

# 用python实现cryptoChat

## 实验要求

任选编程语言和平台，调用主流密码学库，恰当选取参数，实现加密聊天程序。要求：

1. 熟悉多种密码加密、签名等算法的安全使用
2. 加密功能包括DES、AES、RSA、ECC等多种，了解不同算法、参数及加密模式的原理与差异。
3. 聊天过程中的消息使用RSA进行签名和验签。
4. 注意1：实验报告中需要对使用的参数、模式等内容做合理性解释。
5. 注意2：新版实验一与旧版实验一2选1即可，都做取均分

## 实验环境

编程语言：python

平台：vscode

主流密码库：cryptography

## 相关知识

### 工作模式

密码学中，分组密码的工作模式（mode of operation）允许使用同一个分组密码密钥对多于一块的数据进行加密，并保证其安全性。分组密码自身只能加密长度等于密码分组长度的单块数据，若要加密变长数据，则数据必须先被划分为一些单独的密码块。通常而言，最后一块数据也需要使用合适填充方式将数据扩展到符合密码块大小的长度。一种工作模式描述了加密每一数据块的过程，并常常使用基于一个通常称为初始化向量的附加输入值以进行随机化，以保证安全

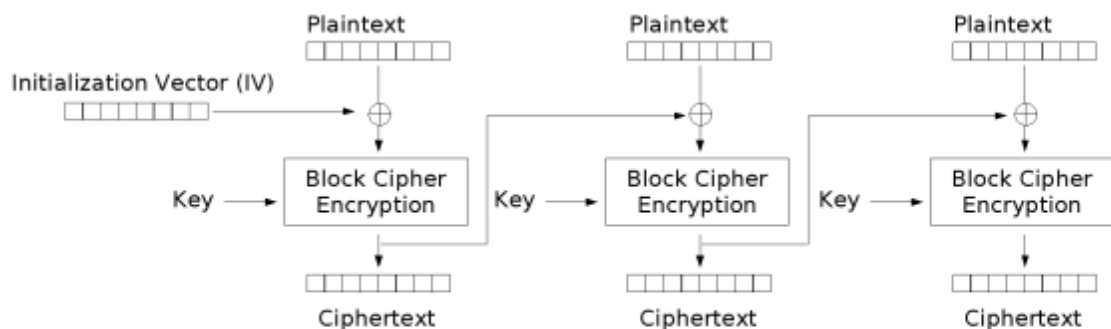
from wiki

主要的工作模式包括ECB，CBC，CFB，OFB和CTR等。

### CBC

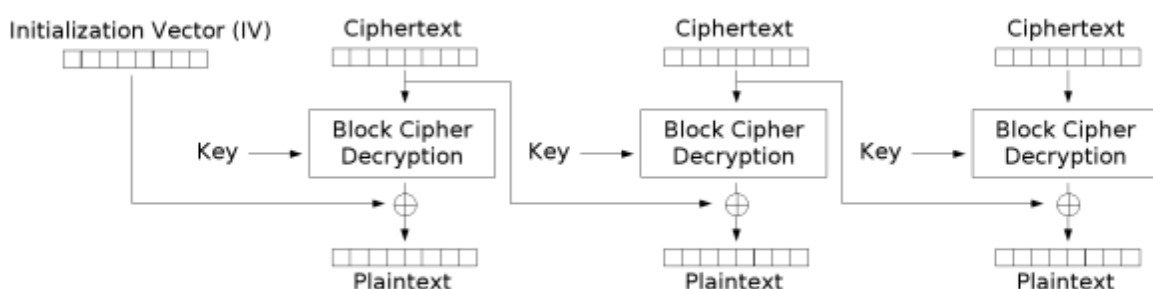
在CBC模式中，每个明文块先与前一个密文块进行异或后，再进行加密。在这种方法中，每个密文块都依赖于它前面的所有明文块。同时，为了保证每条消息的唯一性，在第一个块中需要使用初始化向量。

