山东大学网络空间安全学院

创新创业实践 课程实验报告

|  |  |  |
| --- | --- | --- |
| 学号：202100460047 | 姓名：罗梓珅 | 班级：网安21.2 |
| 实验日期：2023.7.12 | | |
| 实验环境：11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz | | |
| 实验步骤与内容：  Project1: implement the naïve birthday attack of reduced SM3  实施简化的SM3生日攻击是一种密码学攻击方法，用于对SM3哈希算法进行测试。  生日攻击是一种针对哈希函数的攻击方式，旨在找到两个不同的输入值，但它们的哈希值相同。这个概念类似于在一群人中找到两个人生日相同的情况，因此称为生日攻击。  简化的SM3生日攻击的步骤如下：  选择一个适当的消息空间大小，例如2^k个不同的消息。  随机选择并计算一组消息的哈希值，直到找到两个哈希值相同的消息。  如果找到了相同的哈希值，即找到了碰撞（collision），这意味着两个不同的消息具有相同的哈希值。  输出找到的碰撞，作为成功实施生日攻击的结果。  代码实现：  import random  import time  from cryptography.hazmat.primitives import hashes  from cryptography.hazmat.backends import default\_backend  def naive\_birthday\_attack():      k = 16  # 消息空间大小，选择适当的值      hash\_values = {}  # 存储已计算的哈希值        while True:          message = generate\_random\_message(k)  # 生成随机消息          start\_time = time.time()  # 记录开始时间          hash\_value = calculate\_sm3\_hash(message.encode())  # 计算哈希值          end\_time = time.time()  # 记录结束时间            if hash\_value in hash\_values:              collision\_message = hash\_values[hash\_value]              print("Collision found!")              print("Message 1:", collision\_message)              print("Message 2:", message)              print("Time elapsed:", end\_time - start\_time, "seconds")              return collision\_message, message            hash\_values[hash\_value] = message  def calculate\_sm3\_hash(message):      sm3\_hash = hashes.Hash(hashes.SM3(), backend=default\_backend())      sm3\_hash.update(message)      digest = sm3\_hash.finalize()      return digest.hex()  def generate\_random\_message(k):      # 生成随机消息，长度为k位      message = ''.join(random.choices(['0', '1'], k=k))      return message  # 执行生日攻击  collision\_message, message = naive\_birthday\_attack()  输出结果    Project2: implement the Rho method of reduced SM3  Rho方法是一种用于简化SM3密码哈希算法的技术。Rho方法通过引入迭代过程来减少SM3算法中的计算量。它基于一个观察到的现象：在SM3算法中，消息扩展阶段是最耗时的部分之一。因此，可以通过减少消息扩展的次数来提高算法的效率。  具体而言，Rho方法将消息分为多个块，并在每个块上执行SM3算法的初始压缩函数。然后，它使用这些初始压缩函数的输出作为下一个块的输入，以此类推。这样，Rho方法避免了对每个块都进行完整的消息扩展，从而减少了计算量。  代码实现：  import hashlib  def rho\_sm3(message):      # 将消息分块      blocks = [message[i:i+64] for i in range(0, len(message), 64)]        # 初始化初始压缩函数的输入      h = bytearray.fromhex("7380166f4914b2b972fb8951a8274c60"                            "986b5fc0e1e25b88c4dd3a367b0e2e8c")        # 对每个块执行初始压缩函数      for block in blocks:          # 将当前块与初始压缩函数的输入进行异或运算          x = bytearray(len(h))          for i in range(min(len(h), len(block))):           x[i] = h[i] ^ block[i]            # 执行SM3算法的初始压缩函数          h = hashlib.new('sm3', x).digest()        return h.hex()  # 测试示例  message = b'Hello, World!'  hash\_value = rho\_sm3(message)  print("Hash value:", hash\_value)  输出结果  FW7(4TG%7$JWGKW7NK2HW%D  Project3: implement length extension attack for SM3, SHA256, etc  要施SM3的长度扩展攻击，假设我们已经获得了一个SM3哈希值H和相应的消息M攻击者可以利用这个信息来生成一个新的消息M'使得M' || P（其中P是攻者控制的附数据）的哈希值与H相同。具体骤如下：  首先，根据已知的哈希值H和消息M，算出中间状态IV（初始量）。  