学号 专业 信息安全 姓名

实验日期 **2024.12.10** 教师签字 成绩

实验报告

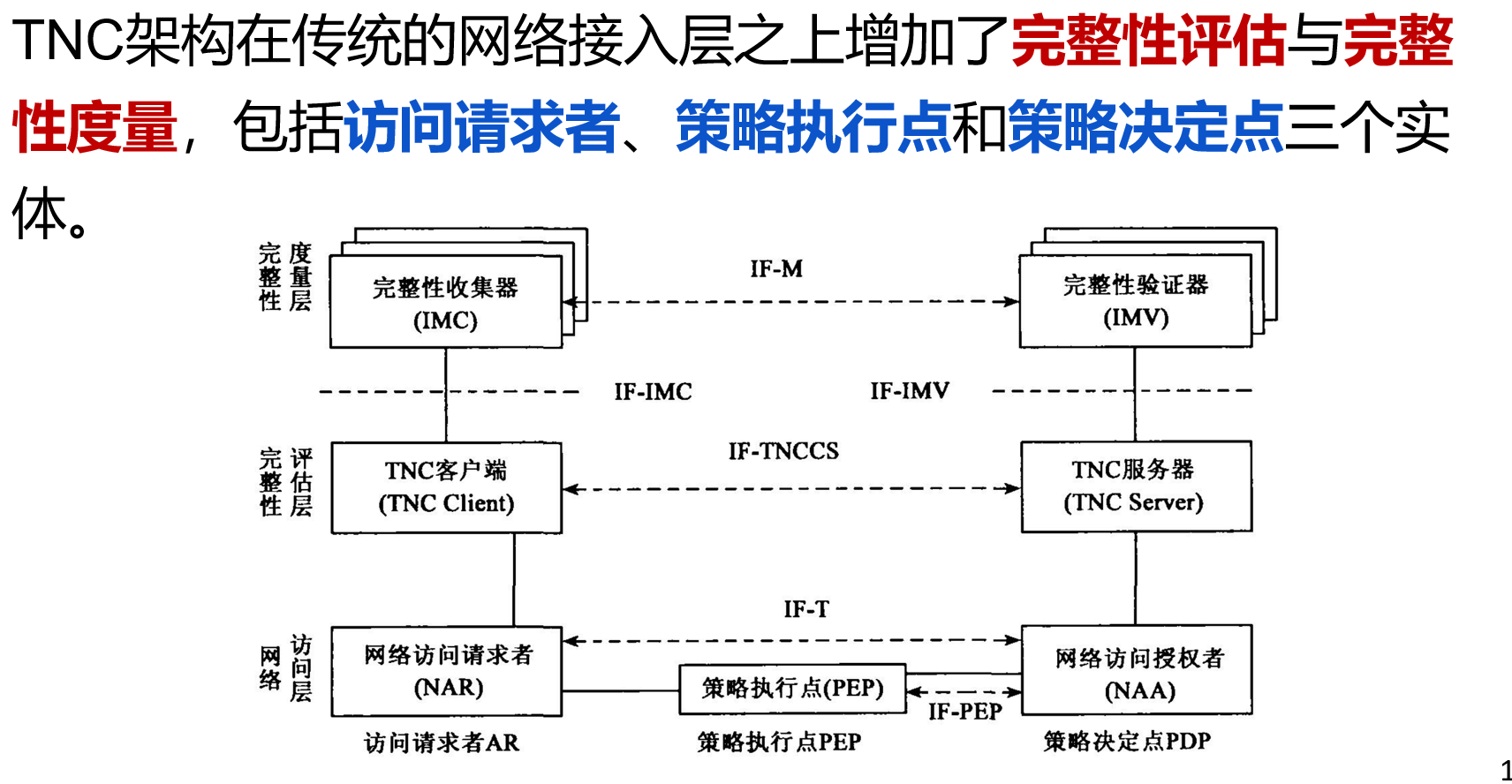
【实验名称】 可信计算第四次实验-版本2

【实验目的】

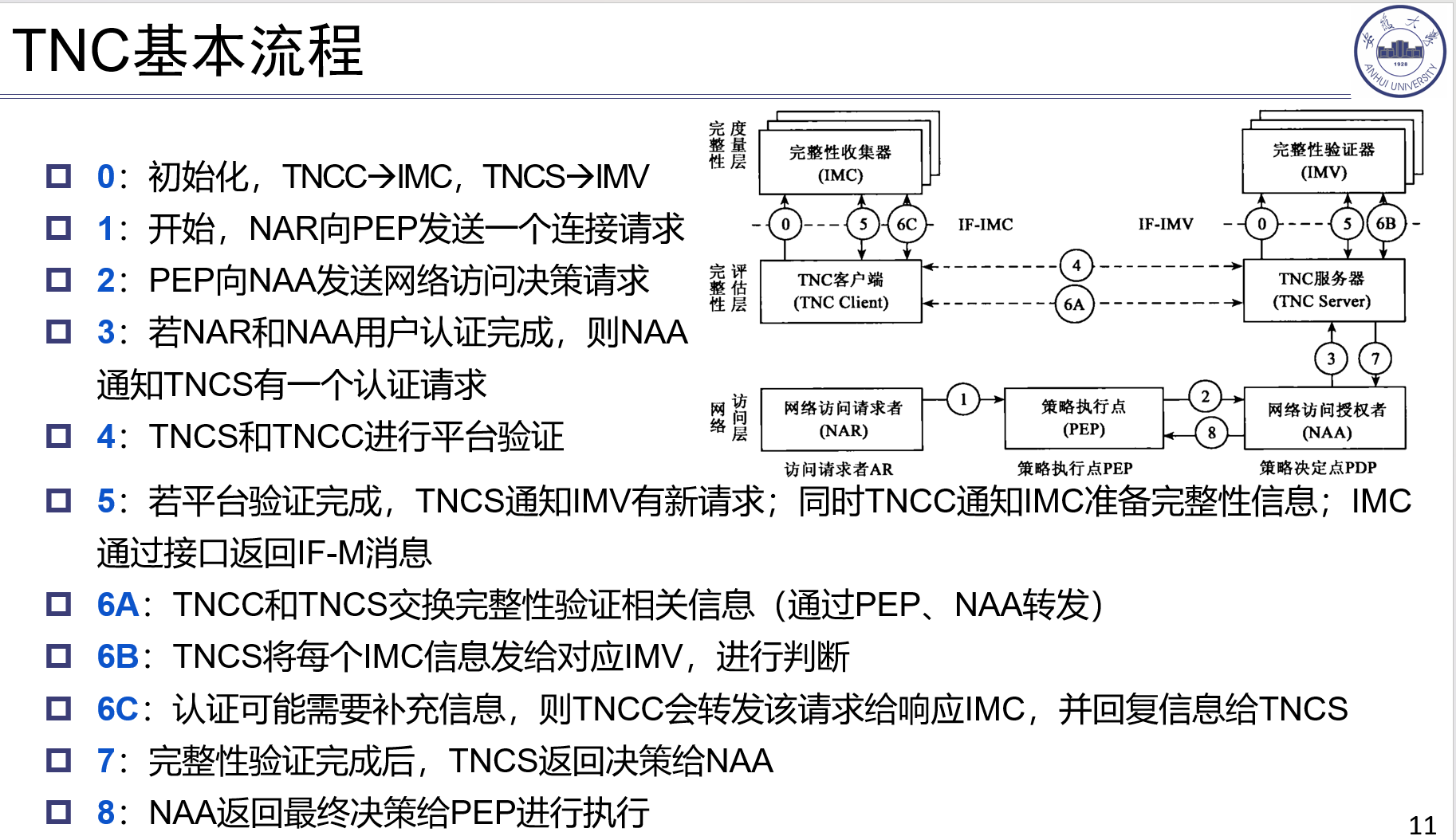
**模拟可信网络连接架构**

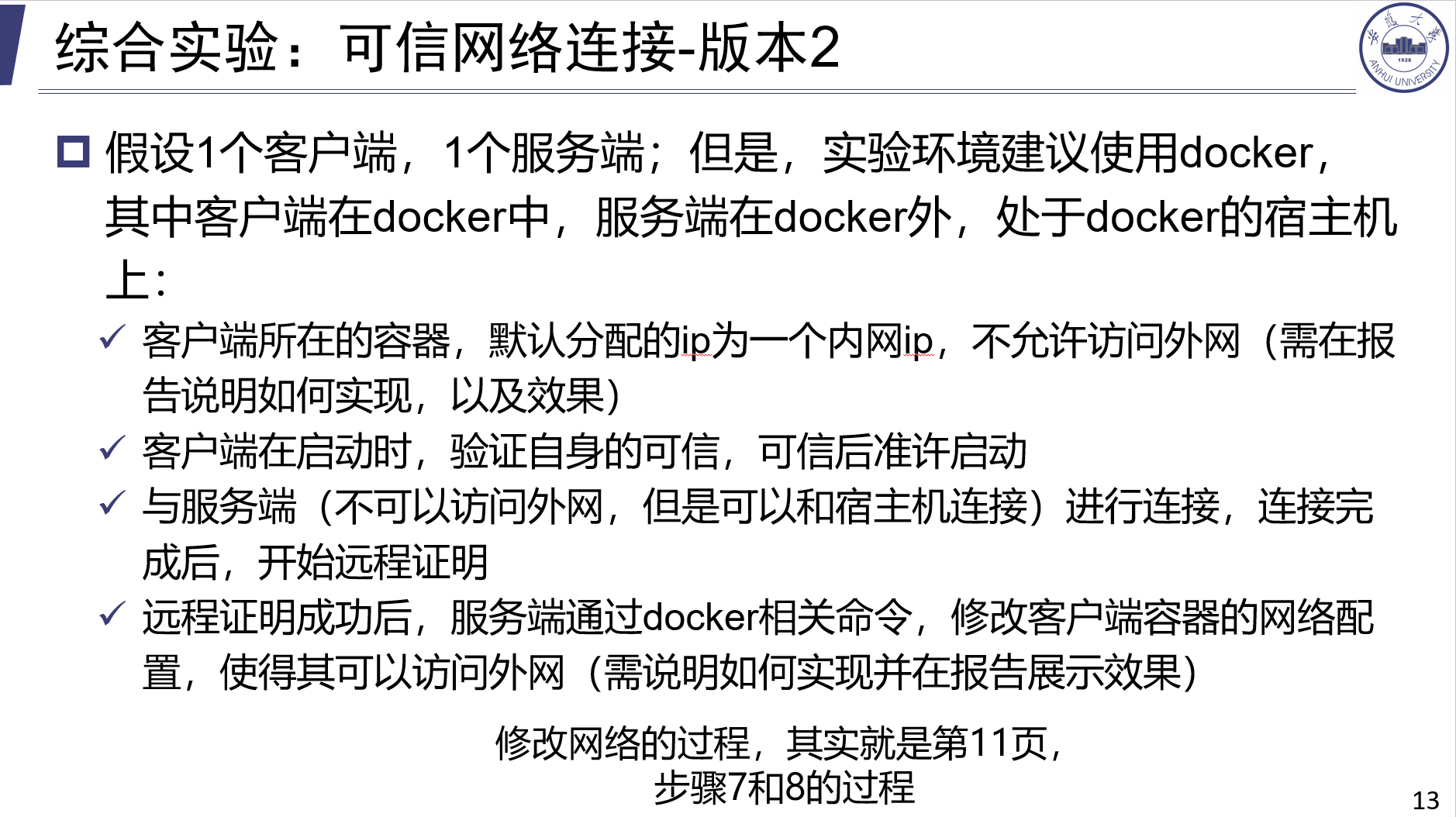
假设1个客户端，1个服务端；但是，实验环境建议使用docker，其中客户端在docker中，服务端在docker外，处于docker的宿主机上：

* + 客户端所在的容器，默认分配的ip为一个内网ip，不允许访问外网（需在报告说明如何实现，以及效果）
  + 客户端在启动时，验证自身的可信，可信后准许启动
  + 与服务端（不可以访问外网，但是可以和宿主机连接）进行连接，连接完成后，开始远程证明
  + 远程证明成功后，服务端通过docker相关命令，修改客户端容器的网络配置，使得其可以访问外网（需说明如何实现并在报告展示效果）

****

**【实验原理】**

****

****

Docker分配内网IP，禁止访问外网的原理：

使用默认的桥接默认创建Docker容器，在桥接模式下，容器通过docker0网卡（默认为172.17.0.2），与宿主机网卡交互，通过宿主机转发流量进而与外部进行通信。默认模式下，容器可以正常访问外网。

因此，通过iptables设置宿主机网络防火墙，丢弃全部来自docker0网卡的转发或转发到docker0的数据包，即可实现禁止docker容器访问外网。同理，当认证通过后，删除该策略，容器可正常访问外网。

