项目报告汇总

2023 年 8 月 4 日

1 环境介绍

操作系统：Windows 10
设备名称：LAPTOP-2ROFCV1G
处理器：AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz
机带 RAM 16.0 GB (15.9 GB 可用)
系统类型 64 位操作系统, 基于 x64 的处理器
软件环境：Visual Studio 2019、PyCharm Community Edition 2021.3.3

2 Project1

实现思路：

1. 首先，需要导入相应的库来计算 SM3 哈希函数和生成随机数。在这个示例中，我们将使用 gmssl 库。

2. 定义生成碰撞的函数：创建一个函数，用于生成哈希碰撞。在这个函数中，你可以使用一个无限循环，不断生成两个随机消息。你可以通过指定字节长度，使用随机库来生成随机字节。

3. 计算哈希值：使用 SM3 哈希函数计算每个消息的哈希值。你可以使用函数将消息转换为字节列表，并将其作为参数传递给此方法。它将返回一个哈希值。

4. 比较哈希值：将两个哈希值进行比较，如果它们相等，则找到了碰撞。

5. 输出结果：在找到碰撞时，输出这两个消息作为碰撞结果。

6. 循环迭代：如果没有找到碰撞，增加迭代次数，并在一定间隔后打印尝试次数。

7. 返回结果：返回找到的碰撞结果。

具体代码请见 project 文件。

3 Project2

实现思路：

这段代码是实现了 Rho 哈希算法的压缩函数。Rho 哈希算法是一种密码学哈希函数，用于将输入消息转换为固定长度的哈希值。

首先，定义了一些辅助函数，包括 ROTATE LEFT 函数用于循环左移位操作，以及 FF 和 GG 函数用于根据不同的条件计算中间结果。

接下来，定义了 rho compress 函数，该函数接受一个消息 M 作为输入。在函数中，初始化了 8 个 32 位的变量 A、B、C、D、E、F、G、H，并赋予了初始的值。

然后，对消息进行分组迭代处理。每次处理 64 个字节的数据块。首先将每个消息分组划分为 16 个 32 位字，并按照大端字节序转换为整数存储在列表 W 中。

接着，使用公式对 W16 到 W67 进行填充，并计算相应的值。

然后，进行 64 轮的迭代计算。根据公式，计算 SS1、SS2、TT1 和 TT2 的值，并更新 A、B、C、D、E、F、G、H 的值。

最后，在处理完所有消息分组后，将 A、B、C、D、E、F、G、H 进行进一步的处理，通过按位异或和循环左移操作，得到最终的哈希值。

测试代码部分，定义了一个消息”abc”，并对消息进行填充和长度处理。然后调用 rho compress 函数计算哈希值，并打印输出。

4 Project3

生日攻击（Birthday Attack）是一种密码学攻击技术，用于找到具有相同哈希值的两个不同输入。它基于生日悖论（Birthday Paradox），该悖论指出在一个集合中选择足够多的元素时，存在很高的概率两个元素会有相同的值。

在密码学中，哈希函数被广泛用于将任意长度的输入转换为固定长度

的哈希值，以实现数据完整性校验、数字签名等功能。然而，由于哈希函数的输出空间比输入空间要小得多，生日攻击利用了这一点。

生日攻击的基本思想是通过构造大量的输入消息，并计算它们的哈希值，以寻找碰撞，即具有相同哈希值的两个不同输入。攻击者可以利用生日悖论，在较小的输入空间内生成大量的输入，并使用快速的算法来搜索相同的哈希值。

具体的骤如下：

1. 攻击者选择一组输入消息进行哈希计算。

2. 对于每个输入消息，将其哈希值与之前计算的哈希值进行比较。

3. 如果发现两个不同的输入消息具有相同的哈希值，攻击成功，即找到了碰撞。

4. 如果没有找到碰撞，则继续增加输入消息的数量，重复步骤 1 和步骤 2，直到找到碰撞为止。

生日攻击的成功概率随着输入空间的大小和哈希值长度的增加而增加。根据生日悖论，只需要约 sqrt(N) 个输入消息就有很高的概率会出现碰撞，其中 N 是输入空间的大小。

具体代码请见 project 文件。

5 Project4

使用了 unrolling 和并行，将函数加速到了可以接受的速度。

图 1: 运行结果

具体代码请见 project 文件。

6 Project5

主要包含以下几个函数：

1. computeHash：计算给定输入字符串的 SHA256 哈希值。2.buildMerkleTree：递归构建 Merkle 树。先构建叶子节点，再构建中间节点，直到只剩下根节点。3. generateAuditPath：生成给定叶子索引的审计路径。从叶子节点开始向上，每层选择相邻的兄弟节点的哈希值。4. verifyAuditPath：验证给定叶子数据、Merkle 根和审计路径是否一致。

