实验报告（数字签名）

【实验目的】

1、 了解并掌握常用的数字签名方法并编程实现；
2、 感受不同数字签名方案的优缺点及适用场景。

【实验环境】

Python 3.9.7 64-bit

一、ElGamal 数字签名方案

1、算法流程

算法原理：

ElGamal 数字签名方案与 ElGamal 公钥加密方案类似，只不过是用私钥进行加密，公钥进行解密的，该方案利用了数论中原根的性质：对于素数q，如果α是q的原根，则有$\alpha,\alpha^2,\cdots,\alpha^{q-1}$取模后各不相同。ElGamal 数字签名方案的参数的素数q和α，其中α是q的原根。方案在计算签名和验证签名时需要用到安全 Hash 算法（SHA256）计算消息的 Hash 值$m=H(M)$，其中$0\leqslant m\leqslant q-1$。

伪代码：

算法1：ElGamal计算签名

输入：消息的Hash值m, 私钥X_A, 随机数K
输出：消息签名(S_1,S_2)

```
function sign(m, X_A, K)
    S_1 ← α^K mod q
    S_2 ← [K^{-1}(m − X_A S_1)] mod (q − 1)
    return (S_1, S_2)
end function
```

算法2：ElGamal验证签名

输入：消息的Hash值m, 公钥Y_A, 签名(S_1,S_2)
输出：签名合法$(True)$或不合法$(False)$

```
function verify(m, Y_A, (S_1, S_2))
    V_1 ← α^m mod q
    V_2 ← Y_A^{S_1} · S_1^{S_2} mod q
    if V_1 = V_2 then
        return True
        else
            return False
    end if
end function
```