sudo iptables -I FORWARD -s 172.17.0.2 -j DROP # 所有来自172.17.0.2的转发数据会被丢弃

命令解释：-I：表示插入一条FORWARD规则，FORWARD表示转发的流量包

-s：表示源地址为172.17.0.2

-j：表示操作：DROP代表丢弃

该命令需要root权限，因此，运行服务端时，需要确保以root权限启动。

sudo iptables -D FORWARD -s 172.17.0.2 -j DROP # 移除该策略

可以通过python的subprocess.run方法执行shell命令。

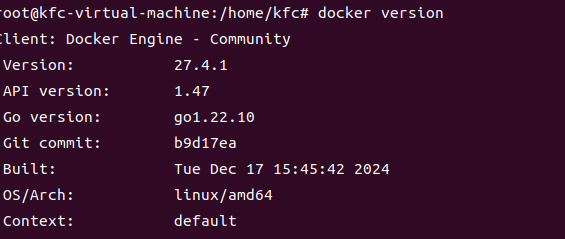
效果：

****

开启策略，数据包全部被丢弃，无法访问外网

**【实验内容】**

系统环境：Ubuntu22.04



Docker版本

Python3.7 第三方依赖库：pycryptodome

整个系统分为客户端与服务端，采用python的socket库实现通讯；采用python的pycryptodome库实现公钥，哈希算法。

**定义数据包格式：**

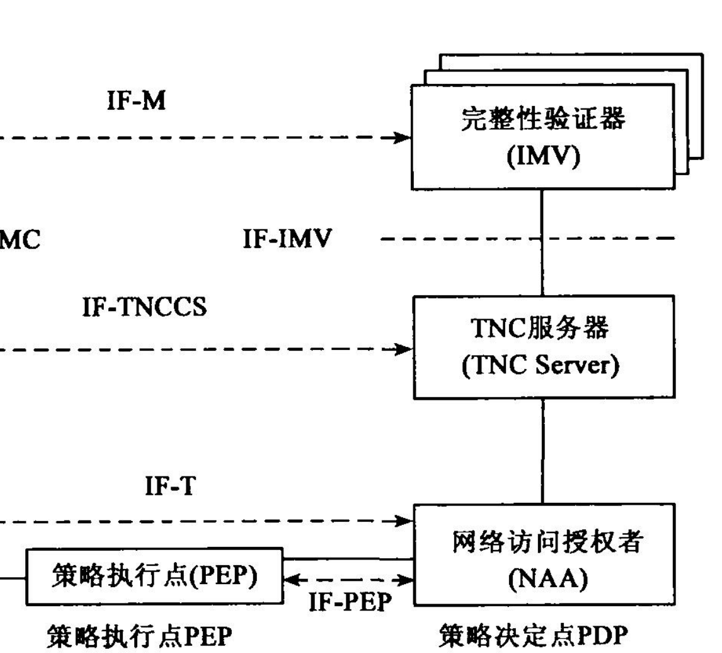
消息头的格式如下：

header = struct.pack('2sI1s', b'CA', file\_length, OperationType.ACTION\_RecvFile.to\_byte())