在 main 函数中，首先定义了一组数据作为叶子节点，然后调用 buildMerkleTree 函数构建 Merkle 树。接着打印出构建的 Merkle 树的每个节点的值和是否是叶子节点。

然后通过调用 generateAuditPath 函数生成叶子索引为 1 的审计路径，并将其打印出来。

最后，调用 verifyAuditPath 函数验证叶子索引为 1 的数据和生成的审计路径是否与 Merkle 根一致，并输出验证结果。

具体代码请见 project 文件。

7 Project6

我们使用 C++ 和 Socket 库实现了在服务端和客户端之间实现基于文本的简单通信协议。配合上方 Project5 的内容可以实现通信交互。

具体代码请见 project 文件。

8 Project7

Project5 与 Project6 配合完成。

9 Project8

使用了 ARM NEON 提供的 SIMD（Single Instruction Multiple Data）指令来进行并行计算，从而提高加密算法的执行速度和效率。

aes encrypt 函数接受输入的明文数据数组 plaintext、密钥数组 key，并将加密后的密文存储到输出数组 ciphertext 中。

首先，使用 vld1q u8 指令将输入的明文数据和密钥分别加载到 128 位的 NEON 寄存器 v plaintext 和 v key 中。

然后，进行轮密钥扩展，将初始轮密钥 v key 保存在 v roundkey 中。

接下来，执行 AES 加密算法的多轮操作。每一轮包括字节替代、行移位、列混合和轮密钥加四个步骤。

在每轮加密中，首先使用 vaesSubq u8 指令对输入的 v plaintext 进行字节替代操作。然后，使用 vaesimcq u8 指令进行行移位操作。接着，使用 vaesdq u8 指令对 v plaintext 和当前轮密钥 v roundkey 进行列混合操作。最后，通过使用 veorq u8 指令将结果与当前轮密钥进行轮密钥加操作。

在所有轮完成之后，执行最后一轮字节替代和轮密钥加操作。

最后，使用 vst1q u8 指令将加密后的结果存储到输出缓冲区 ciphertext 中。

不过由于环境配置原因，代码一直没能正常运行。

具体代码请见 project 文件。

10 Project9

AES（Advanced Encryption Standard）是一种对称密钥加密算法，它广泛应用于数据保密领域。AES 算法可以分为两个步骤：密钥扩展和加密轮。

1. 密钥扩展：- 首先，将输入的密钥按字节分割成多个字。- 接着，根据密钥长度选择适当的循环次数，对密钥进行轮密钥生成。- 每一轮密钥生成包括以下操作：- 字节替代：使用 S 盒将每个字节替换为一个新的字节。- 行移位：按照指定规则对每一行进行循环左移。- 列混合：使用特定的矩阵乘法对每一列进行混合。- 轮密钥加：将每一轮生成的密钥与明文进行异或操作。

2. 加密轮：- 首先，将输入的明文按字节分割成多个字。- 接着，执行多轮的加密处理。- 每一轮加密包括以下操作：- 字节替代：使用 S 盒将每个字节替换为一个新的字节。- 行移位：按照指定规则对每一行进行循环左移。- 列混合：使用特定的矩阵乘法对每一列进行混合。- 轮密钥加：将当前轮生成的密钥与明文进行异或操作。

最后一轮加密之后，得到的密文就是加密后的结果。

SM4 是中国自主设计的分组密码算法，也被称为公安部密码局标准算法。它是对称密钥算法，用于加密和解密数据。

SM4 算法的特点如下：- 分组长度为 128 位，密钥长度为 128 位。- 使用 32 轮迭代结构，每轮包括字节替代、行移位、列混淆和轮密钥加四个步骤。- 每轮的轮密钥由主密钥经过密钥扩展生成。- 算法设计注重安全性和效率，在满足各种攻击模型的前提下提供高强度的数据保护。

SM4 算法的加密过程如下：1. 将输入的明文按照 128 位进行分组。2. 进行 32 轮的迭代操作，每轮包括以下步骤：- 字节替代：使用 S 盒进行字节替代操作。- 行移位：按照指定规则对每一行进行循环左移操作。- 列混淆：使用特定的矩阵乘法对每一列进行混淆操作。- 轮密钥加：将当前轮的轮密钥与明文进行异或操作。3. 最后一轮迭代之后，得到加密后的密文。

SM4 算法的解密过程与加密过程相反，使用相同的密钥和相同的迭代步骤进行逆向操作，将密文解密为原始明文。

具体代码请见 project 文件。

11 Project10

具体 report 内容请见 project 文件。

12 Project11

RFC 6979 是一项标准，定义了 SM2 密码算法与 RFC 6979 结合使用的方法。这种方法用于生成 SM2 数字签名算法中的确定性（deterministic）参数，以提高算法的安全性。