加密流程图



Cipher Block Chaining (CBC) mode encryption

解密流程图



Cipher Block Chaining (CBC) mode decryption

CBC是最为常用的工作模式。它的主要缺点在于加密过程是串行的，无法被并行化，而且消息必须被填充到块大小的整数倍。而下面就是相关的填充算法。

## 填充算法

填充算法主要在特定工作模式和散列函数中会涉及到，以下我们将实验涉及到的填充算法按照在实例中的应用进行粗略的划分。

### PKCS7

PKCS#7 定义于 [RFC 5652](#)（征求意见稿第 5652 号）。

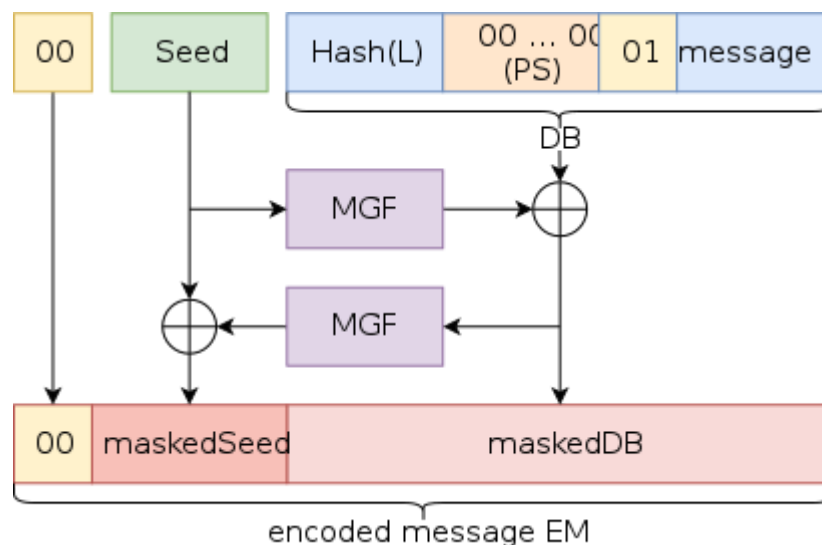
每个填充字节的值是用于填充的字节数，即是说，若需要填充  $N$  个字节，则每个填充字节值都是  $N$ 。填充的字节数取决于算法可以处理的最小数据块的字节数量。

P7定义了一种通用的消息语法，包括数字签名和加密等用于增强的加密机制，PKCS#7与PEM兼容，所以不需其他密码操作，就可以将加密的消息转换成PEM消息。

### rsa加密过程中的OAEP填充

In [cryptography](#), **Optimal Asymmetric Encryption Padding (OAEP)** is a [padding scheme](#) often used together with [RSA encryption](#).