认证字段CA；消息（文件）长度；操作类型

**服务端实现：**

服务端包含以下几个模块：



**构造函数：**

class Server:  
 def \_\_init\_\_(self, host, port, container\_ip):  
 self.container\_ip = container\_ip  
 self.socket = socket.socket()  
 self.socket.bind((host, port))  
 self.socket.listen(5)  
 self.auth\_started = False  
 self.required\_files = [  
 "pcr\_values.txt",  
 "Sig\_CA\_AIK",  
 "AIK\_public.pem",  
 "Sig\_AIK\_PCR"  
 ]

在构造函数中，绑定监听地址与端口（这里设置0.0.0.0与8888，监听所有IPV4请求），设置客户端（容器）IP，确认需要从客户端接收的文件。

**接收新的客户端连接：**（当前服务端没有做多线程处理新的连接的功能，一次只能与一个客户端进行通信）

def accept\_connection(self):  
 self.conn, self.addr = self.socket.accept()  
 print("Connected by", self.addr)

**接收请求，生成响应：**

def run(self):  
 self.accept\_connection()  
 while True:  
 message\_parts = self.recv\_message()  
 if message\_parts is None:  
 break  
 ca, op\_type, message = message\_parts  
 op, response = self.handle\_message(ca, op\_type, message) # 处理请求，生成响应(包含响应内容与响应操作)  
 self.send\_response(op, response)  
 self.close\_connection()

**处理接收的请求，返回消息头部认证字段，操作类型，以及数据部分**

def recv\_message(self):  
 header = self.conn.recv(9)  
 if not header:  
 return None  
 ca, length, op\_type = struct.unpack('2sI1s', header)  
 message = self.conn.recv(length)  
 return ca, op\_type, message

**根据操作类型响应请求：**

def handle\_message(self, ca, op\_type, message):  
 if ca != b'CA':  
 print("Authentication failed")  
 return None, None  
  
 operation = OperationType.from\_byte(op\_type)  
 if not operation:  
 print("Unknown operation type:", op\_type)  
 return None, "Unknown operation type."  
 # 根据操作类型，执行操作并生成响应  
 if operation == OperationType.ACTION\_A:  
 try:  
 message\_decoded = message.decode('utf-8')  
 print("ACTION\_A - Message:", message\_decoded)  
 response = "Hello Client!."  
 except UnicodeDecodeError:  
 print("Failed to decode ACTION\_A message.")  
 response = "Invalid message format."  
 return operation, response  
  
 elif operation == OperationType.ACTION\_B:  
 print("ACTION\_B - Performing Action B.")  
 response = "Action B completed."  
 return operation, response  
  
 elif operation == OperationType.ACTION\_C:  
 print("ACTION\_C - Performing Action C.")  
 response = "Action C completed."  
 return operation, response  
  
 elif operation == OperationType.ACTION\_RecvFile:  
 success = self.receive\_file(message)  
 response = "File received." if success else "File reception failed."  
 operation = OperationType.ACTION\_RecvFile  
 return operation, response  
  
 else:  
 print("Unhandled operation type:", operation)  
 response = "Unhandled operation type."  
 return operation, response

当检测到OperationType.ACTION\_RecvFile的操作类型时，进行接收文件操作

**接收文件：**

def receive\_file(self, file\_data):  
 *"""  
 解析并保存接收到的文件数据，并进行文件累计确认。  
 """* if len(file\_data) < 4:  
 print("接收到的文件数据过短。")  
 return False  
  
 # 解析文件名长度  
 filename\_length = struct.unpack('I', file\_data[:4])[0]  
 if len(file\_data) < 4 + filename\_length:  
 print("接收到的文件数据不完整。")  
 return False  
  
 # 解析文件名  
 try:  
 filename = file\_data[4:4 + filename\_length].decode('utf-8')  
 except UnicodeDecodeError as e:  
 print(f"文件名解码失败: {e}")  
 return False  
  
 # 解析文件内容  
 file\_content = file\_data[4 + filename\_length:]  
  
 filename = os.path.basename(filename)  
  
 # 保存文件到服务器指定目录（例如当前目录）  
 save\_path = os.path.join(os.getcwd(), filename)  
 try:  
 with open(save\_path, 'wb') as f:  
 f.write(file\_content)  
 print(f"成功接收并保存文件：{save\_path} ({len(file\_content)} 字节)")  
  
 # 检查所有必需文件是否存在  
 all\_files\_present = all(  
 os.path.isfile(os.path.join(os.getcwd(), req\_file)) for req\_file in self.required\_files)  
 if all\_files\_present and not self.auth\_started:  
 print("所有必需文件已接收，开始平台认证。")  
 self.auth\_started = True # 设置标志，防止重复启动  
 # 启动一个新的子线程执行平台认证函数  
 auth\_thread = threading.Thread(target=self.platform\_authenticate, daemon=True)  
 auth\_thread.start()  
  
 return True  
 except IOError as e:  
 print(f"保存文件时出错: {e}")  
 return False

这里使用的策略是，每接收到一个文件，都在指定的目录下进行一次判断，判断所需的四个文件（构造函数中给出）是否都已存在。若存在，则进行平台验证。

但是这里需要注意，每接收完一个文件应当给客户端返回一个响应，告知客户端可以继续发送下一个文件或是文件已经全部正确送达。因此，当服务端已经接收到最后一个所需的文件时，此时如果直接在主线程中开启平台验证，那么会造成响应无法及时返回，导致客户端无法及时接收到响应。因此，需要创建一个子线程进行平台验证。

**平台验证：**

def platform\_authenticate(self):  
 *"""  
 平台认证函数  
 将在接收所有必需文件后被调用。  
 """* print("开始执行平台认证...")  
 verca = Crypto\_Helper.verify\_file\_signature("AIK\_public.pem", "Sig\_CA\_AIK", "CA\_public.pem")  
 if not verca:  
 print("CA验签失败，程序退出")  
 # 此处应当向客户端发送一个响应  
 exit(0)  
 else:  
 print("CA认证通过")  
 verAIK = Crypto\_Helper.verify\_file\_signature("pcr\_values.txt", "Sig\_AIK\_PCR", "AIK\_public.pem")  
 if not verAIK:  
 print("AIK验证失败，程序退出")  
 # 此处应当向客户端发送一个响应  
 exit(0)  
 else:  
 print("AIK验证通过")  
 # 对比pcr文件内容  
 IMV.IMC\_Check() # 服务端生成自己的pcr报告  
 with open("Server\_pcr\_values.txt", 'rb') as f1, open("pcr\_values.txt", 'rb') as f2:  
 pcr1 = f1.read()  
 pcr2 = f2.read()  
 if not pcr1 == pcr2:  
 print("完整性认证不通过，拒绝访问")  
 # 此处应当向客户端发送一个响应  
 exit(0)  
 else:  
 print("完整性验证通过，准许连接网络")  
 response = "PASS"  
 Net\_Manager.connect\_to\_net(self.container\_ip)  
 self.send\_response(OperationType.Action\_Accept, response)