流程图：

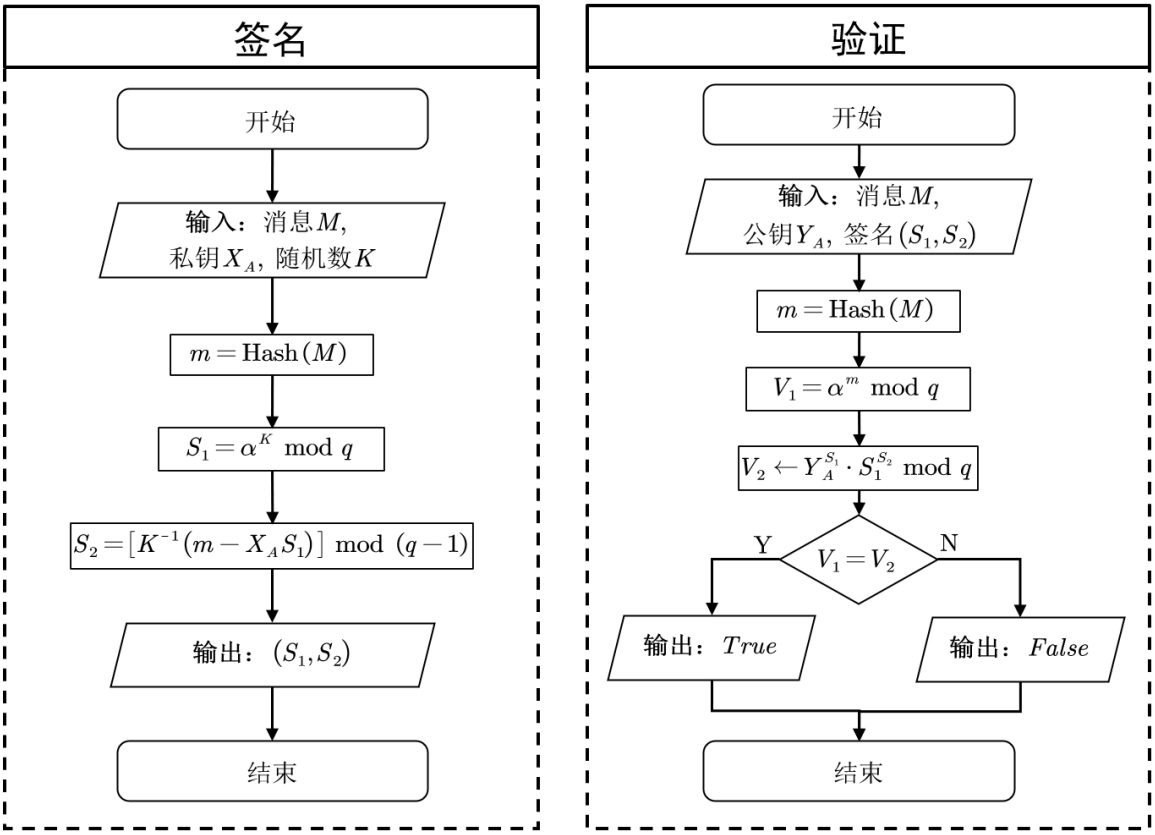

图 1 ElGamal 数字签名流程

函数调用关系图如下：

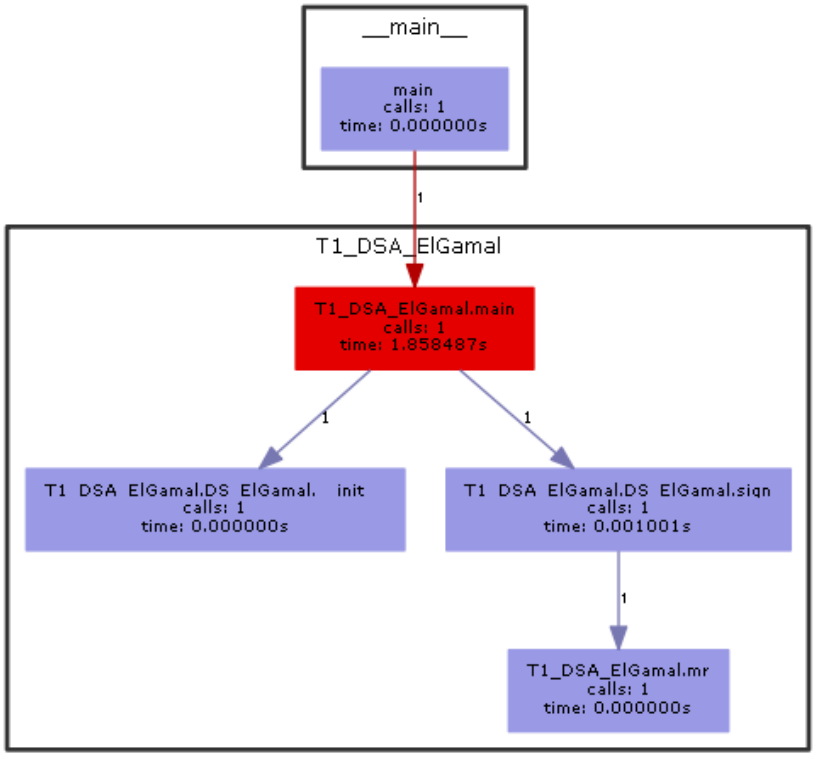

图 2 ElGamal 数字签名函数调用关系（签名样例）

2、测试样例及结果截图

```
831405185079551754106024075111536077567801699902164413420827763748903920817478392920438247434999773335902924584146404475752662517131542309831954456089299
是 谁 憋 疯 了
Sign
328184395775097434154144979187402537428854741120972941162970079238285513103683215949163395229447844352642615901661628862884291578166401885909593995794546116146932291399580649737566118173788796661188561970209856546716335210952273513994268068902156078706896505127053943319299856343991991156725344969188151374610
Process finished with exit code 0
```

图 3 ElGamal 数字签名测试样例——签名

```
782613910964314550208749953517170281184613193083126153226620598866189016804477709861377311183183907889314655619888955311590043078113949759286907221174250810
你所热爱的，就是你的绿园
Vrfy
945052293383388638901707582500514723736198483111928463183762853329398323477744931080444647463836513156750531196660114448817941593225748204974753002862498131029584072256029821607655552421763274783742458932563239954076494002721116821
True
Process finished with exit code 0
```

图 4 ElGamal 数字签名测试样例——验证

3、讨论与思考

- 由于需要频繁使用参数q, α，将 ElGamal 数字签名封装成了 class。
- 但实际上没啥必要，后面将用全局变量 global 代替。
- 为了代码的简洁程度，最后提交的代码没用之前写的快速模幂的函数，直接调用 Python 的 pow，可以看出，快速模幂对于运算效率的提高还是比较明显的。

评测编号	提交时间	提交状态	代码语言	最大运行时间	最大运行内存	详细信息
26017	2022-06-02 15:12:02	Accepted	Python	33ms	10992KB	
25479	2022-06-01 15:04:28	Accepted	Python	19ms	10928KB	

图 5 用内置的 pow（上） & 用之前写的快速模幂（下） 运行时间对比

安全性：

- Elgamal 数字签名在实际使用中存在一些限制，如会话密钥K的值必须随机选取，且必须确保在签不同的信息时会话密钥没有重复使用过，必须避免选到弱随机数 2 或者 3。

二、Schnorr 数字签名方案

1、算法流程

算法原理：

Schnorr 数字签名方案也是基于离散对数的，其主要优势是可以将生成签名所需的消息计算量最小化，生成签名的主要步骤不依赖于消息，可以在处理器空闲时执行，生成与消息有关的部分只需要进行一个$2n$位的整数与n位的整数相乘。

该方案基于素数模p，且$p-1$包含大素数因子q，即$p-1\equiv 0 \pmod q$。一般取$p\approx 2^{1024}$和$q\approx 2^{160}$，即p为 1024 位整数，q为 160 位整数，也正好等于 SHA-1 中 Hash 值的长度。通常选择 SHA-1 位内部 Hash 算法。由于 DSA 公开参数的选择与 Schnorr 签名方案完全一样，我们可以直接使用 FIPS 186 标准中的参数生成算法。