然，根据SM3的缩函数和填充规则，构造一个虚拟的消息块，消息块包含P以及填充和长度信息。  将虚拟消息块与已知的中状态IV一起输入SM3压缩函数，到新的哈希值H'。  最后，将P附加到原消息M之后，即可得到具有相同哈希值H的新消息M'。  对于SHA256哈希函数的长度扩攻击，方法类似。SHA256也是一种常用的密码杂凑算法，其结构和工作原理SM3类似。攻击者可以利用已知的哈希值H和消息M来生成具有同哈希值但不同输入的数据。具步骤如下：  根据已知的哈希值H和消息M，计算出中状态IV。  构造一个虚拟的消息块，该消息块包含P以及填和长度信息。  将虚拟消息块与已知的中间状态IV一起输入SHA256压函数，得到新的希值H'。  将P附加到原消息M之后，即可得到具有相同希值H的新消息M'。  SM3的无法编写成功，我最终测试了SHA256的  代码实现  import hashlib  def length\_extension\_attack(hash\_func, original\_message, original\_hash, extension\_data):      # 获取原始消息的长度      original\_length = len(original\_message)      # 构造填充数据      padding = b'\x80' + b'\x00' \* ((64 - (original\_length + 8) % 64) % 64)      length\_bits = (original\_length + len(padding) + 8) \* 8      padding += length\_bits.to\_bytes(8, 'big')      # 计算新消息的哈希值      hash\_obj = hash\_func()      hash\_obj.update(extension\_data + padding)      new\_hash = hash\_obj.digest()      return new\_hash.hex()  # 示例使用SHA256哈希函数进行长度扩展攻击  original\_message = b'Original Message'  extension\_data = b'Extension Data'  # 计算原始消息的哈希值  original\_hash = hashlib.sha256(original\_message).digest()  # 进行长度扩展攻击  new\_hash = length\_extension\_attack(hashlib.sha256, original\_message, original\_hash, extension\_data)  print("Original Hash:", original\_hash.hex())  print("New Hash:", new\_hash)  输出结果  IMG_256  **Project4: do your best to optimize SM3 implementation (software)**  我主要使用了循环展开来优化（数据对齐怎么都写不对，放弃了）  import struct  import numpy as np  import time  def rotate\_left(x, n):      return ((x << n) | (x >> (32 - n))) & 0xFFFFFFFF  def padding(message):      ml = len(message) \* 8      message += b'\x80'      message += b'\x00' \* ((56 - len(message) % 64) % 64)      message += struct.pack('>Q', ml)      return message  def sm3(message):      message = padding(message)      h = [0x7380166F, 0x4914B2B9, 0x172442D7, 0xDA8A0600,           0xA96F30BC, 0x163138AA, 0xE38DEE4D, 0xB0FB0E4E]      for i in range(0, len(message), 64):          w = list(struct.unpack('>16I', message[i:i+64]))          # 循环展开          for j in range(16, 64):              w.append(   rotate\_left(w[j-16] ^ w[j-9] ^ (rotate\_left(w[j-3], 15)), 1) ^ (rotate\_left(w[j-13], 7)) ^ w[j-6]              )              w.append(   rotate\_left(w[j-15] ^ w[j-8] ^ (rotate\_left(w[j-2], 15)), 1) ^ (rotate\_left(w[j-12], 7)) ^ w[j-5]              )              w.append(   rotate\_left(w[j-14] ^ w[j-7] ^ (rotate\_left(w[j-1], 15)), 1) ^ (rotate\_left(w[j-11], 7)) ^ w[j-4]              )              w.append(   rotate\_left(w[j-13] ^ w[j-6] ^ (rotate\_left(w[j], 15)), 1) ^ (rotate\_left(w[j-10], 7)) ^ w[j-3]              )      return ''.join([format(x, '08x') for x in h])  # 测试  message = b'Hello, world!'  start\_time = time.time()  hash\_value = sm3(message)  end\_time = time.time()  print(hash\_value)  print("Time elapsed:", end\_time - start\_time, "seconds")  输出结果  IMG_256  还是挺快的  **Project5: Impl Merkle Tree following RFC6962**  Merkle Tree是一棵二叉树，其中个叶节点表示一个数据元素，例如证书或哈希值。树是通过递归地对节点对进行哈希运算构建的，直到获得单根节点。这个根节点被称为Merkle根，它作为整个数据集的紧凑表示。  