这就是实验3远程证明的部分。客户端相当于示证者，服务端相当于验证者。

服务端收到消息后，首先利用验证{, }的合法性，通过的话则再用验证REPORT, 的合法性，比对REPORT和自己的是否一致（这里应该是IMV模块的作用，实验中图方便写到一块了）。若全部通过则认证通过，准许连接外部网络。

**公钥密码函数：**

直接调的pycrytodome库。实现了计算文件哈希，构建信任链，验证信任链，密钥对生成，签名，验签的功能。

这里服务端只需要验签就行了

from Crypto.PublicKey import RSA  
from Crypto.Cipher import PKCS1\_OAEP  
from Crypto.Signature import pkcs1\_15  
from Crypto.Hash import SHA256  
import binascii  
  
  
def verify\_file\_signature(file\_path, signature\_file\_path, public\_key\_path):  
 *"""从文件中读取内容和签名，使用指定的公钥验证签名"""* # 从文件中读取内容  
 with open(file\_path, "rb") as file:  
 message = file.read()  
  
 # 读取签名文件  
 with open(signature\_file\_path, "rb") as sig\_file:  
 signature = binascii.unhexlify(sig\_file.read())  
  
 # 从文件中读取公钥  
 with open(public\_key\_path, "rb") as pub\_file:  
 public\_key = RSA.import\_key(pub\_file.read())  
  
 # 对文件内容进行哈希处理  
 h = SHA256.new(message)  
  
 # 验证签名  
 try:  
 pkcs1\_15.new(public\_key).verify(h, signature)  
 return True  
 except (ValueError, TypeError):  
 return False

**对客户端的网络管理：**

import subprocess  
import platform  
import sys  
  
  
def is\_linux():  
 *"""  
 检测当前操作系统是否为 Linux。  
 """* return platform.system().lower() == 'linux'

def rule\_exists(container\_ip):  
 *"""  
 检查 iptables 规则是否已存在  
 """* check\_command = ['iptables', '-C', 'FORWARD', '-s', container\_ip, '-j', 'DROP']  
 try:  
 subprocess.run(check\_command, check=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE)  
 return True # 规则存在  
 except subprocess.CalledProcessError:  
 return False # 规则不存在

def forbidden\_from\_net(container\_ip):  
 *"""  
 执行 iptables 命令，阻止指定容器 IP 访问外网。  
 """* if not is\_linux():  
 print("错误: 此程序只能在 Linux 系统上运行。")  
 sys.exit(1)  
 # 定义要执行的命令  
 command = ['iptables', '-I', 'FORWARD', '-s', container\_ip, '-j', 'DROP']  
  
 try:  
 # 执行命令  
 result = subprocess.run(command, check=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE)  
 print(f"成功添加 iptables 规则：{' '.join(command)}")  
 except subprocess.CalledProcessError as e:  
 print(f"添加 iptables 规则失败：{e.stderr.decode().strip()}")  
 except Exception as e:  
 print(f"发生错误：{str(e)}")  
  
  
def connect\_to\_net(container\_ip):  
 *"""  
 移除之前添加的 iptables 规则，恢复容器对外网的访问。  
 """* if not is\_linux():  
 print("错误: 此程序只能在 Linux 系统上运行。")  
 sys.exit(1)  
 command = ['iptables', '-D', 'FORWARD', '-s', container\_ip, '-j', 'DROP']  
 try:  
 result = subprocess.run(command, check=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE)  
 print(f"成功移除 iptables 规则：{' '.join(command)}")  
 except subprocess.CalledProcessError as e:  
 print(f"移除 iptables 规则失败：{e.stderr.decode().strip()}")  
 except Exception as e:  
 print(f"发生错误：{str(e)}")

首先检查一下规则是否已存在，防止重复插入

container\_ip是客户端的ip地址。首先通过执行

iptabels -I FORWARD -s ${container\_ip} -j DROP

实现丢弃客户端全部的转发包，使客户端无法访问外网。

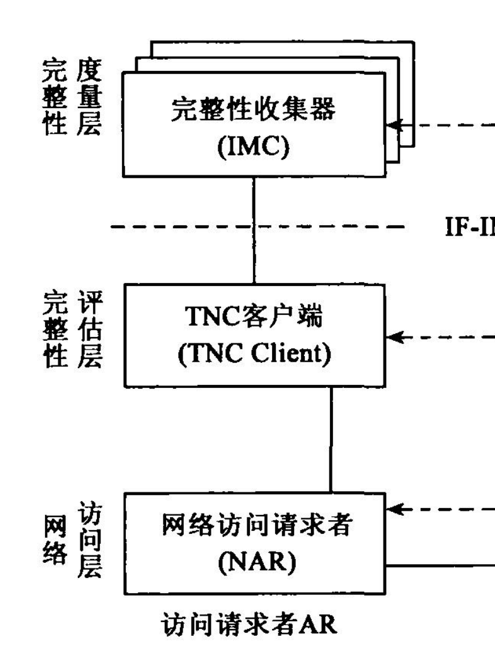
当平台认证通过后，执行

Iptables -D FORWARD -s ${container\_ip} -j DROP

删除之前的策略，恢复客户端对外部网络的访问

**客户端实现：**

客户端分为以下几个部分：



**客户端构造函数：**

class NAR:  
 def \_\_init\_\_(self, host, port):  
 self.host = host  
 self.port = port  
 self.socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
 self.listen\_thread = None  
 self.listening = False  
 self.response\_queue = queue.Queue() # 用于存储服务器响应

**连接到服务器：**

def connect(self):  
 self.socket.connect((self.host, self.port)) #   
 print("已连接到服务器", self.host, self.port)

根据成员变量地址与端口，绑定到服务端

**发送单个文件：**

def send\_file(self, file\_path):  
 *"""  
 发送文件到服务端，按照以下协议：  
 - 负载：  
 - 4 字节：文件名长度（无符号整数）  
 - n 字节：文件名（UTF-8 编码）  
 - m 字节：文件内容  
 """* # 检查文件是否存在  
 if not os.path.isfile(file\_path):  
 print("文件不存在")  
 return False  
  
 # 读取文件内容  
 with open(file\_path, 'rb') as file:  
 file\_data = file.read()  
 # 获取文件名并编码  
 filename = os.path.basename(file\_path)  
 filename\_bytes = filename.encode('utf-8')  
 filename\_length = len(filename\_bytes)  
 # 构建文件数据：4字节文件名长度 + 文件名 + 文件内容  
 file\_payload = struct.pack('I', filename\_length) + filename\_bytes + file\_data  
 # 计算总长度  
 total\_length = len(file\_payload)  
 # 构建头部  
 header = struct.pack('2sI1s', b'CA', total\_length, OperationType.ACTION\_RecvFile.to\_byte())  
 try:  
 # 发送头部和文件数据  
 self.socket.sendall(header + file\_payload)  
 print(f"文件 {file\_path} 已发送到服务器。")  
 return True  
 except Exception as e:  
 print(f"发送文件时出错: {e}")  
 return False