函数调用关系图：

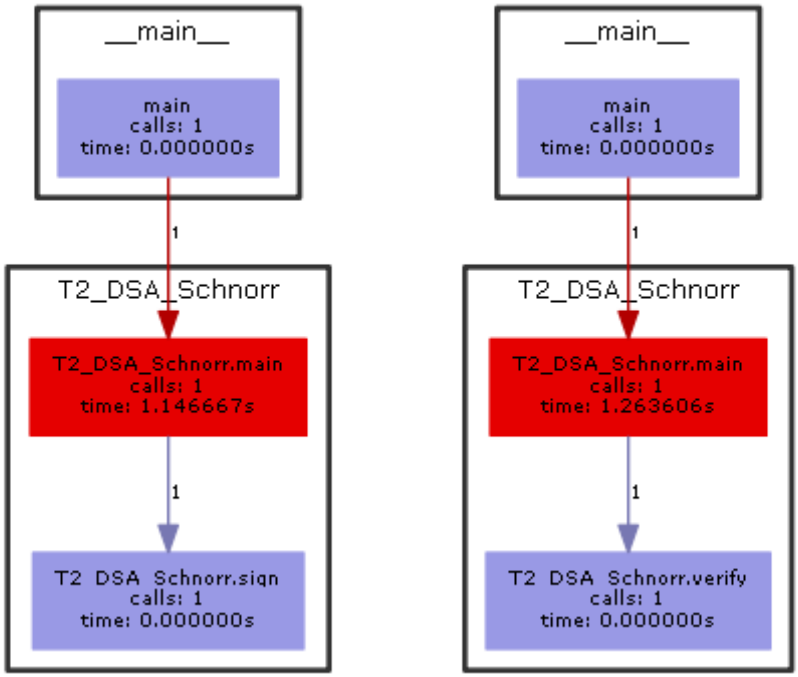

图 6 函数调用关系图（左：签名样例 右：验证样例）

伪代码：

算法3：Schnorr计算签名

输入：消息M, 私钥s, 随机数r
输出：消息签名(e,y)

1. **function** $\text{sign}(M,s,r)$
2. $x\leftarrow \alpha^r \bmod p$
3. $e\leftarrow H(M||x)$
4. $y\leftarrow (r+s\cdot e) \bmod q$
5. **return** (e,y)
6. **end function**

算法4：Schnorr验证签名

输入：消息M, 公钥v, 签名(e,y)
输出：签名合法$(True)$或不合法$(False)$

```
1.  function verify(m, Y_A, (S_1, S_2))
2.      x' ← α^y v^e mod p
3.      if e = H(M||x') then
4.          return True
5.          else
6.              return False
7.      end if
8.  end function
```

