*x1+b)%p:
        return False
    x2,y2=mul_add(x1, y1, da)
    if (len(hex(p)[2:]) * 4 == 192):
        x2, y2 = '{:0192b}'.format(x2), '{:0192b}'.format(y2)
    else:
        x2, y2 = '{:0256b}'.format(x2), '{:0256b}'.format(y2)
    klen=len(c2)*4
    t=KDF(x2+y2, klen)
```

7. 密钥交换

哈希函数以及椭圆曲线上运算与 SM2 加解密相同。

原理可以理解为：Alice 随机生成一个 ra 计算[ra]G 发送给 bill，bill 随机生成一个 rb 计算[rb]G 发送给 Alice。对于 BILL，可以使用 Alice 的 ID 以及公钥计算 Za，同样可以计算自己的 Zb，之后利用 Alice 发送的 Ra 按照密钥交换 协议可以计算一个密钥。对于 Alice，同样可以计算 Za 和 Zb，并利用 Bill 的 Rb 按照密钥交换协议计算一个密钥。

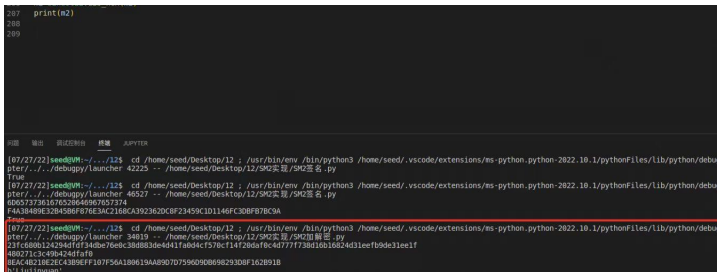
二者计算的密钥相同，从而达到密钥交换。所以在代码中先呈现基础操作 leftshift（）；Ti（）；FFi（）；GGi（）P0();P1()。最后实现密钥交换。

```
#ra=randint(1,n)
ra=0x83A2C9C8B96E5AF70BD480B472409A9A327257F1EBB73F5B073354B248668563
Ra=mul_add(Gx,Gy,ra)


Pa=mul_add(Gx,Gy,da)
m="huangtingchenyi"
c1,c2,c3=encrypt(m)
m2=decrypt(c1,c2,c3)
m2=binascii.a2b_hex(m2)
print(m2)

#rb=randint(1,n)
rb=0x33FE21940342161C55619C4A0C060293D543C80AF19748CE176D83477DE71C80
Rb=mul_add(Gx,Gy,rb)
print(B_key_exchange(*Ra))
print(A_key_exchange(*Rb))
```

可以看到结果



二．Project14

1. PGP 介绍

PGP 是一个加密程序，为数据通信提供加密隐私和身份验证。PGP 用于对文本、电子邮件、文件、目录和整个磁盘分区进行签名、加密和解密，并提高电子邮件通信的安全性。PGP 加密使用散列，数据压缩，对称密钥加密，最后是公钥加密的串行组合。其中最关键的是两种形式的加密的组合：对称密钥加密和非对称密钥加密。

在实现 PGP 加密的过程中，首先使用对称密钥加密算法对原始数据进行加密。对称密钥加密算法包括 DES、AES 等，这些算法能够快速地加密和解密数据，但是需要发送方和接收方之间共享密钥。

为了避免在网络上传输密钥，PGP 使用了公钥加密算法。其中公钥用于加密，而私钥用于解密。公钥加密算法包括 RSA、DSA 等，他们具有极高的安全性，但是加密和解密速度比对称密钥加密算法慢得多。

2. 大致原理

调用 GMSSL 库中封装好的 SM2/SM4 加解密函数；加密时使用对称加密算法 SM4 加密消息，非对称加密算法 SM2 加密会话密钥；解密时先使用 SM2 解密求得会话密钥，再通过 SM4 和会话密钥求解原消息。

3.代码说明

def epoint_mod(a, n)

是椭圆曲线上的模运算（a mod n）；

def epoint_modmult(a, b, n)

是椭圆曲线上的模乘运算（a*b^(-1)mod n）；

def epoint_add(P, Q, a, p)

是椭圆曲线上的点乘运算，返回值是 P+Q;

def epoint_mult(k, P, a, p)

是椭圆曲线上的点乘运算，返回值等于 k * P；

def keygen(a, p, n, G)

生成 SM2 算法的公私钥对。

三．Project5

是默克尔树的构建过程：

将待验证数据划分为固定大小的数据块。每个数据块都被哈希函数哈希，并产生一个唯一的哈希值。

如果数据块的数量为奇数，则将最后一个数据块复制一份，使得数据块的数量为偶数。
将所有数据块的哈希值两两配对，并将每对的哈希值连接在一起。
对每对哈值进行再次哈希，得到新的哈希值。
重复步骤 3 和步骤 4，直到只剩下一个哈希值，这个哈希值就是默克尔树的根哈希值，也称为树根。
通过构建默克尔树，可以有效验证大量数据的完整性。如果有任何数据块被篡改，或者数据块的顺序被改变，那么最终计算得到的树根哈希值也会发生变化。因此，只需要比对树根哈希值，就可以检测到数据是否被篡改。
附上代码：