发送文件时，除了数据部分外，还需要附带文件名与文件名长度，方便服务端接收。

**发送多个文件到服务端：**

def send\_multiple\_files\_sequentially(self, file\_paths):  
 *"""  
 依次发送多个文件，确保每个文件发送后收到响应再发送下一个文件。  
 """* for file\_path in file\_paths:  
 print(f"准备发送文件: {file\_path}")  
 success = self.send\_file(file\_path)  
 if not success:  
 print(f"发送文件失败: {file\_path}")  
 continue # 继续尝试发送下一个文件  
  
 try:  
 # 等待响应，超时可以根据需要设置  
 response = self.response\_queue.get(timeout=5) # 5秒超时  
 operation, message = response  
 if operation == OperationType.ACTION\_RecvFile:  
 print(f"文件 {file\_path} 已被服务器接收，服务器响应: {message}")  
 else:  
 print(f"收到非文件接收的响应: 操作类型 {operation}, 消息 {message}")  
 except queue.Empty:  
 print(f"等待服务器响应超时，未能确认文件 {file\_path} 是否被接收。")  
 except Exception as e:  
 print(f"处理响应时出错: {e}")

为确保服务端正确接收全部文件，客户端应当每发送完一次文件后，都应当等待服务端正确接收文件的响应，得到响应后才能继续发送下一个文件。

因此，需要设置一个响应队列，每当接收到服务端正确接收文件的响应时，就将该响应入队；当客户端发送完一次文件后，就从队列中检测是否有请求到达。若超过等待时间，则认为文件传输发生错误，可以进行重传。

**客户端监听响应：**

def listen\_for\_responses(self):  
 *"""  
 监听来自服务器的响应。  
 """* self.listening = True  
 try:  
 while self.listening:  
 response = self.receive\_response()  
 if response:  
 self.handle\_response(response)  
 except Exception as e:  
 print("监听异常:", str(e))  
 finally:  
 self.close()

可以看到，该监听函数包含了一个无限while循环，这会导致阻塞主进程。因此，可以采用为监听函数创建子线程的方式解决阻塞问题：

**多线程解决监听阻塞问题：**

def start\_listening(self):  
 *"""  
 启动监听线程。  
 """* self.listen\_thread = threading.Thread(target=self.listen\_for\_responses, daemon=True) # 创建子线程  
 self.listen\_thread.start()  
 print("已启动监听线程。")

* 使用 threading.Thread 创建一个守护线程 (daemon=True) 来监听服务器的响应。
* 这样即使主线程结束，监听线程也会自动退出。

**接收响应：**

def receive\_response(self):  
 header = self.socket.recv(9) # Assume the same header structure as the server  
 if not header:  
 return None  
 ca, length, op\_type = struct.unpack('2sI1s', header)  
 message = self.socket.recv(length).decode('utf-8')  
 return OperationType.from\_byte(op\_type), message # 此处对操作进行解码，后续无需进行解码操作  
  
def handle\_response(self, response):  
 *"""  
 处理来自服务器的响应。  
 """* operation, message = response  
 print(f"收到响应 - 操作类型: {operation.name}, 消息: {message}")  
 # 根据操作类型执行相应的逻辑  
 if operation == OperationType.ACTION\_A:  
 print("处理 ACTION\_A 响应")  
 # 添加相应的处理逻辑  
 elif operation == OperationType.ACTION\_B:  
 print("处理 ACTION\_B 响应")  
 # 添加相应的处理逻辑  
 elif operation == OperationType.ACTION\_C:  
 print("处理 ACTION\_C 响应")  
 # 添加相应的处理逻辑  
 elif operation == OperationType.ACTION\_RecvFile:  
 print("处理 ACTION\_RecvFile 响应")  
 # 将响应放入队列  
 self.response\_queue.put(response)  
 # 添加相应的处理逻辑  
 elif operation == OperationType.Action\_Accept:  
 print("可信验证通过，现在可以访问外网")  
 self.ping\_baidu()  
  
 else:  
 print("未处理的操作类型:", operation)

处理逻辑和服务端是一样的，都是根据操作类型去执行相应操作。

**客户端完整性验证（IMC）：**

这其实就是实验一，实验二的步骤，这里采用当前路径下四个.py文件作为信任链构建的文件。

FILE1\_PATH = 'Client.py'  
FILE2\_PATH = 'Crypro\_Helper.py'  
FILE3\_PATH = 'NAR.py'  
FILE4\_PATH = 'IMC.py'

def compute\_hash(file\_path):  
 *"""计算文件的SHA-256哈希值。"""* h = SHA256.new()  
 try:  
 with open(file\_path, 'rb') as f:  
 data = f.read()  
 h.update(data)  
 return h.hexdigest()  
 except FileNotFoundError:  
 print(f"文件未找到: {file\_path}")  
 return None  
  
  
def compute\_hash\_with\_link(file\_path, linked\_hash):  
 *"""计算文件与另一个哈希链接后的哈希值。"""* h = SHA256.new()  
 try:  
 with open(file\_path, 'rb') as f:  
 data = f.read()  
 h.update(data)  
 h.update(linked\_hash.encode())  
 return h.hexdigest()  
 except FileNotFoundError:  
 print(f"文件未找到: {file\_path}")  
 return None  
  
  
def build\_trust\_chain():  
 *"""从文件4到文件1构建信任链。"""* H4 = compute\_hash(FILE4\_PATH)  
 if H4 is None:  
 return None  
 H3 = compute\_hash\_with\_link(FILE3\_PATH, H4)  
 if H3 is None:  
 return None  
 H2 = compute\_hash\_with\_link(FILE2\_PATH, H3)  
 if H2 is None:  
 return None  
 H1 = compute\_hash\_with\_link(FILE1\_PATH, H2)  
 if H1 is None:  
 return None  
 return H1, H2, H3, H4  
  
  
def verify\_trust\_chain(H1, H2, H3, H4):  
 *"""验证存储的哈希与信任链是否匹配。"""* if compute\_hash\_with\_link(FILE1\_PATH, H2) != H1:  
 print(compute\_hash\_with\_link(FILE1\_PATH, H2))  
 print("H1验证失败")  
 return False  
 if compute\_hash\_with\_link(FILE2\_PATH, H3) != H2:  
 print("H2验证失败")  
 return False  
 if compute\_hash\_with\_link(FILE3\_PATH, H4) != H3:  
 print("H3验证失败")  
 return False  
 if compute\_hash(FILE4\_PATH) != H4:  
 print("H4验证失败")  
 return False  
 print("信任链验证成功")  
 return True  
  
  
def read\_pcr\_values():  
 *"""从信任链文件中读取值。"""* if not os.path.exists(TRUST\_FILE\_PATH):  
 return [None] \* 4  
 with open(TRUST\_FILE\_PATH, 'r') as file:  
 return [line.strip() for line in file.readlines()]  
  
  
def write\_trust\_values(pcr\_values):  
 *"""将信任链值写入文件。"""* with open(TRUST\_FILE\_PATH, 'w') as file:  
 for value in pcr\_values:  
 file.write(f"{value}\n")  
  
  
def write\_pcr\_values(pcr\_values):  
 *"""将PCR值写入文件。"""* with open(PCR\_REPORT\_PATH, 'w') as file:  
 for value in pcr\_values:  
 file.write(f"{value}\n")  
  