流程图：

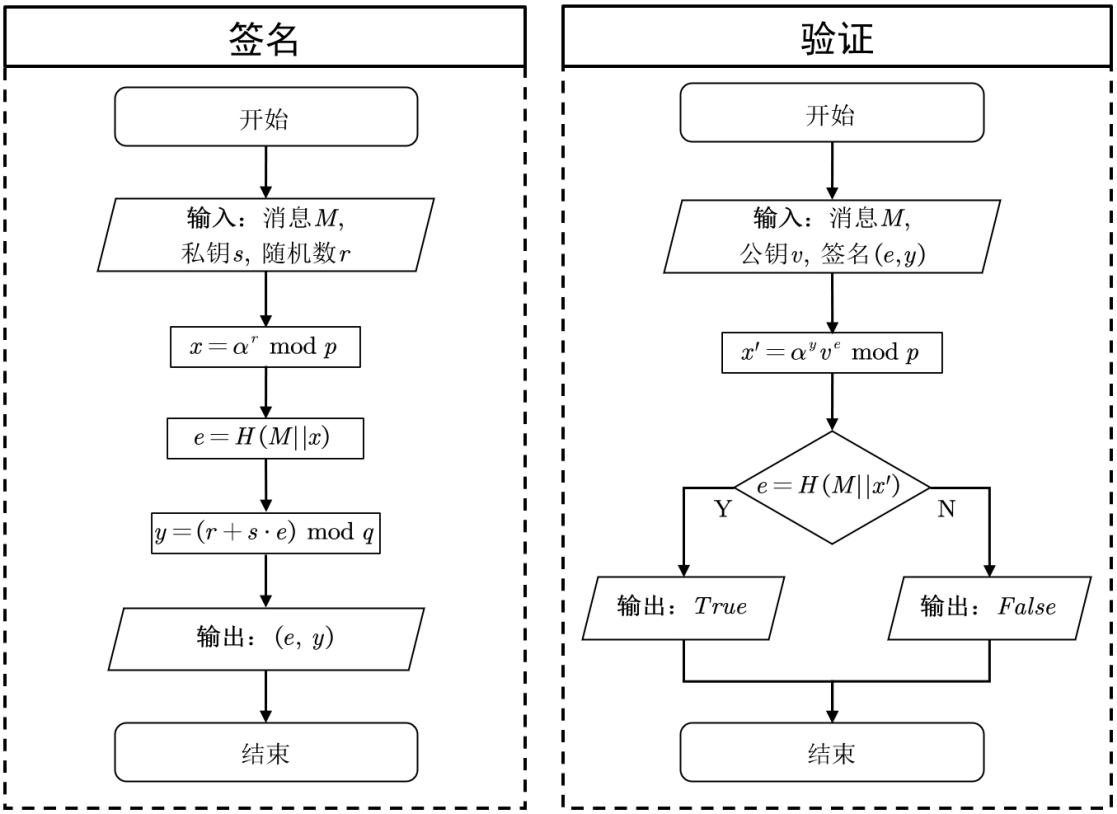

图 7 Schnorr 数字签名流程

2、测试样例及结果截图

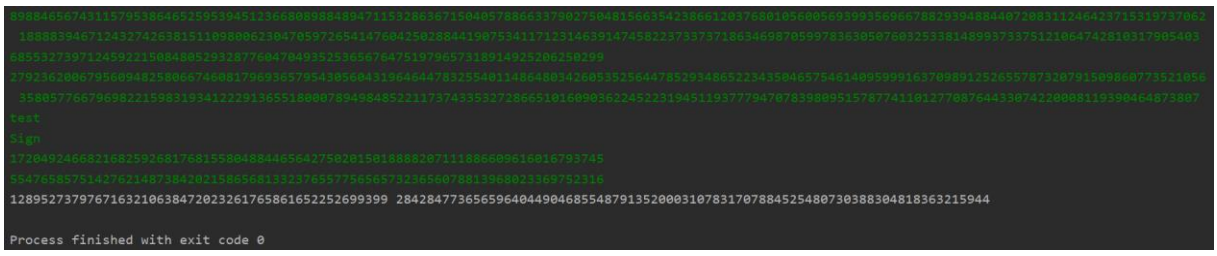

图 8 Schnorr 数字签名测试样例——签名

图 9 Schnorr 数字签名测试样例——验证

3、讨论与思考

- 用了全局变量 global 避免多次输入函数自变量。

Schnorr **签名算法优点：**

- Schnorr 方案将生成签名所需的消息计算量最小化。生成签名的主要工作不依赖于消息，可以在处理器空闲时执行。
- Schnorr 签名算法有可证明安全性。
- Schnorr 签名算法具有不可延展性，第三方无法在不知道私钥的情况下，针对某一公钥和消息的有效签名，改造成针对该公钥和信息的另一有效签名。
- Schnorr 签名算法是线性的，使得多个合作方能生成对他们的公钥之和也有效的签名。

三、SM2 数字签名方案

1、算法流程

算法原理：

利用素域$GF(p)$或$GF(2^m)$域上的椭圆曲线都可以构成椭圆曲线数字签名方案，基于椭圆曲线的数字签名方案有着安全、密钥短、软硬件实现更优等特点，NIST 发布的数字签名标准（DSS）的较新版 FIPS 186-3 中已引入基于椭圆曲线的数字签名算法，我国也于 2012 年颁布了椭圆曲线数字签名标准 SM2。SM2 标准中规定了唯一的椭圆曲线，并给出了参数，在计算签名和验证签名时候需要用到安全 Hash 算法，本节中我们针对签名算法和验证算法进行了实现。

除了椭圆曲线的系统参数，我们还需要给出用户 A 的密钥对，包括私钥d_A和公钥$P_A=[d_A]\cdot G=(x_A,y_A)$。以及签名者的一个长度为$entlen_A$比特的可辨别标识$ID_A$，将整数$entlen_A$用两个字节表示，记作$ENTL_A$，首先要计算用户 A 的 Hash 值$Z_A$，其值为