在传统的非确定性数字签名算法中，随机数的生成是一个关键步骤。然而，在使用非确定性随机数生成器时存在一定的安全性风险。RFC 6979 引入了一种使用伪随机函数（PRF）和消息哈希函数来确定性地生成随机数的方法，从而解决了传统随机数生成的安全性问题。

使用 RFC 6979 生成 SM2 数字签名的过程如下：1. 首先，将待签名的消息进行哈希计算，得到消息摘要。2. 然后，使用 RFC 6979 定义的伪随机函数，利用私钥、消息摘要和附加信息（可选）生成一个确定性的随机数 k。3. 接下来，根据 SM2 的签名算法，使用生成的随机数 k 和私钥 d 计算

签名的两个参数 r 和 s。4. 最后，将 r 和 s 作为数字签名输出。

具体代码请见 project 文件。

13 Project12

代码构成：

函数 verifyAES 用于验证 AES 加密和解密的正确性。它接受明文（plaintext）、密文（ciphertext）、密钥（key）和初始化向量（iv）作为参数。

在函数中，首先创建了一个 CBC Mode 的 Encryption 对象，并使用 SetKeyWithIV 函数设置密钥和初始化向量。这里使用的是 AES 算法和 CBC 模式。

然后，通过 StringSource 和 StreamTransformationFilter 将明文进行加密，将结果保存在变量 encryptedtext 中。

接着，如果得到的密文与输入的密文相同，输出"Encryption is correct."；否则，输出"Encryption is incorrect!"，并打印预期的密文和实际的密文。

然后，创建一个 CBC Mode 的 Decryption 对象，并使用 SetKeyWithIV 函数设置相同的密钥和初始化向量。

同样地，通过 StringSource 和 StreamTransformationFilter 将密文进行解密，将结果保存在变量 decryptedtext 中。

最后，如果解密得到的明文与输入的明文相同，输出"Decryption is correct."；否则，输出"Decryption is incorrect!"，并打印预期的明文和实际的明文。

在 main 函数中，定义了明文、密文、密钥和初始化向量，并调用 verifyAES 函数进行验证。

具体代码请见 project 文件。

14 Project13

实现 ECMH（Elliptic Curve Menezes-Qu-Vanstone Hybrid）方案涉及到椭圆曲线密码学的各个方面，包括密钥生成、加密、解密和签名等。

首先，我们需要引入所需的模块，然后，我们定义 ECMH 方案的关键步骤：

生成 ECMH 密钥对：生成椭圆曲线的私钥和公钥。

对称加密和解密：使用 AES 对称加密算法对消息进行加密和解密。

椭圆曲线非对称加密和解密：使用椭圆曲线公钥和私钥对消息进行加密和解密。

数字签名和验证：使用椭圆曲线私钥和公钥对消息进行签名和验证。

具体代码请见 project 文件。

15 Project14

实现 PGP（Pretty Good Privacy）方案：

1. 生成密钥对：使用 SM2 曲线参数生成一个 SM2 椭圆曲线密钥对，包括一个私钥和对应的公钥。

2. 对称加密：选择一种对称加密算法（如 AES），生成对称密钥。使用该对称密钥对要加密的消息进行加密，并返回加密后的密文。

3. 非对称加密：使用接收者的公钥对对称密钥进行加密，得到加密后的对称密钥密文。

4. 签名：使用发送者的私钥对消息进行签名。首先，对消息进行哈希处理（如 SHA-256），然后使用私钥对哈希值进行签名，并返回签名结果。

5. 封装数据包：将加密后的密文、加密后的对称密钥密文和签名结果封装成一个数据包。

6. 解封数据包：接收者首先使用自己的私钥对接收到的对称密钥密文进行解密，获得对称密钥。然后使用对称密钥对密文进行解密，还原出原始的消息。最后，使用发送者的公钥对接收到的签名结果进行验证，确保消息的完整性和真实性。