  
def build\_PCR():  
 H4 = compute\_hash(FILE4\_PATH)  
 if H4 is None:  
 return None  
 H3 = compute\_hash(FILE3\_PATH)  
 if H3 is None:  
 return None  
 H2 = compute\_hash(FILE2\_PATH)  
 if H2 is None:  
 return None  
 H1 = compute\_hash(FILE1\_PATH)  
 if H1 is None:  
 return None  
 return H1, H2, H3, H4  
  
  
def IMC\_Check():  
 pcr\_values = read\_pcr\_values() # 读取预存的hash，进行信任链验证  
 # 检查是否所有信任链值都存在且有效  
 if None not in pcr\_values and len(pcr\_values) == 4:  
 print("存在有效的信任链值，直接进行信任链验证...")  
 H1, H2, H3, H4 = pcr\_values  
 if verify\_trust\_chain(H1, H2, H3, H4):  
 print("信任链验证成功。")  
 pcr\_chain = build\_PCR()  
 write\_pcr\_values(pcr\_chain)  
 print("PCR值已更新。生成PCR文件")  
 print("开始NAR验证")  
 return True  
 else:  
 print("信任链验证失败。")  
 return False  
 else:  
 print("不存在有效的信任链值，需要构建信任链...")  
 trust\_chain = build\_trust\_chain()  
 if trust\_chain is None:  
 print("构建信任链失败。")  
 return False  
  
 if verify\_trust\_chain(\*trust\_chain):  
 write\_trust\_values(trust\_chain)  
 pcr\_chain = build\_PCR()  
 write\_pcr\_values(pcr\_chain) # 生成pcr文件  
 print("PCR值已更新。生成PCR文件")  
 print("信任链验证成功。")  
 print("开始NAR验证")  
 return True  
 else:  
 print("信任链验证失败。")  
 return False

实现信任链构建：

先计算最后一个文件4哈希，然后把这个和前一个文件3链接后计算哈希，再和前一个文件2链接计算哈希， 再和前一个文件1链接计算哈希。则建立起了一个信任链🡪🡪🡪。

实现信任链验证：

利用前面计算出来的信任链🡪🡪🡪，先读取文件1，拼接上𝐻\_2，计算哈希值，判断和预存的𝐻\_1是否相等，再判断文件2和𝐻\_3的哈希值是否和预存𝐻\_2相等，再判断文件3和𝐻\_4的哈希值是否和预存的𝐻\_3相等，最后再判断文件4的哈希值和预存的𝐻\_4是否相等。

验证通过的话计算，更新一下PCR文件的内容（每个文件的哈希）

**公钥密码函数：**

和服务端一样，调用的pycryptodome库，客户端在平台验证的过程中充当实验三中示证者的身份，因此客户端需要完成的操作有：

生成AIK密钥对

使用SK\_CA对PK\_AIK进行签名

使用SK\_AIK对PCR文件进行签名

**网络测试函数：**

用subprocess执行一下ping -c 4 [www.baidu.com](http://www.baidu.com)命令。

**系统整体运行流程（py文件执行）：**

注意，本系统默认已经完成了和CA有关的操作，默认客户端和服务端都正确持有CA密钥对。

**启动服务端：**

container\_ip = '172.17.0.2'  
Net\_Manager.forbidden\_from\_net(container\_ip)  
host, port = "0.0.0.0", 8888 # 监听所有IPV4请求  
server = Server(host, port, container\_ip)  
server.run()

首先添加禁止容器访问外部网络的策略，丢弃所有容器的转发流量。

指定客户端IP，监听地址与监听端口。创建服务端实例，开始监听。

**启动客户端，进行完整性校验：**

if not IMC\_Check():  
 print("完整性认证失败，退出程序")  
 exit(0)

客户端进行完整性校验，判断自己的四个py文件是否被修改。若完整性校验不通过，则退出程序。

**客户端连接到服务端：**

client = NAR('192.168.3.96', 8888) # 改成宿主机IP  
client.connect()  
client.start\_listening() # 开启监听子线程

完整性校验通过后，客户端指定服务端的地址与端口，进行连接。同时，创建一个子线程，监听服务端的响应。

**客户端生成AIK密钥对，准备进行平台认证：**

# 生成AIK密钥对，开始远程证明  
Crypro\_Helper.generate\_keys("AIK\_private.pem", "AIK\_public.pem")  
# 对PCR文件进行签名  
Crypro\_Helper.sign\_file("AIK\_public.pem", "CA\_private.pem", "Sig\_CA\_AIK")

Crypro\_Helper.sign\_file("pcr\_values.txt", "AIK\_private.pem", "Sig\_AIK\_PCR")

客户端生成生成AIK密钥对，使用SK\_CA对PK\_AIK进行签名；使用SK\_AIK对PCR文件进行签名。签名的内容均保存到文件中。

**客户端发送文件到服务端，由服务端进行验证：**

files\_to\_send = [  
 "pcr\_values.txt",  
 "Sig\_CA\_AIK",  
 "AIK\_public.pem",  
 "Sig\_AIK\_PCR"  
 # 添加更多文件路径  
]  
client.send\_multiple\_files\_sequentially(files\_to\_send)

设置一个列表，存放待发送的文件路径，调用发送多个文件的函数对文件进行有序发送。

**服务端接收所有文件，开始验证：**

服务端如果检测到请求中的操作类型为OperationType.ACTION\_RecvFile: b'R'

则调用receive\_file函数，首先解析出文件名长度与文件名，之后将文件内容写入文件。当所需文件全部接收完毕后，新建子线程进行验签。

**服务端验证客户端可信：**

调用platform\_authenticate函数，首先使用验证{, }的合法性，通过的话则再用验证REPORT, 的合法性。服务端拥有客户端的四个.py文件。服务端对这个四个文件生成PCR报告，与从客户端接收的PCR报告进行对比，若通过，则证明了客户端的可信，准许客户端访问外部网络。

**服务端允许客户端访问外网：**

当服务端验证成功后，执行connect\_to\_net函数，删除丢弃转发规则，恢复容器对外网的访问。

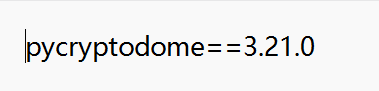
**客户端访问外网：**

result = subprocess.run(  
 ['ping', '-c', '4', 'baidu.com'],  
 stdout=subprocess.PIPE,  
 stderr=subprocess.PIPE,  
 text=True  
)

ping下百度。

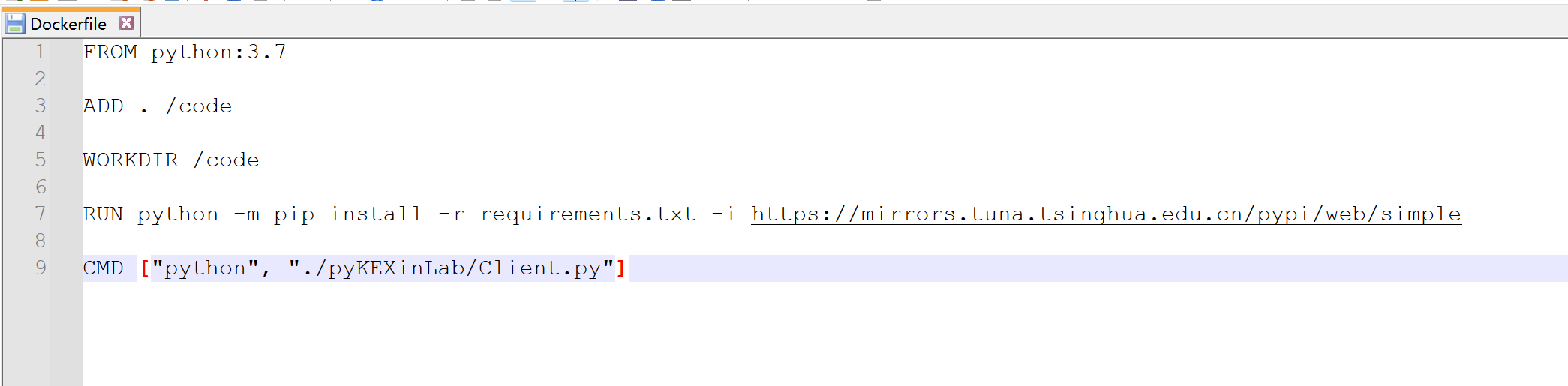
**Docker镜像的构建与系统运行效果：**

要构建一个docker镜像，需要找出python项目所有依赖的第三方库。这里我只使用了pycryptodome一个第三方库，所以requirements.txt直接写就行了。如果依赖库很多的话，建议用pip freeze或者pipreqs。