$$Z_A=H_{256}(ENTL_A||ID_A||a||b||x_G||y_G||x_A||y_A)$$

本题中 Hash 函数采用 SM3。

函数调用关系图：

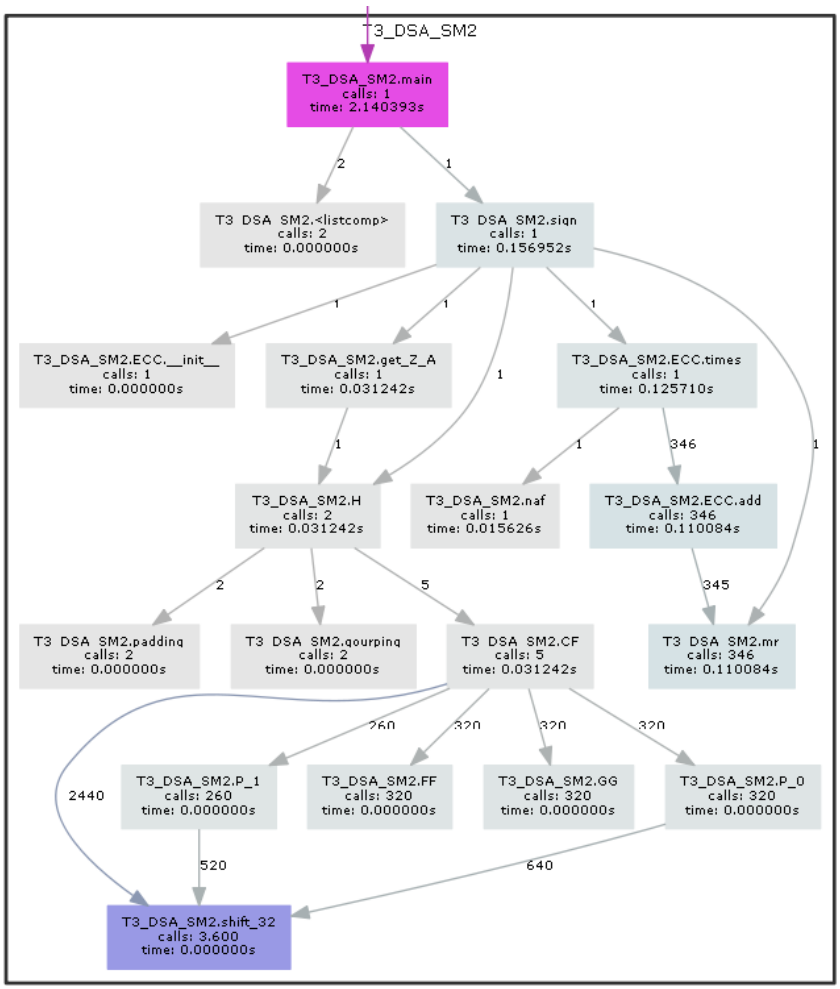

图 10　函数调用关系图（签名样例）

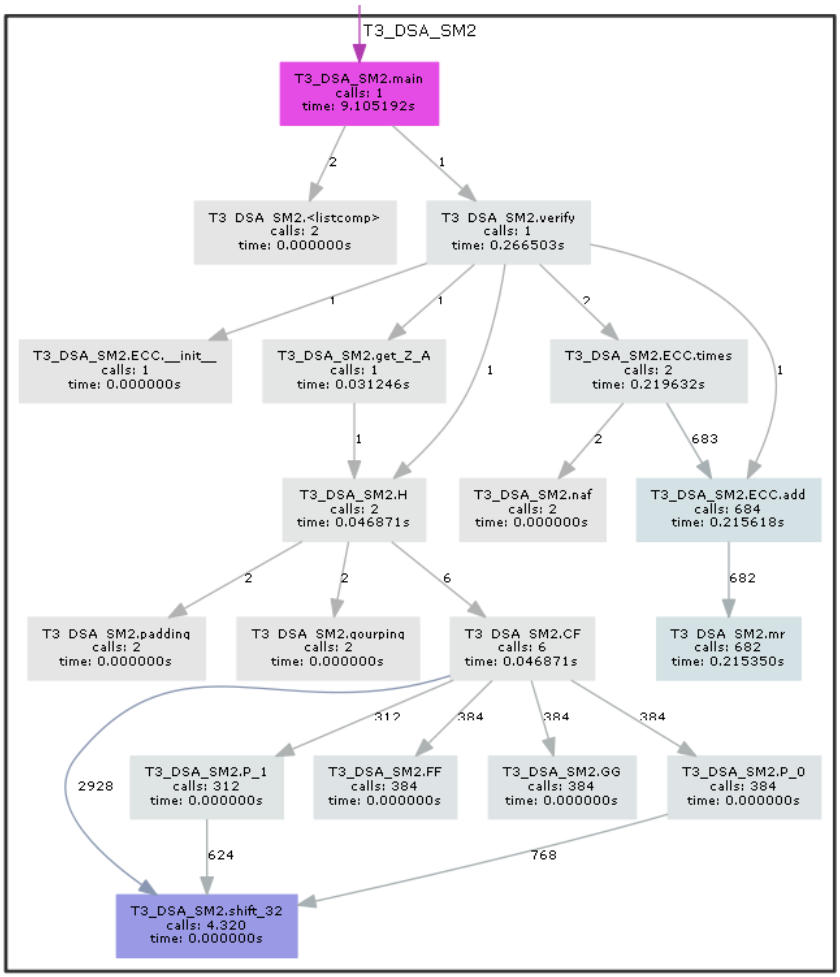

图 11 函数调用关系图（验证样例）

2、测试样例及结果截图

```
6027570200924509638568617151521989641629712149940225095553785768388554194118754492052985589574080443685629857027481671841726313362585597978545915325572248451831853936081346014255069855018812318761355191033760963918538733704700980742990551425407836123641846908047770823434349966291667120909283832980018022508529405937379755419157903904478921572542806770830401260612308519640632340013146027570200924509638568617151521989641591964469845342405556166525133029628152741ALICE123@YAHOO.COM4927346340877997421592888003129352901369751434954921663604743238822873158794560907753313590753025460164147405799146121926495834596450107501082600869008236message digestSign838755194778401207140007147159631205354287074082149471312072617733306092400349165263701565432377505549247848435858362931747789390865593867043744446085487293754636895866940044417977668126984755739382563637800894258018470594425215535055807175413403780973844050746030729265458324116628415789532724189798694397 5

Process finished with exit code 0
```

图 12 SM2 数字签名测试样例——签名

```
6027570200924509638568617151521989641629712149940225095553785768388554194118754492052985589574080443685629857027481671841726313362585597978545915325572248451831853936081346014255069855018812318761355191033760963918538733704700980742990551425407836123641846908047770823434349966291667120909283832980018022508529405937379755419157903904478921572542806770830401260612308519640632340013146027570200924509638568617151521989641591964469845342405556166525133029628152713neverGonnaGive@You.up219814080649322261353012027715617621433359852819130558804271704563304668913492802858928398040344749450431090607460809047118036824973441031971313869224999513never gonna let you downVrfy4806344975560987687853229279905938965304711838081468045273127175601881095840012071032352070378296001266231648972411992535359641329464973392771258376729799True

Process finished with exit code 0
```

图 13 SM2 数字签名测试样例——验证

3、讨论与思考

- 按理说应该用多文件，SM3 直接调就行；为了保证本地文件的完整性和规范性，我将 SM3 又写进了程序里。
- 虽然每部分的代码都写得比较规范，但整体上来看是有些冗余的，按理说也应该将 SM3 封装成一个类，但属实是又没必要。
- 出题的助教说多文件万岁，虽然看起来合理，但对于一向对文件整理有强迫症的同学，这是必然不可能的（为什么？因为与其将文件扯来扯去，不如直接把需要的代码块 copy 过来）。
- 为了方便 debug，不妨在拼接时先全部转成 hex 字符串，最后再 bytes.fromhex( )转成字节（字节转 hex 字符串可以直接.hex( )）。