```
#include <iostream>
#include <string>
#include <vector>
#include <openssl/ssl.h>
#include <openssl/err.h>

// Structure representing a Merkle Tree node
struct MerkleNode {
    std::string hash;
    MerkleNode* left;
    MerkleNode* right;
};

// Function to create a Merkle Tree from certificate hashes
MerkleNode* createMerkleTree(const std::vector<std::string>& certHashes) {
    std::vector<MerkleNode*> nodes;

    // Create leaf nodes for each certificate hash
    for (const auto& hash : certHashes) {
        MerkleNode* node = new MerkleNode();
        node->hash = hash;
        node->left = nullptr;
        node->right = nullptr;
        nodes.push_back(node);
    }

    // Build the Merkle Tree by combining nodes pairwise
    while (nodes.size() > 1) {
        std::vector<MerkleNode*> parentNodes;
        for (size_t i = 0; i < nodes.size(); i += 2) {
            MerkleNode* parentNode = new MerkleNode();
            parentNode->hash = nodes[i]->hash + nodes[i + 1]->hash; // Combine hashes
            parentNode->left = nodes[i];
            parentNode->right = nodes[i + 1];
            parentNodes.push_back(parentNode);
```

```
            }
            if (nodes.size() % 2 == 1) {
                  parentNodes.push_back(nodes[nodes.size() - 1]); // If odd number of nodes, add
the last one
            }
            nodes = parentNodes;
      }

      return nodes[0];}

std::string retrieveCertificate(const std::string& hash) {

      std::string certificate = "<certificate>"; // Sample certificate
      return certificate;
}

// Function to verify a certificate by comparing its hash against the Merkle Tree
bool verifyCertificate(const std::string& certificate, MerkleNode* merkleRoot) {
      // Calculate the hash of the received certificate
      // and compare it with the hash stored in the Merkle Tree
      // If they match, the certificate is valid

      std::string receivedHash = "<hash>"; // Replace with actual hash calculation

      return receivedHash == merkleRoot->hash;
}

int main() {
      // Sample certificate hashes
      std::vector<std::string> certificateHashes = {
            "<hash1>",
            "<hash2>",
            "<hash3>",
            // Add more certificate hashes as needed
      };

      // Create the Merkle Tree
      MerkleNode* merkleRoot = createMerkleTree(certificateHashes);

      // Retrieve a certificate by its hash
      std::string certificateHashToRetrieve = "<hash1>"; // Replace with actual hash
      std::string retrievedCertificate = retrieveCertificate(certificateHashToRetrieve);

      // Verify the retrieved certificate
```

```
    bool isCertificateValid = verifyCertificate(retrievedCertificate, merkleRoot);

    if (isCertificateValid) {
        std::cout << "Certificate is valid." << std::endl;
    } else {
        std::cout << "Certificate is invalid." << std::endl;
    }