OAEP填充算法

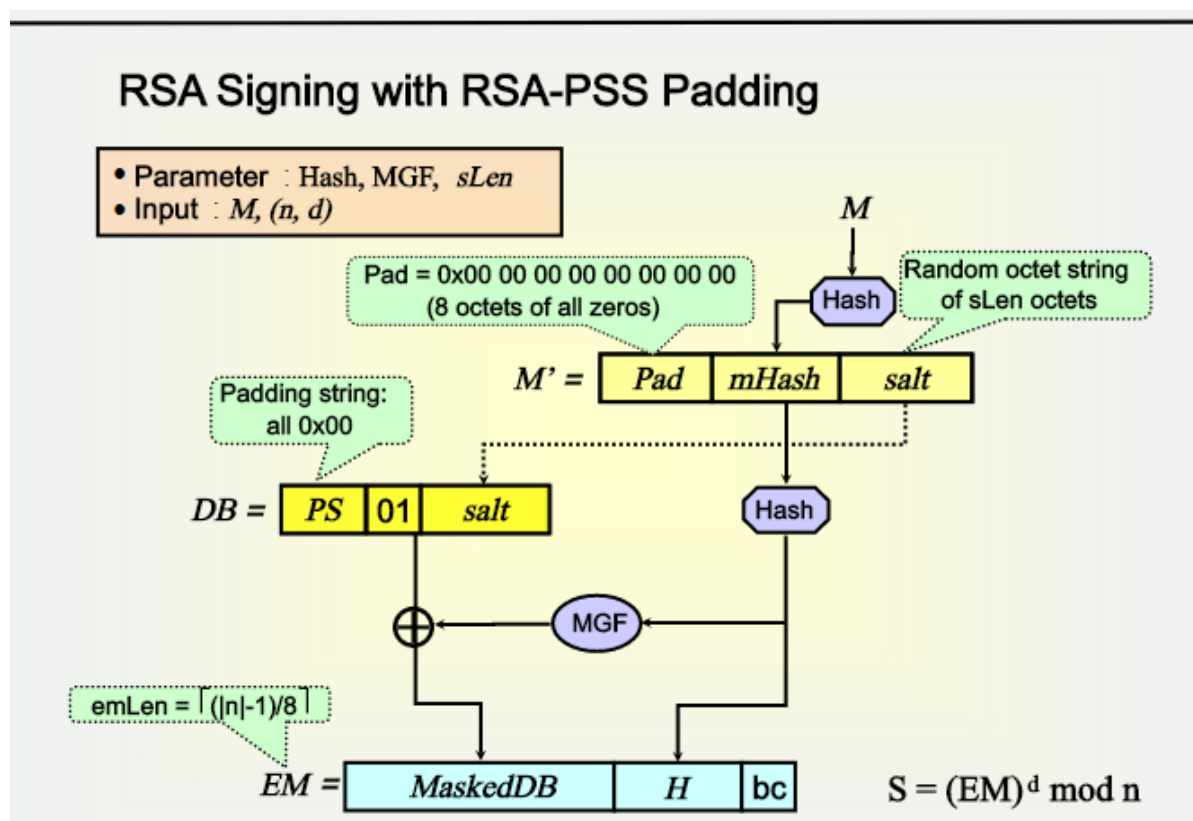


## rsa签名中的PSS填充

PSS (Probabilistic Signature Scheme) 私钥签名流程的一种填充模式。目前主流的RSA签名包括RSA-PSS和RSA-PKCS#1 v1.5。相对应PKCS (Public Key Cryptography Standards) 是一种能够自我签名，而PSS无法从签名中恢复恢复原来的签名。openssl-1.1.x以后默认使用更安全的PSS的RSA签名模式。

由于RSA算法比较慢，一般用于非对称加密的private key签名和public key验证。因RSA算法没有加入乱数，当出现重复性的原始资料，攻击者会通过相同加密密文而猜测出原文，因此导入padding的机制来加强安全性。

PSS填充算法



# 密钥派生

## HKDF

HKDF是一个基于HMAC的密钥派生算法

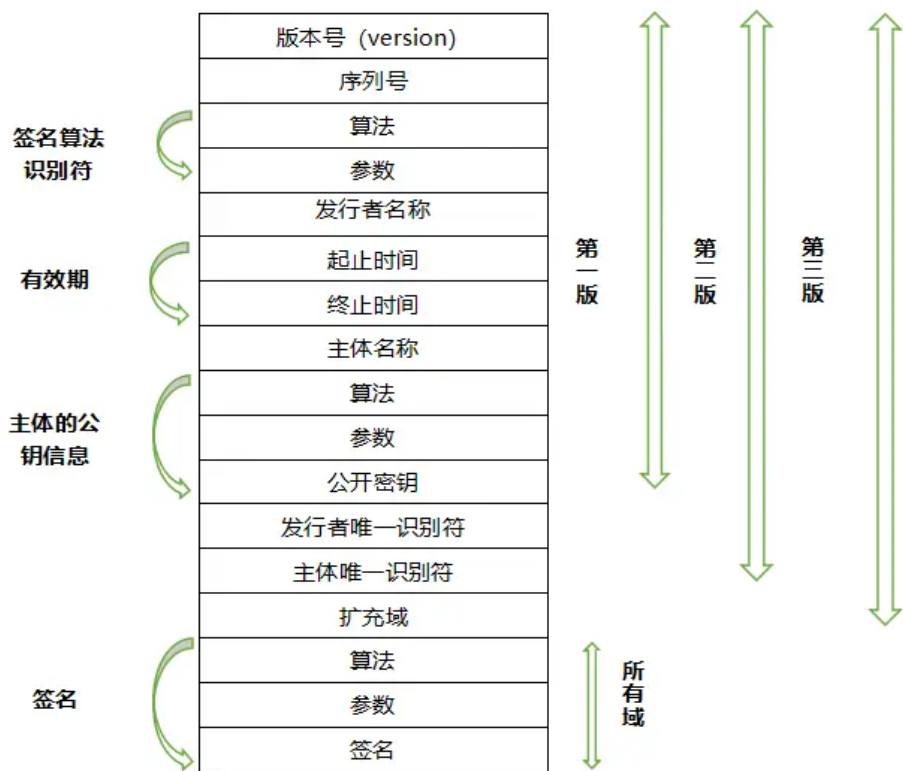
HKDF遵循的主要方法是“提取后扩展”范式，其中KDF在逻辑上由两个模块组成：第一阶段获取输入键控材料并从中“提取”一个固定长度的伪随机键，然后第二阶段将该键“扩展”为几个额外的伪随机键（KDF的输出）。