**编写DockerFile:**

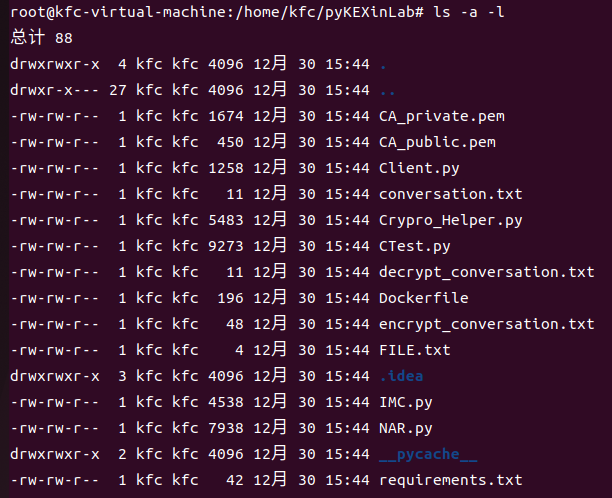
放在客户端项目的路径下



指定基础镜像为Python3.7，将当前路径下全部文件拷贝到镜像文件夹中（/code）

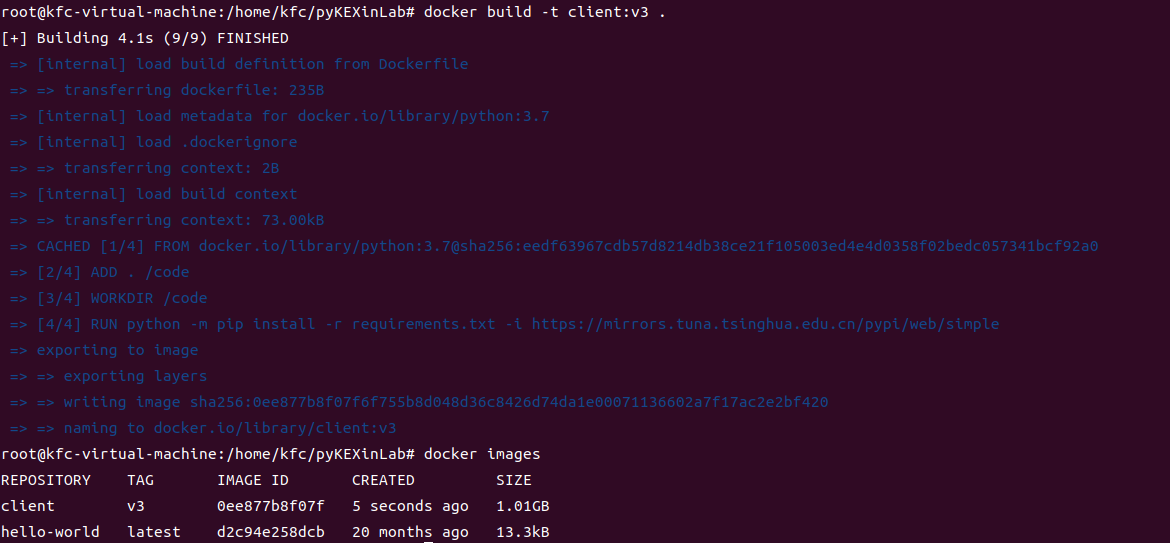
并设置工作路径为/code

把整个文件夹拷贝到Ubuntu中，准备构建Docker镜像



**构建docker镜像：**

执行命令：docker build -t client:v3 .