    // Clean up memory
    // Implement your own cleanup logic based on your needs

    return 0;
}
```

四. project6

提供了一个基本的按照 RFC6962 实现 Merkle 树的示例，包括根据证书哈希创建 Merkle 树、通过网络通信获取证书以及验证获取的证书与 Merkle 树的一致性。你需要将占位符 <hash> 和 <certificate> 替换为实际的哈希值和证书，并根据需求实现自己的网络通信逻辑，并根据需求处理内存管理

附上代码：

```
#include <iostream>
#include <string>
#include <vector>
#include <openssl/ssl.h>
#include <openssl/err.h>

// 表示 Merkle 树节点的结构体
struct MerkleNode {
    std::string hash;
    MerkleNode* left;
    MerkleNode* right;
};

// 创建 Merkle 树，用于存储证书哈希
MerkleNode* createMerkleTree(const std::vector<std::string>& certHashes) {
    std::vector<MerkleNode*> nodes;

    // 为每个证书哈希创建叶节点
    for (const auto& hash : certHashes) {
        MerkleNode* node = new MerkleNode();
        node->hash = hash;
```

```
            node->left = nullptr;
            node->right = nullptr;
            nodes.push_back(node);
        }

        // 通过两两合并节点构建 Merkle 树
        while (nodes.size() > 1) {
            std::vector<MerkleNode*> parentNodes;
            for (size_t i = 0; i < nodes.size(); i += 2) {
                MerkleNode* parentNode = new MerkleNode();
                parentNode->hash = nodes[i]->hash + nodes[i + 1]->hash; // 合并哈希值
                parentNode->left = nodes[i];
                parentNode->right = nodes[i + 1];
                parentNodes.push_back(parentNode);
            }
            if (nodes.size() % 2 == 1) {
                parentNodes.push_back(nodes[nodes.size() - 1]); // 如果节点数为奇数，添加最后一个节点
            }
            nodes = parentNodes;
        }

        return nodes[0]; // 返回 Merkle 树的根节点
}

// 根据哈希值从网络中获取证书
std::string retrieveCertificate(const std::string& hash) {
        // 在此处实现你的网络通信代码，从可信源使用提供的哈希值获取证书
        std::string certificate = "<certificate>"; // 示例证书
        return certificate;
}

// 对比证书的哈希值与 Merkle 树中的哈希值进行验证
bool verifyCertificate(const std::string& certificate, MerkleNode* merkleRoot) {
        // 计算接收到的证书的哈希值并与 Merkle 树存储的哈希值进行对比
        // 如果相匹配，则证书有效

        std::string receivedHash = "<hash>"; // 替换为实际的哈希值计算

        return receivedHash == merkleRoot->hash;
}

int main() {
        // 示例证书哈希值
```

```
    std::vector<std::string> certificateHashes = {
        "<hash1>",
        "<hash2>",
        "<hash3>",
        // 根据需要添加更多证书哈希值
    };

    // 创建 Merkle 树
    MerkleNode* merkleRoot = createMerkleTree(certificateHashes);

    // 根据哈希值从网络中获取证书
    std::string certificateHashToRetrieve = "<hash1>"; // 替换为实际的哈希值
    std::string retrievedCertificate = retrieveCertificate(certificateHashToRetrieve);

    // 验证获取到的证书
    bool isCertificateValid = verifyCertificate(retrievedCertificate, merkleRoot);

    if (isCertificateValid) {
        std::cout << "证书有效。" << std::endl;
    } else {
        std::cout << "证书无效。" << std::endl;
    }

    // 清理内存
    // 根据需求实现自己的内存清理逻辑

    return 0;
}
```

五. project8

要在 ARM 指令集上实现 AES 算法，可以使用 AES 的基本操作：SubBytes、ShiftRows、MixColumns 和 AddRoundKey.在 ARM Cortex-M 系列微控制器上实现 128 位密钥的 AES 算法。实现了 AES 加密的基本步骤，使用的是逐字节处理的方式

```
.section .text
.syntax unified

.global aes_encrypt

aes_encrypt:
    push {r4, r5, r6, r7, lr}

    ldr r4, [r0]            @ r4 = plaintext[0]
```

```
    ldr r5, [r0, #4]        @ r5 = plaintext[4]
    ldr r6, [r1]              @ r6 = key[0]
    ldr r7, [r1, #4]        @ r7 = key[4]