RFC 6962引了与CT相关的Merkle Tree的几关键概念和操作：  一致性：Merkle Tree用于确保日志随时间的一致性。通过比较两个不同快照Merkle根，可以测到任何经授权的修改或篡改。  包含明：包含明允许客户端证明特定的数据元素包含在Merkle Tree中，而无需透露整个树。它由沿着从叶节点到根节点的路径的一组中间节点和它们的兄弟节点组成。  Merkle Tree遍历RFC 6962定义使用称为紧凑Merkle Tree（Compact Merkle Tree，CMT）的紧凑表示形式高效遍历Merkle Tree的过程CMT允许客端以最小的计开销导航树并生成证。  Python代码跑不动，cpp的代码我又不太会写，OpenSSL库也难以装上  Python代码如下：  import hashlib  class MerkleTree:      def \_\_init\_\_(self, leaves):          self.leaves = leaves          self.tree = self.build\_tree()      def build\_tree(self):          if len(self.leaves) == 0:              return []          tree = [leaf for leaf in self.leaves]          while len(tree) > 1:              tree\_level = []              for i in range(0, len(tree), 2):                  left\_child = tree[i]                  right\_child = tree[i + 1] if i + 1 < len(tree) else tree[i]          parent\_hash = self.hash\_children(left\_child, right\_child)          tree\_level.append(parent\_hash)          tree = tree\_level          return tree[0]  def hash\_children(self, left\_child, right\_child):          combined\_data = left\_child + right\_child          combined\_hash = hashlib.sha256(combined\_data.encode()).hexdigest()          return combined\_hash  # test  leaves = ["A", "B", "C", "D"]  merkle\_tree = MerkleTree(leaves)  root\_hash = merkle\_tree.tree  print("Root Hash:", root\_hash)  跑不了一点，实在没办法  **Project6: impl this protocol with actual network communication**  此处指的是Range Proof With Hash Function  具体步骤如下：  输入：存在一个需要验证的值v，以及一个表示范围的上限和下限（例如L和H）。  将v与特定的随机数（称为“挑战”）结合，并进行哈希运算，得到一个哈希值h。  将哈希值h与上限和下限进行比较，验证是否满足条件（即L ≤ h ≤ H）。  如果条件成立，则范围证明有效，表示值v在指定的范围内。  代码实现  import hashlib  import random  def range\_proof(min\_value, max\_value):      #在范围内生成一个机数      r = random.randint(min\_value, max\_value)      #计算哈希值      hasi = hashlib.sha256(str(r).encode()).hexdigest()      print(hasi)      #将数据发送给验证者      return r, hasi  def verify\_range\_proof(min\_value, max\_value, r, h):      #重新计算哈希值      h\_prime = hashlib.sha256(str(r).encode()).hexdigest()      print(h\_prime)      # tset      if min\_value <= r <= max\_value and h == h\_prime:          return True      else:          return False  # 例子  min\_value = 0  max\_value = 100  r, h = range\_proof(min\_value, max\_value)  is\_valid = verify\_range\_proof(min\_value, max\_value, r, h)  print("范围有效:", is\_valid)  输出结果  _05f94ac9450d6de9574f3ad2e23c0d76_856068367_mmexport1689820037746  **Project7: Try to Implement this scheme：**  这里指的是  Generalizing Hash Chains: Putting All Things Together  GeneralizedHashChain 类表示哈希链，它具有一个列表 chain 用于存储哈希元素。  add\_element 方法用于将元素添加到链中。如果链为空，则将元素本身的哈希附加到链末尾。否则，通过将前一个哈希与当前元素连接起来，并使用 SHA-256 对结果进行哈希运算，得到哈希值，然后将其附加到链末尾。  get\_last\_hash 方法返回链中的最后一个哈希。  verify\_chain 方法检查链的完整性。它遍历链中的每个元素，并验证每个计算出的哈希是否与链中相应的元素匹配。  