SM2 数字签名优势：

SM2 基于 ECC，同等强度下其密钥更短，因此签名速度快。同等安全强度下，SM2 算法在用私钥签名时，速度远超 RSA 算法。

四、【选做】RSA-PSS 数字签名算法

1、算法流程

算法介绍：

RSA 概率签名方案（RSA-PSS）是基于 RSA 的一种签名方案，被 RSA 实验室推荐为 RSA 各类方案中最安全的一种。RSA-PSS 数字签名算法包括密钥生成算法（实验中未涉及）、消息编码算法、签名算法和验证算法。RSA-PSS 算法的参数和函数如下：

表 1 RSA-PSS 函数、参数表

类 别	内 容	说 明
选项	Hash	Hash 函数。实验选取 SHA-1，输出字节长度$hLen=20$
	MGF	掩码生成函数，目前使用MGF1
	$sLen$	$salt$ 的字节长度，一般$sLen=hLen$。当前版本$sLen=20$
输入	M	消息
	$emBits$	编码消息EM的比特长度，比 RSA 模数n的长度小
输出	EM	编码后的消息，用于加密后形成消息签名的消息摘要
参数	$emLen$	编码消息EM的字节长度，$emLen=\lceil emBits \div 8 \rceil$
	$padding_1$	8 字节的 00：$(00\ 00\ 00\ 00\ 00\ 00\ 00\ 00)_{16}$
	$padding_2$	$[(emLen-sLen-hLen-2)$字节的$00] \mid\mid 01$
	$salt$	盐，一组伪随机数
	bc	‘BC’ 的十六进制值

函数调用关系：

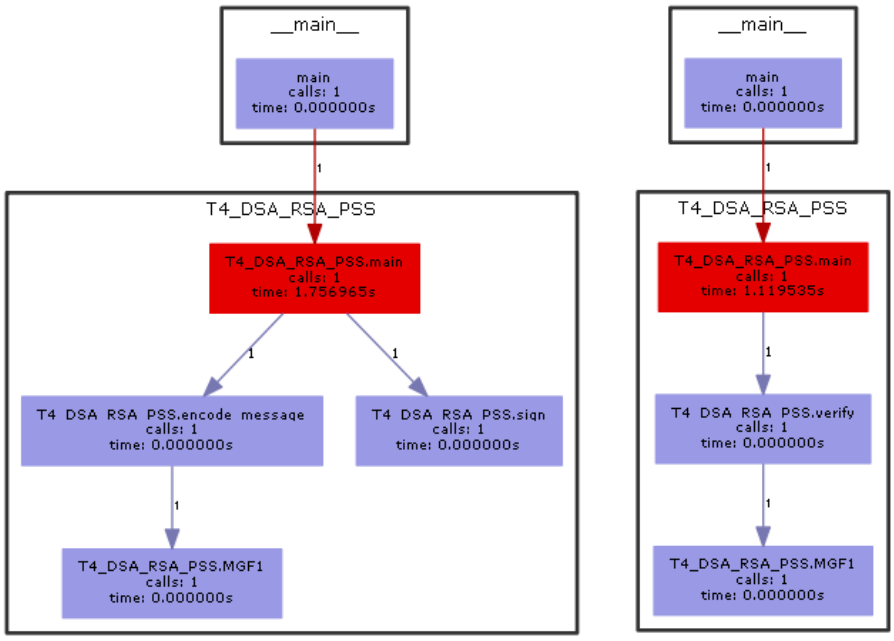

图 14 函数调用关系图（左：签名 右：验证）

伪代码：

算法5：掩码产生函数MGF1

输入：被掩码的字节串X, 掩码的字节长度$maskLen$
输出：掩码$mask$

1. **function** MGF1$(X, maskLen)$
2. **set** T // 初始化T为空字节串
3. $k \leftarrow \lceil maskLen \div hLen \rceil - 1$
4. **for** $counter \leftarrow 0$ **to** k **then**
5. 将$counter$表示为4字节的C
6. $T \leftarrow T \mid\mid \text{Hash}(X \mid\mid C)$
7. **end for**
8. $mask \leftarrow T$的前$maskLen$字节
9. **return** $mask$
10. **end function**

算法6：消息编码算法

输入：待编码消息M, 盐(伪随机数)$salt$, EM比特长度$emBits$
输出：编码后消息EM, 用于加密后形成消息签名的消息摘要

1. **function** encode_message$(M, salt, emBits)$
2. $emLen \leftarrow \lceil emBits \div 8 \rceil$
3. **set** $padding_1 \leftarrow 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00$
4. **set** $padding_2 \leftarrow 00 \times (emLen - sLen - hLen - 2)\ 01$
5. $mHash \leftarrow \text{Hash}(M)$
6. $M' \leftarrow padding_1 \mid\mid mHash \mid\mid salt$
7. $H \leftarrow \text{Hash}(M')$
8. $DB \leftarrow padding_2 \mid\mid salt$
9. $DBmask \leftarrow \text{MGF}(H, emLen - hLen - 1)$
10. $maskedDB \leftarrow DB \oplus DBmask$
11. $EM \leftarrow maskedDB \mid\mid H \mid\mid \text{bc}$
12. **return** EM
13. **end function**

算法7：签名算法

输入：编码后的消息EM, 私钥d, RSA模数n
输出：编码后消息EM, 用于加密后形成消息签名的消息摘要

1. **function** sign(EM, d, n)
2. 将字符串EM作为无符号的非负二进制整数m
3. $s \leftarrow m^d \bmod n$
4. $k \leftarrow n$的字节长度 // 本题n固定为1024比特，即128字节
5. 将s转换成k字节的串S
6. **return** S
7. **end function**