    @ Perform AddRoundKey
    eor r4, r4, r6
    eor r5, r5, r7

    @ Perform SubBytes
    bl sub_bytes

    @ Perform ShiftRows
    bl shift_rows

    @ Perform MixColumns
    bl mix_columns

    @ Perform AddRoundKey
    ldr r6, [r1, #8]        @ r6 = key[8]
    ldr r7, [r1, #12]      @ r7 = key[12]
    eor r4, r4, r6
    eor r5, r5, r7

    @ Store the encrypted data
    str r4, [r2]
    str r5, [r2, #4]

    pop {r4, r5, r6, r7, pc}


sub_bytes:
    @ SubBytes lookup table
    ldr r3, =sbox

    @ SubBytes for r4
    ldrb r0, [r4]
    ldr r1, [r3, r0]
    mov r4, r1

    @ SubBytes for r5
    ldrb r0, [r5]
    ldr r1, [r3, r0]
    mov r5, r1
```

```
        bx lr


shift_rows:
        @ ShiftRows for r4
        lsr r0, r4, #8
        lsl r1, r4, #24
        orr r4, r0, r1

        @ ShiftRows for r5
        lsr r0, r5, #16
        lsl r1, r5, #16
        orr r5, r0, r1

        @ Rotate r5 to the right by 8 bits
        ror r5, r5, #8

        bx lr


mix_columns:
        @ MixColumns for r4
        push {r0-r3}
        mov r0, r4
        lsr r1, r4, #8
        lsl r2, r4, #16
        lsr r3, r4, #24
        eor r4, r0, r1
        eor r4, r4, r2
        eor r4, r4, r3
        eor r4, r4, lsr #31
        eors r4, r4
        pop {r0-r3}

        @ MixColumns for r5
        push {r0-r3}
        mov r0, r5
        lsl r1, r5, #8
        lsr r2, r5, #16
        lsl r3, r5, #24
        eor r5, r0, r1
        eor r5, r5, r2
        eor r5, r5, r3
        eor r5, r5, lsr #31
```

```
    eors r5, r5
    pop {r0-r3}

    bx lr


.section .data
.align 4
sbox:
    .byte 0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5
    .byte 0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76
    .byte 0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0
    .byte 0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0
    .byte 0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC
    .byte 0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15
    .byte 0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A
    .byte 0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75
    .byte 0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0
    .byte 0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84
    .byte 0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B
    .byte 0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF
    .byte 0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85
    .byte 0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8
    .byte 0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5
    .byte 0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2
    .byte 0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17
    .byte 0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19, 0x73
    .byte 0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88
    .byte 0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x0B, 0xDB
    .byte 0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C
    .byte 0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79
    .byte 0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9
    .byte 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08
    .byte 0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6
    .byte 0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A
    .byte 0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E
    .byte 0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E
    .byte 0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94
    .byte 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF
    .byte 0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68
    .byte 0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16
```

六. project9

AES 的软件实现
采用了 openssl1.1 的库配置

```
#include <iostream>
#include <openssl/aes.h>

int main() {
	AES_KEY aesKey;
	unsigned char key[] = "0123456789abcdef";
	unsigned char plainText[] = "This is a plaintext message.";
	unsigned char cipherText[AES_BLOCK_SIZE];

	AES_set_encrypt_key(key, 128, &aesKey);
	AES_encrypt(plainText, cipherText, &aesKey);

	std::cout << "Cipher Text: ";
	for (int i = 0; i < AES_BLOCK_SIZE; i++) {
		std::cout << std::hex << (int)cipherText[i];
	}
	std::cout << std::endl;

	return 0;
}
```

七. project10

ECDSA 是一种基于椭圆曲线密码学的数字签名算法。ECDSA 基于离散对数问题，它利用了椭圆曲线上的数论性质和难解问题来实现数字签名的生成和验证。与传统的 RSA 签名算法相比，ECDSA 具有相同的安全性，但却使用更短的密钥长度，提供了更高的性能和效率。
ECDSA 的原理可以简单概括为以下几个步骤：
密钥生成：首先选择一个椭圆曲线和一个生成点，然后生成一个私钥（随机数），并通过椭圆曲线乘法运算得到公钥。
签名生成：使用私钥对要签名的数据进行哈希计算，并通过一系列数学运算生成签名值。
签名验证：使用公钥和已知的签名值对接收到的数据进行验证，确认其完整性和真实性。
代码采用了 openssl1.1 的库配置