import hashlib  class GeneralizedHashChain:      def \_\_init\_\_(self):          self.chain = []      def add\_element(self, element):          if len(self.chain) == 0:              self.chain.append(element)          else:                previous\_hash = self.get\_last\_hash()              new\_hash = hashlib.sha256((previous\_hash + element).encode()).hexdigest()              self.chain.append(new\_hash)      def get\_last\_hash(self):          return self.chain[-1]        def verify\_chain(self):          for i in range(1, len(self.chain)):              previous\_hash = self.chain[i-1]              current\_element = self.chain[i]              computed\_hash = hashlib.sha256((previous\_hash + current\_element).encode()).hexdigest()              if computed\_hash != current\_element:                  return False          return True  # 示例用法  chain = GeneralizedHashChain()  chain.add\_element("test1")  chain.add\_element("test2")  chain.add\_element("test3")  print("hash链", chain.chain)  print("最终hash值:", chain.get\_last\_hash())  print("正确性", chain.verify\_chain())  很可惜的是，这段代码实在有问题，始终无法正确实现，得到的验证结果一直是false    **Project9: AES / SM4 software implementation**  AES和sm4的原理我就不赘述了，直接上代码  Aes  from Crypto.Cipher import AES  from Crypto.Util.Padding import pad, unpad  import base64  def aes\_jiami(mingwen, key):      cipher = AES.new(key, AES.MODE\_ECB)      padded\_ming = pad(mingwen.encode(), AES.block\_size)      miwen\_result = cipher.encrypt(padded\_ming)      return base64.b64encode(miwen\_result).decode()  def aes\_jiemi(miwen, key):      cipher = AES.new(key, AES.MODE\_ECB)      miwen\_de = base64.b64decode(miwen)      jiemi\_ = cipher.decrypt(miwen\_de)      result\_ = unpad(jiemi\_, AES.block\_size)      return result\_.decode()  # 测试  key = b'Sixteen byte key'  mingwen = 'SDU'  miwen = aes\_jiami(mingwen, key)  result = aes\_jiemi(miwen, key)  print('明文为：', mingwen)  print('密文为：', miwen)  print('解密得到的：', result)    **Sm4**  from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes  from cryptography.hazmat.primitives import padding  import binascii  key = binascii.unhexlify('0123456789abcdeffedcba9876543210')  mingwen = b"SDU GAI"  cipher = Cipher(algorithms.SM4(key), modes.ECB())  encryptor = cipher.encryptor()  #填充  padder = padding.PKCS7(128).padder()  padded\_result = padder.update(mingwen) + padder.finalize()  #加密  miwen = encryptor.update(padded\_result) + encryptor.finalize()  #解密并还原  decryptor = cipher.decryptor()  padded\_result = decryptor.update(miwen) + decryptor.finalize()  unpadder = padding.PKCS7(128).unpadder()  result = unpadder.update(padded\_result) + unpadder.finalize()  print('明文：', mingwen)  print('密文：', miwen.hex())  print('解密结果：', result)    符合预期  **Project10: report on the application of this deduce technique in Ethereum with ECDSA**  ECDSA是一种基于椭圆曲线数学原理的数字签名算法。它使用一个椭圆曲线公钥和私钥对数据进行签名和验证。在以太坊中，ECDSA广泛应用于交易和智能合约的验证过程。  当用户在以太坊上发送交易时，需要对交易进行签名，以证明该交易是由有效的发送者发起的。发送者使用其私钥对交易信息进行签名，并附加到交易中。接收者可以使用发送者的公钥和签名来验证交易的真实性和完整性。  ECDSA在以太坊中还用于智能合约的验证。智能合约是在以太坊区块链上执行的自动化合约，其中包含了一些条件和操作。当用户与智能合约交互时，会涉及对交易和合约状态的验证。ECDSA可用于验证智能合约的调用和执行，并确保只有授权的用户可以访问和修改合约的状态。  总结来说，ECDSA在以太坊中被广泛应用于交易和智能合约的验证过程。它确保交易的真实性和完整性，并且只允许授权用户对智能合约进行操作。通过使用ECDSA，以太坊实现了安全和可靠的数字签名机制，为用户提供了信任和保障。  （以上为查询结果） | | |