具体代码请见 project 文件。

16 Project15、16

与上方的 Project6 基本一致，不过使用的是较为便捷的 python 进行 Socket。

下面是关于 SM2 的一些简要说明：

密钥生成：SM2 算法使用椭圆曲线生成密钥对。通常，一个随机数作为私钥，通过椭圆曲线上的运算可以计算出对应的公钥。

加密算法：SM2 使用非对称加密算法进行加密。发送方使用接收方的公钥对消息进行加密，得到加密后的密文。

解密算法：接收方使用自己的私钥对收到的密文进行解密，还原出原始的消息。

数字签名：SM2 也提供了数字签名算法。发送方使用自己的私钥对消息进行签名，生成签名结果。接收方使用发送方的公钥对签名结果进行验证，确保消息的完整性和真实性。

密钥交换：SM2 也可以用于密钥交换协议，使双方可以安全地协商一个共享的对称密钥，用于后续的对称加密通信。

具体代码请见 project 文件。

17 Project17

从 github 获取的信息，从本机需寻找源代码没有成功。

具体 report 请见 project 文件。

18 Project18

Python 脚本使用了 bitcoinrpc 库来连接比特币测试网络的 RPC 接口，并实现了发送交易和解析交易数据的功能。

首先，脚本通过设置 rpc user、rpc password、rpc host 和 rpc port 来配置 RPC 连接。然后使用 AuthServiceProxy 创建了一个 RPC 连接对象 rpc connection。

send transaction() 函数实现了发送交易的功能。它首先通过 rpc connection.getnewaddress() 创建一个发送方地址，并使用 rpc connection.generatetoaddress() 生成一些测试用的比特币到该地址。接着，使用 rpc connection.getnewaddress() 创建一个接收方地址。

构建交易输入时，使用 rpc connection.listunspent() 获取发送方地址的未花费输出列表（UTXO），并将其转化为 inputs 数组。构建交易输出时，将接收方地址和要发送的比特币数量设置为 outputs 字典。

然后，使用 rpc connection.createrawtransaction() 构建原始交易数据，并使用 rpc connection.signrawtransactionwithwallet() 对交易进行签名。最后，使用 rpc connection.sendrawtransaction() 广播已签名的交易，并打印

出交易的 TXID。

decode transaction(txid) 函数实现了解析交易数据的功能。它使用 rpc connection.getrawtransaction() 获取特定 TXID 的交易信息，并根据该交易的 vin 和 vout 字段解析交易的输入和输出。计算出总的输入金额和总的输出金额，并计算出交易费用。最后，打印交易的详细信息。

根据比特币节点的配置，设置 rpc user、rpc password、rpc host 和 rpc port。并将 sender address 和 receiver ddress 设置为实际的比特币地址。根据需要修改 outputs 来指定实际发送的比特币数量。调用 send transaction() 函数发送交易，调用 decode transaction(txid) 函数解析特定交易的详细信息时，将 txid 替换为要解析的交易的 TXID。

具体代码请见 project 文件。

19 Project19

编写了一个生成比特币地址和进行数字签名的程序。

要假装自己是中本聪，可以修改以下部分：

1. 修改公钥：将 ‘std::string publicKey‘中的值替换为任何你想要假装拥有的公钥。请注意，这将影响生成的比特币地址。

2. 修改私钥：将 ‘std::string privateKey‘中的值替换为与所选公钥匹配的私钥。

3. 修改签名数据：将 ‘std::string data‘中的值替换为你想要进行数字签名的数据。可以使用任何字符串。

在修改完以上代码后，运行程序即可生成相应的比特币地址和数字签名。

具体代码请见 project 文件。

20 Project20

布置重复无需作业。

21 Project21

Schnorr-Batch 签名方案是一种基于 Schnorr 签名的批量签名方案，它允许同时对多个消息进行签名，并生成相应的聚合签名。此方案的优点是在批量签名过程中实现高效的计算和通信。由于使用了聚合签名，相比于逐个独立签名，可以大幅减少签名的大小以及验证的开销。此外，该方案还具有简洁、安全和隐私性高的特点，因此被广泛应用于多方协作密钥生成、区块链交易等领域。下面是其具体原理：

1. 密钥生成：选择一个大素数 p 和一个生成元 g，确保 p 是一个安全的素数。

随机选择一个私钥 x，范围在 [1, p-1] 之间，并计算对应的公钥 Y = g^{x} mod p。

2. 签名生成：对于每个要签名的消息 m i，执行以下步骤：随机选择一个挑战数 k i，范围在 [1, p-1] 之间。计算 R i = g^{k} i mod p。计算挑战哈希值 e i = Hash(m i || R i)。计算响应值 s i = (k i + x * e i) mod (p-1)。聚合所有的响应值 s i，得到聚合的响应 S = sum(s i) mod (p-1)。

3. 聚合签名输出：使用第一个签名中的 R 1 和聚合的响应 S 作为聚合签名的输出。

4. 验证：对于每个签名的消息-响应对 (m i, s i)，执行以下步骤：计算挑战哈希值 e i = Hash(m i || R i)。计算验证值 v i = (g$^{(}$s i) * Y$^{(}$e i)) mod p。如果所有的验证值 v i 都等于对应的 R i，则验证通过；否则，验证失败。

具体代码详见 py 文件

22 Project22

具体 report 请见 project 文件。