算法8：验证算法

输入：消息M, 签名S, 公钥e, 模数n, EM比特长度$emBits$
输出：签名合法$(True)$或不合法$(False)$

function verify$(M,S,e,n,emBits)$
    将16进制的签名字符串S转化为整数s
    $emLen \leftarrow \lceil emBits \div 8 \rceil$
    $m \leftarrow s^e \bmod n$
    将整数m转换成$emLen$的字节串EM　// 恢复消息编码
    $maskedDB \mid\mid H \mid\mid \text{bc} \leftarrow EM$　// 取出对应位即为$maskedDB$, H
    $mHash \leftarrow \text{Hash}(M)$
    $DBmask \leftarrow \text{MGF}(H, emLen - hLen - 1)$
    $DB \leftarrow maskedDB \oplus DBmask$
    $M' \leftarrow padding_1 \mid\mid mHash \mid\mid salt$
    $H' \leftarrow \text{Hash}(M')$
    if $H' = H$ **then**
        return $True$
        else
            return $False$
    end if
end function

流程图：

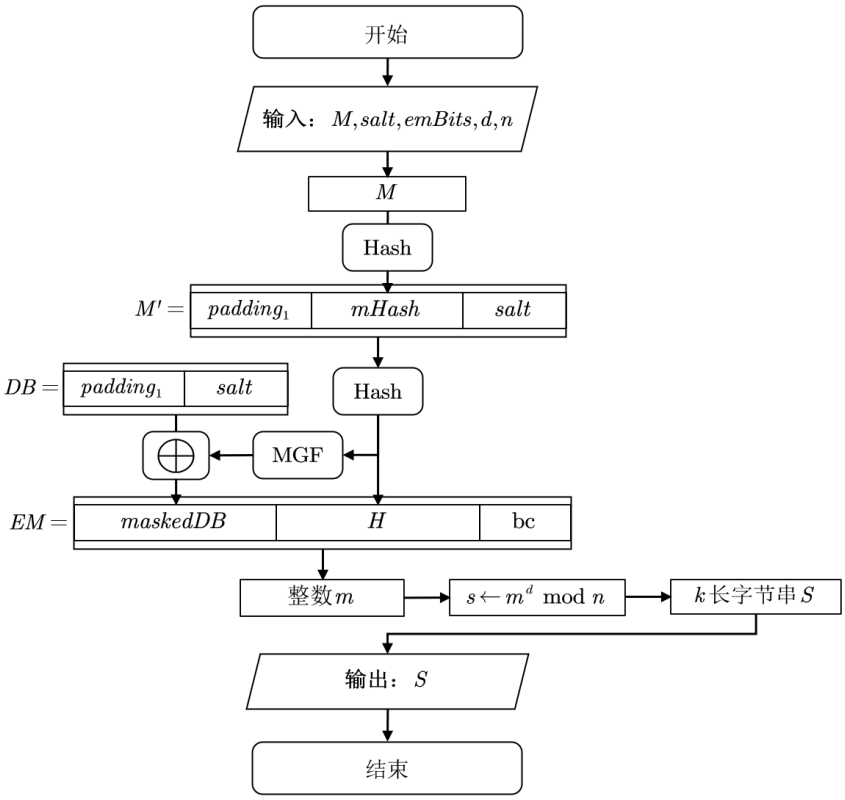

图 15　RSA-PSS 签名流程

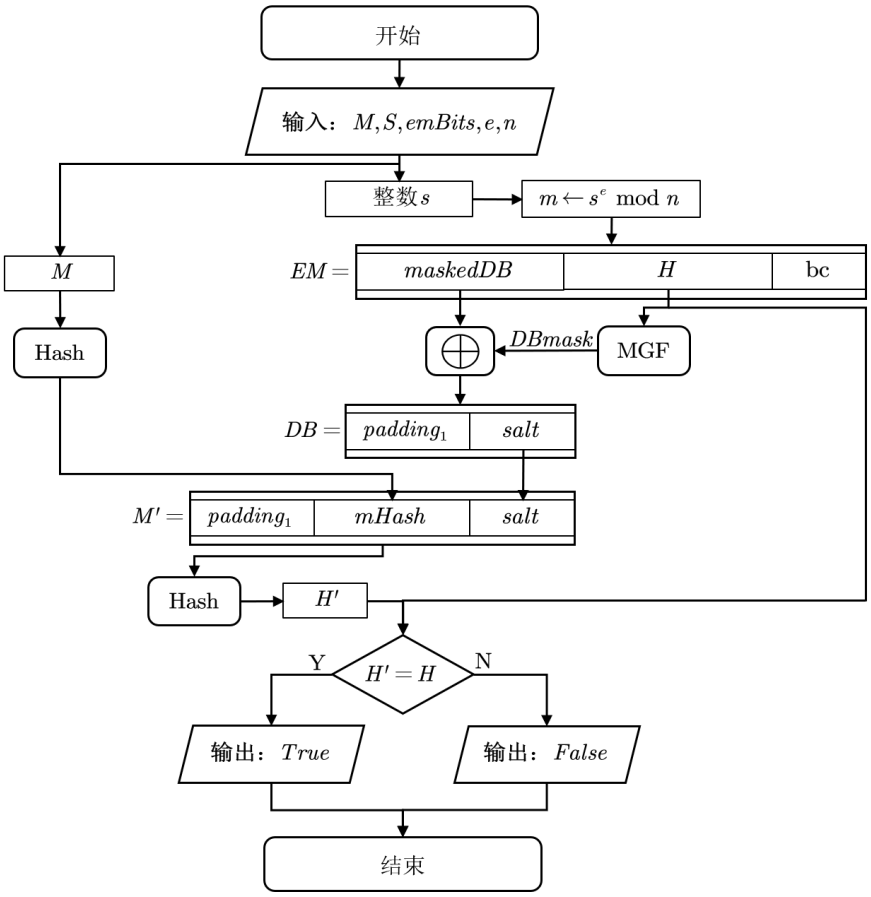

图 16 RSA-PSS 验证流程

2、测试样例及结果截图

```
0523冲突事件后续诉求处理进展https://temp.roycent.cn
14739253690918193643764968753627523660839711181086691196103132140635720094914949749087482991898329078996
 3713720012525170038437097493163076328008025719473276301079739164714793313287061056821305985932767583558
 7174560833398353831638152962270358674121437398154032279494404983832474409623108805045615535553060087179
96
Sign
14285145826411277026259652009081662753944317422425430689691366722468264974037157357330381537389131630420
 3750125881889645262660699584408938152189567219539376806703195857109568944402128751354374513661514717314
 1628205648136202913763465584871206788552076897754666704835587414644109160276016068455175385383216015234
4758a1c40f6ead39d8281779d63e0bd5377f19a5
a5b452b0a2fc78d4c1c04af0067c3e4e890d4bfddc1642bcdcfa8b30b98d2debde6ee1239907e18d19f8edf1394b5997962836ee
 4904b740b6a71e6b6df821fb572791b12a95c68a7537c014bca4cd0e7a2a67559f7c13bbacda841fc8c1f9eda8ee36478d42d4f
 4f15e958d0dac00695a515083d81cc6fccf91fc210b9b58b7

Process finished with exit code 0