一般可以用其将通过 Diffie-Hellman 交换的共享密钥转换为适合用于加密、完整性检查或身份验证的密钥材料。

## X.509

In [cryptography](#), **X.509** is an [International Telecommunication Union](#) (ITU) standard defining the format of [public key certificates](#).<sup>[1]</sup> X.509 certificates are used in many Internet protocols, including [TLS/SSL](#), which is the basis for [HTTPS](#),<sup>[2]</sup> the secure protocol for browsing the [web](#). They are also used in offline applications, like [electronic signatures](#).

x.509数字证书的结构图



X.509 证书数字证书结构图

# 加密通讯

## DES--TripleDES

```
class cryptography.hazmat.decrepit.ciphers.TripleDES(key)
```

New in version 43.0.0.

Triple DES (Data Encryption Standard), sometimes referred to as 3DES, is a block cipher standardized by NIST. Triple DES has known crypto-analytic flaws, however none of them currently enable a practical attack. Nonetheless, Triple DES is **not recommended** for new applications because it is incredibly **slow**; old applications should consider moving away from it.

- Parameters:

**key** ([bytes-like](#)) – The secret key. This must be kept secret. Either 64, 128, or 192 [bits](#) long. DES only uses 56, 112, or 168 bits of the key as there is a parity byte in each component of the key. Some writing refers to there being up to three separate keys that are each 56 bits long, they can simply be concatenated to produce the full key.

## 库函数3DES字段

```
#\Lib\site-packages\cryptography\hazmat\primitives\ciphers\algorithms.py
class TripleDES(BlockCipherAlgorithm):
    name = "3DES"
    block_size = 64
    key_sizes = frozenset([64, 128, 192])
```

## 基本参数

密钥长度 `des_key`: 192位随机

工作模式 `mode`: CBC (Cipher-block chaining)

初始向量 `iv`: 8位随机

在des实验中，我们未涉及des对称密钥的建立过程，选择直接存储在内存中

```
#triple-des key
des_key = b'"\xfd\xcf'if+\xe8w\x04\xc7\xdc\x14\xe^>\xc8e\x90\x15\xb0\t\xbeF'
iv=b'\x1d\xb8\xc5\x85n\xc4B\xf7'
cipher = Cipher(TripleDES(des_key), mode=modes.CBC(iv), backend=default_backend())
```

## 加密

首先对字符串进行**编码**，然后利用PKCS7填充算法对消息进行**填充**，最后再对填充后消息进行**加密**

```
#des
data=v.get().encode()
des_encryptor = cipher.encryptor()
padder = PKCS7(algorithms.TripleDES.block_size).padder()
padded_data = padder.update(data) + padder.finalize()
enc_data=des_encryptor.update(padded_data)+des_encryptor.finalize()
```

## 解密

解密和加密过程是相反的。

```
#des
des_decryptor=cipher.decryptor()
info=des_decryptor.update(info)+des_decryptor.finalize()
unpadder = PKCS7(algorithms.TripleDES.block_size).unpadder()
info = unpadder.update(info)+unpadder.finalize()
```

## AES--Fernet

Fernet is built on top of a number of standard cryptographic primitives. Specifically it uses:

- [AES](#) in [CBC](#) mode with a 128-bit key for encryption; using [PKCS7](#) padding.
- [HMAC](#) using [SHA256](#) for authentication.
- Initialization vectors are generated using `os.urandom()`.