```
#include <iostream>
#include <string>
#include <openssl/ec.h>
#include <openssl/ecdsa.h>
#include <openssl/obj_mac.h>

using namespace std;

void generateECKeyPair(EC_KEY*& ecKey)
```

```
{
	EC_GROUP* ecGroup = EC_GROUP_new_by_curve_name(NID_secp256k1);
	if (ecGroup == nullptr) {
		cout << "Failed to create EC group." << endl;
		return;
	}

	ecKey = EC_KEY_new();
	if (ecKey == nullptr) {
		cout << "Failed to create EC key." << endl;
		EC_GROUP_free(ecGroup);
		return;
	}

	if (EC_KEY_set_group(ecKey, ecGroup) != 1) {
		cout << "Failed to set EC group for key." << endl;
		EC_KEY_free(ecKey);
		EC_GROUP_free(ecGroup);
		return;
	}

	if (EC_KEY_generate_key(ecKey) != 1) {
		cout << "Failed to generate EC key pair." << endl;
		EC_KEY_free(ecKey);
		EC_GROUP_free(ecGroup);
		return;
	}
}

string signData(const string& data, EC_KEY* privateKey)
{
	const unsigned char* msg = reinterpret_cast<const unsigned char*>(data.c_str());
	size_t msgLen = data.size();

	ECDSA_SIG* signature = ECDSA_do_sign(msg, static_cast<int>(msgLen), privateKey);
	if (signature == nullptr) {
		cout << "Failed to generate ECDSA signature." << endl;
		return "";
	}

	const BIGNUM* r = nullptr;
	const BIGNUM* s = nullptr;
	ECDSA_SIG_get0(signature, &r, &s);
```

```
    char* rHex = BN_bn2hex(r);
    char* sHex = BN_bn2hex(s);
    string signatureHex = string(rHex) + string(sHex);

    OPENSSL_free(rHex);
    OPENSSL_free(sHex);
    ECDSA_SIG_free(signature);

    return signatureHex;
}

bool verifySignature(const string& data, const string& signatureHex, EC_KEY* publicKey)
{
    const unsigned char* msg = reinterpret_cast<const unsigned char*>(data.c_str());
    size_t msgLen = data.size();

    BIGNUM* r = BN_new();
    BIGNUM* s = BN_new();
    if (BN_hex2bn(&r, signatureHex.substr(0, 64).c_str()) == 0 ||
        BN_hex2bn(&s, signatureHex.substr(64).c_str()) == 0) {
        BN_free(r);
        BN_free(s);
        return false;
    }

    ECDSA_SIG* signature = ECDSA_SIG_new();
    if (signature == nullptr) {
        BN_free(r);
        BN_free(s);
        return false;
    }

    ECDSA_SIG_set0(signature, r, s);

    int result = ECDSA_do_verify(msg, static_cast<int>(msgLen), signature, publicKey);
    ECDSA_SIG_free(signature);

    return result == 1;
}

int main()
{
    EC_KEY* privateKey = nullptr;
    EC_KEY* publicKey = nullptr;
```

```
    generateECKeyPair(privateKey);
    publicKey = EC_KEY_dup(privateKey);

    string data = "Hello, Ethereum!";
    string signature = signData(data, privateKey);

    bool isVerified = verifySignature(data, signature, publicKey);

    if (isVerified) {
        cout << "Signature is verified. Data is authentic." << endl;
    } else {
        cout << "Signature verification failed. Data may be tampered with or the wrong public
key." << endl;
    }

    EC_KEY_free(privateKey);
    EC_KEY_free(publicKey);

    return 0;
}
```