```

图 18 RSA-PSS 签名样例

```
这是一个正经的测试点
10598965578772786787478831010427641858172860556166544145689090980283917890345617696922235117174698859243
 5407067984662090109769588473021303915685394978038077688926263034160014542380069428280082820211043832262
 6205490005874416228922566181219524163947347845484624714385925447950144349168209782213928101841611510431
023
Vrfy
32903811767573058993775407658279372554462447818880720216377984749260878509799520087409508083277983480631
 1350873193507512864955907154653912515714116754076
73bdd78b946d4c9097d8147beceed190466bb9bf70916eef0b2f72ff10496cf3bbbe5f8dccadaeec76341822cd4f41167d740756
 a1ed4d768d83482686a628dc15d67fdde1d6a156fecb4ecba9b9a3c0883698d03ca48060b7683410b2dfeab6b6fe2448f227b9e
 d54cb990748295b65eed9d4091833e533e6aff6d005b06058
True

Process finished with exit code 0
```

图 18 RSA-PSS 验证样例

3、讨论与思考

安全性：

签名算法中，RSA-PSS 方案与其他基于 RSA 的方案相比，该方案使用了随机化的处理过程，能够证明其安全性与 RSA 算法的安全性相关。因为每次使用时盐的值都不同，所以使用相同的私钥对相同的消息进行两次签名，将得到两个不同的结果，这增加了签名方案的安全度。

RSA 的填充模式：

- PSS 是 RSA 的填充模式中的一种。
- 完整的 RSA 的填充模式包括：
  - RSA_SSLV23_PADDING（SSLv23 填充）
  - RSA_NO_PADDING（不填充）
  - RSA_PKCS1_OAEP_PADDING（RSAES-OAEP 填充，强制使用 SHA1，加密使用）
  - RSA_X931_PADDING（X9.31 填充，签名使用）
  - RSA_PKCS1_PSS_PADDING（RSASSA-PSS 填充，签名使用）
  - RSA_PKCS1_PADDING（RSAES-PKCS1-v1_5/RSASSA-PKCS1-v1_5 填充，签名可使用）
- 其中主流的填充模式是 PKCS1 和 PSS 模式。
- PSS 的优缺点如下：
  - PKCS#1 v1.5 比较简易实现，但是缺少 Security Proof。
  - PSS 更安全，所以新版的 openssl-1.1.x 优先使用 PSS 进行私钥签名（具体在 ssl 握手的 server key exchange 阶段）

【收获与建议】

1、收获

- ~~加深了对数字签名的理解。~~
- ~~提高了对 Python 的熟练度。~~
- ~~巩固了排版能力。~~
- 收获了劳累与繁忙。

通过本次实验，我了解并掌握了常用的数字签名方法并编程实现，感受到不同数字签名方案的优缺点及适用场景。数字签名（又称公钥数字签名）是只有信息的发送者才能产生的别人无法伪造的一段数字串，这段数字串同时也是对信息的发送者发送信息真实性的一个有效证明。它是一种类似写在纸上的普通的物理签名，但是使用了公钥加密领域的技术来实现的，用于鉴别数字信息的方法。一套数字签名通常定义两种互补的运算，一个用于签名，另一个用于验证。数字签名是非对称密钥加密技术与数字摘要技术的应用。

2、建议

- 无。

【思考题】

- 无