[Fernet \(symmetric encryption\) — Cryptography 43.0.0.dev1 documentation](#)

### 初始化

利用以ECC算法密钥协商并扩展的密钥 `associ_key`，我们可以初始化Fernet（AES加密函数）得到fernet实例

```
aes_fernet=Fernet(base64.urlsafe_b64encode(associ_key))
```

### 加密

直接调用fernet的加密函数

```
enc_data=aes_fernet.encrypt(v.get().encode())
```

### 解密

调用fernet的解密函数

```
info = aes_fernet.decrypt(info)
```

## RSA--primitives.asymmetric

[RSA](#) is a [public-key](#) algorithm for encrypting and signing messages.

Generates a new RSA private key. `key_size` describes how many [bits](#) long the key should be. Larger keys provide more security; currently `1024` and below are considered breakable while `2048` or `4096` are reasonable default key sizes for new keys. The `public_exponent` indicates what one mathematical property of the key generation will be. Unless you have a specific reason to do otherwise, you should always [use 65537](#).

```
from cryptography.hazmat.primitives.asymmetric import rsa
private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048,
)
```

## 基本参数

公共参数 `public_exponent`: 65537

私钥长度 `key_size`: 2048

```
#rsa key
private_key= rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048,
)
```

## 加密

利用对方公钥进行加密（加密函数的参数涉及到消息的填充算法）

```
#rsa
enc_data = pk_partner.encrypt(
    v.get().encode(),
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    )
)
```

## 解密

接受方利用自己的私钥解密（与加密函数的参数涉及到消息的填充算法相同）

```
#rsa
info= private_key.decrypt(
    info,
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    )
)
```

## ECC--primitives.asymmetric(for key association)

The Elliptic Curve Diffie-Hellman Key Exchange algorithm standardized in NIST publication 800-56A.

For most applications the `shared_key` should be passed to a key derivation function. This allows mixing of additional information into the key, derivation of multiple keys, and destroys any structure that may be present.

## 生成密钥对

```
#ecc key
ecc_private_key = ec.generate_private_key(ec.SECP384R1())
```

其中, `SECP384R1()` 是NIST规定的一个具有384位素数域的椭圆曲线标准


**secp384r1**

384-bit prime field Weierstrass curve.

Also known as: **P-384 ansip384r1**

$$y^2 \equiv x^3 + ax + b$$

## Parameters

Name	Value
p	0xffe00000000000000000ffff
a	0xffe00000000000000000ffff
b	0xb3312fa7e23ee7e4988e056be3f82d19181d9c6efe8141120314088f5013875ac656398d8a2ed19d2a85c8edd3e
G	(0xaa87ca22be8b05378eb1c71ef320ad746e1d3b628ba79b9859f741e082542a385502f25dbf55296c3a545e3872 0x3617de4a96262c6f5d9e98bf9292dc29f8f41dbd289a147ce9da3113b5f0b8c00a60b1ce1d7e819d7a431d7c90e
n	 0xfffc7634d81f4372ddf581a0db248b0a77aecec196acc
h	0x1

在库函数中实现的对应字段

```
class SECP384R1(EllipticCurve):
    name = "secp384r1"
    key_size = 384
```

## 密钥协商

根据DH协议，利用对方公钥和自己私钥生成共享密钥，并对共享密钥进行派生



```

shared_key = ecc_private_key.exchange(
    ec.ECDH(), ecc_pk_partner)
# Perform key derivation.
associ_key = HKDF(
    algorithm=hashes.SHA256(),
    length=32,
    salt=None,
    info=b'handshake data',
).derive(shared_key)