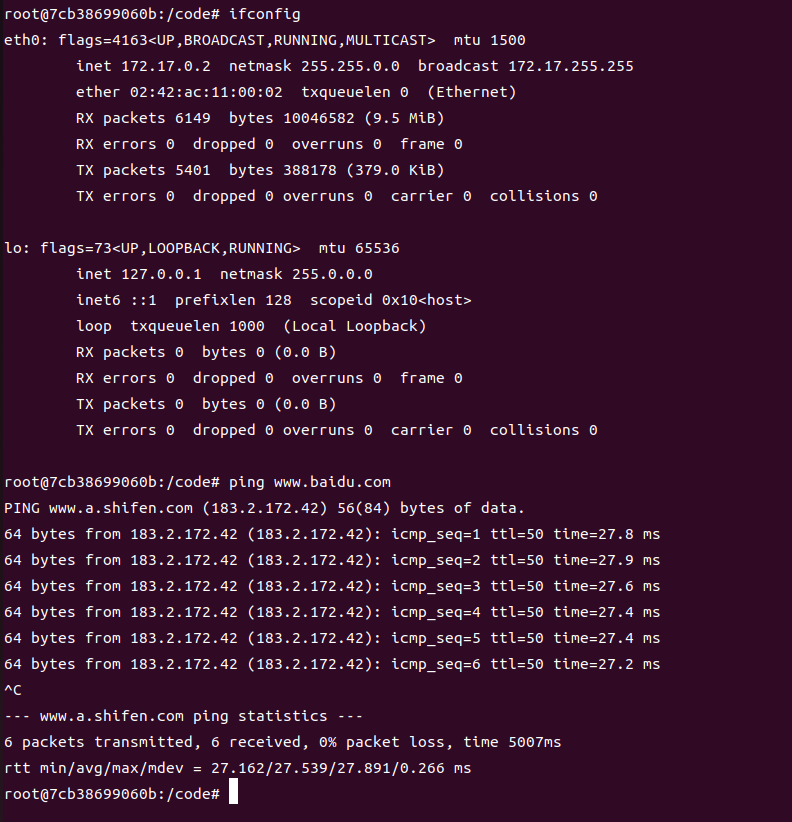


成功构建镜像，镜像名称为client:v3

**创建docker容器：**

docker run -it --name=container client:v3 /bin/bash

进入容器后，安装ifconfig与ping

****

客户端尝试ping外网，正常

**以root权限运行服务端:**

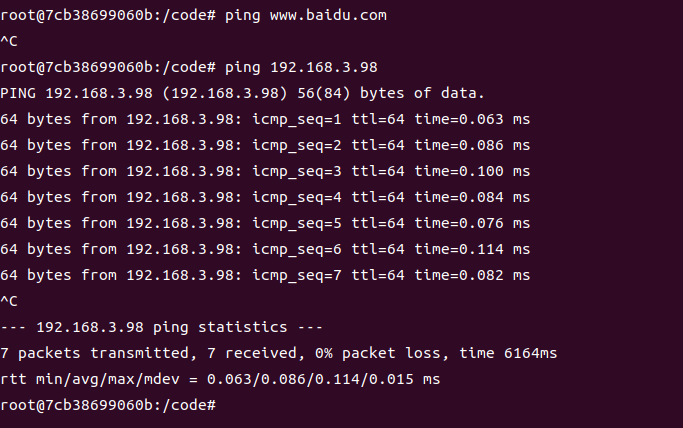
服务端输出：



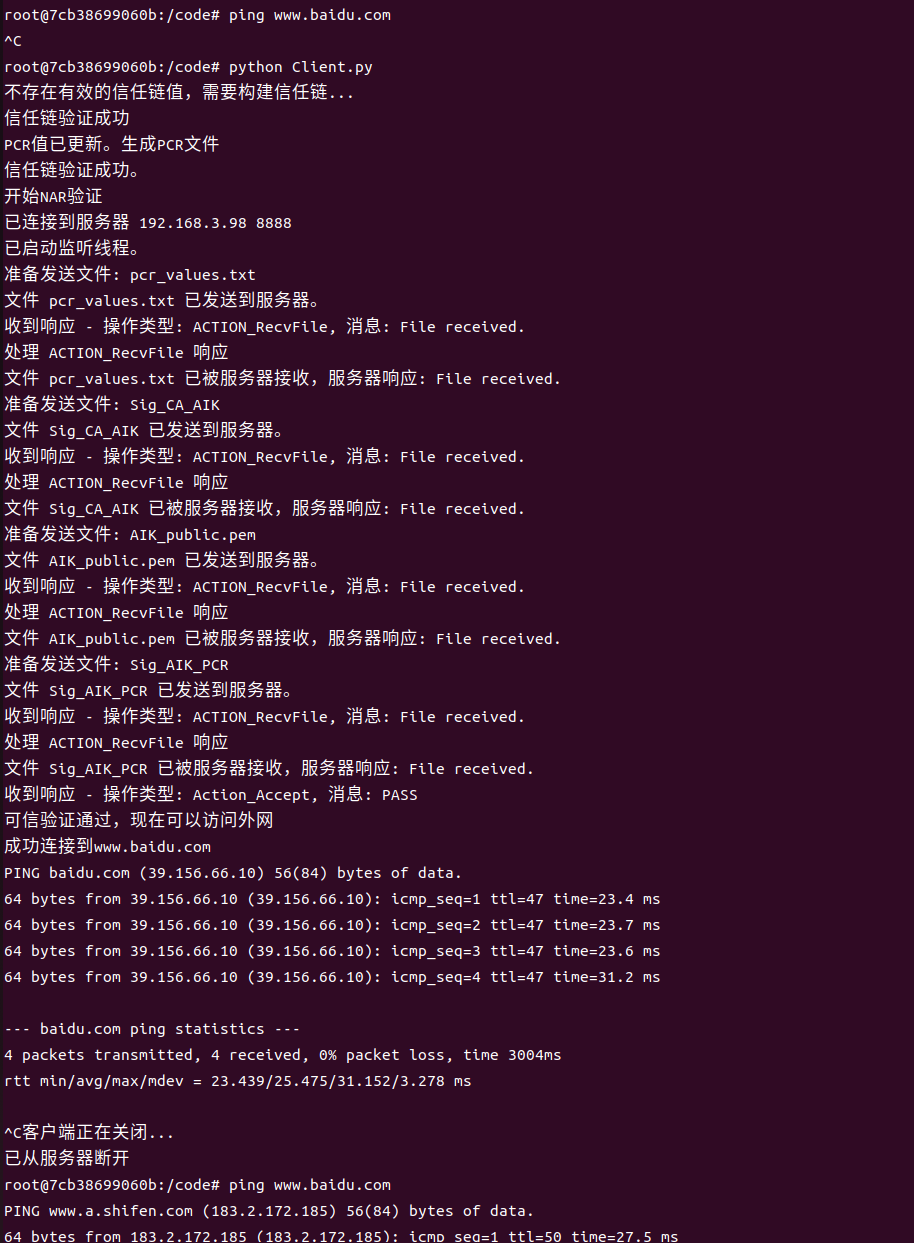
**以root权限运行服务端：**

Ping外网，不通

只能ping通宿主机



执行Client.py,开始可信验证：



**【实验小结】**

本次实验是可信计算的综合实验，实验内容覆盖了信任链的构建与验证，以及远程证明的过程。在此基础上，通过docker容器与宿主机的通信，模拟了可信网络连接的过程。客户端服务端之间的通信以及信任链构建，验证前面三次实验报告都说明过了，这里主要说一下本次实验关于docker遇到的问题与收获吧。

1. Docker是什么

我的理解就是，一个轻量化的虚拟机。把程序和依赖环境一起打包。与虚拟机不同的是，docker中的操作系统基础镜像舍弃了内核空间，只留下了用户空间，程序直接运行在宿主机的内核上。

1. DockerFile,Docker镜像和Docker容器

DockerFile：

用来构建镜像的文本文件，文本内容包含了一条条构建镜像所需的指令和说明。

FROM:指定基础镜像

RUN:执行后面的命令行命令

Docker镜像：

在含有DockerFile的目录下执行docker build命令时，系统会将程序与环境依赖打包为一个类似于压缩包的东西，这就是Docker镜像。通过Docker镜像，我们可以实例化容器

Docker容器：

容器时镜像的实例化（类似于对象和类的关系？）。通过docker run命令，可以将指定的docker镜像“解压”出一个独立的环境与应用。通过一个镜像可以创建多个容器。本质上容器是个自带运行环境特殊的进程，没有像虚拟机那样有完整的操作系统，也没有虚拟的硬件。

1. 碰到的坑

首先就是平台的选择了，一开始因为用的是Python，在Pycharm里面写的代码，所以这次实验就想在windows上做，然后就去下了Docker Desktop。打包是挺顺利的，但处理容器网络的时候就不对劲了，网上给的教程基本上全是iptables，没办法，用Ubuntu了。

容器网络的选择：

一开始我觉得容器肯定得和宿主机在一个网段下面，发现通过macvlan可以做到这一点。创建macvlan虚拟网络，分配IP，创建容器后一看，诶，外网能通，和宿主机不能通？把实验要求彻底搞反了？查资料发现macvlan网络创建的容器默认不能和宿主机通信，得再创建一个macvlan连接当中继才行。现在宿主机和容器通了，接下来该限制容器访问外网了，我采用的策略是让宿主机丢弃全部来自容器的转发包（FORWARD链）。结果发现容器还是能随便通外网，这肯定不对啊。又找，发现macvlan其实是给容器创了虚拟网卡，数据包不需要通过宿主机转发。FORWARD检测的是转发包，所以规则不起作用。

后来发现使用默认的桥接模式就能解决这个问题了。默认桥接模式下，容器有个docker0网卡，流量都是要经过宿主机转发的，这样宿主机只要把来自docker0的这些数据包丢弃就ok了。

还有个坑，就是iptables插入规则的时候可以多次插入同一条规则，然后删除只能删一条。一开始没做重复校验，这就导致服务端每次启动，不管可信验证通不通过，都要加一次禁止访问的规则。这就导致如果在认证失败后又验证成功，服务端执行了两次插入规则的操作，但只执行了一次删除规则，所以容器还是访问不了外网。解决方法也很简单，服务端每次插入规则的时候用iptables -C检查一下规则是否存在就行了