```

## 身份验证

### 证书验证

实验中我们进行自签名，并在本地进行验证

#### 1. 生成私钥并存储

```

# Generate our key
au_privkey = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048,
)
# Write our key to disk for safe keeping
with open("./pyCryptoChat/info_attached/key2.pem", "wb") as f:
    f.write(au_privkey.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.TraditionalOpenSSL,

        encryption_algorithm=serialization.BestAvailableEncryption(b"passphrase"),
    ))

```

#### 2. 生成证书

```

# Various details about who we are. For a self-signed certificate the
# subject and issuer are always the same.
subject = issuer = x509.Name([
    x509.NameAttribute(NameOID.COUNTRY_NAME, "US"),
    x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, "California"),
    x509.NameAttribute(NameOID.LOCALITY_NAME, "San Francisco"),
    x509.NameAttribute(NameOID.ORGANIZATION_NAME, "My Company"),
    x509.NameAttribute(NameOID.COMMON_NAME, "mysite.com"),
])
cert = x509.CertificateBuilder().subject_name(
    subject
).issuer_name(
    issuer
).public_key(
    au_privkey.public_key()
).serial_number(
    x509.random_serial_number()
).not_valid_before(

```

```

        datetime.datetime.now()
    ).not_valid_after(
        # Our certificate will be valid for 10 days
        datetime.datetime.now() + datetime.timedelta(days=10)
    ).add_extension(
        x509.SubjectAlternativeName([x509.DNSName("localhost")]),
        critical=False,
    # Sign our certificate with our private key
    ).add_extension(
        x509.KeyUsage(
            digital_signature=True,
            key_encipherment=True,
            key_cert_sign=False,
            crl_sign=True,
            content_commitment=False,
            data_encipherment=False,
            key_agreement=False,
            encipher_only=False,
            decipher_only=False
        ),
        critical=True,
    ).add_extension(
        x509.BasicConstraints(ca=False, path_length=None),
        critical=True,
    ).sign(au_privkey, hashes.SHA256())
    # Write our certificate out to disk.
    with open("./pyCryptoChat/info_attached/certificate2.pem", "wb") as f:
        f.write(cert.public_bytes(serialization.Encoding.PEM))

```

### 3. 验证证书

```

with open("./pyCryptoChat/info_attached/certificate1.pem", "rb") as f:
    cert=load_pem_x509_certificate(f.read())
    # Verify the certificate
    try:
        cert.public_key().verify(
            cert.signature,
            cert.tbs_certificate_bytes,
            padding=cert.signature_algorithm_parameters,
            algorithm=cert.signature_hash_algorithm,
        )
        print("Certificate verification successful.")
    except Exception as e:
        print("Certificate verification failed:", e)

```

## 消息验证

在上述加密问题中，可以看到我们有rsa作为加密算法。而由于其非对称性，rsa还可以作为签名算法，在加密使用的公私钥上与rsa加密算法相反

## 签名（先加密后签名）

由于安全性因素，签名密钥一般和加密密钥不同，需要区别处理。这里我们对密文进行签名后，密文和签名一并发送。

```
#sign the enc_messages
signature = private_key.sign(
    enc_data,
    padding.PSS(
        mgf=padding.MGF1(hashes.SHA256()),
        salt_length=padding.PSS.MAX_LENGTH
    ),
    hashes.SHA256()
)
msg=enc_data+b"signatureofenc_data"+signature
s2.send(msg)
```

## 验证

接受方通过公钥验证对密文的签名。验证失败 `verify()` 会产生异常。

```
#extinguish the received data
data=sock.recv(4096)
info, signature = data.split(b"signatureofenc_data")
#verify the signature
pk_partner.verify(
    signature,
    info,
    padding.PSS(
        mgf=padding.MGF1(hashes.SHA256()),
        salt_length=padding.PSS.MAX_LENGTH
    ),
    hashes.SHA256()
)
```

## 实验结果

---

建立连接，选择加密方式



