**密码学基础课程项目**

**EC schnorr 签名的实现**

# 实验背景

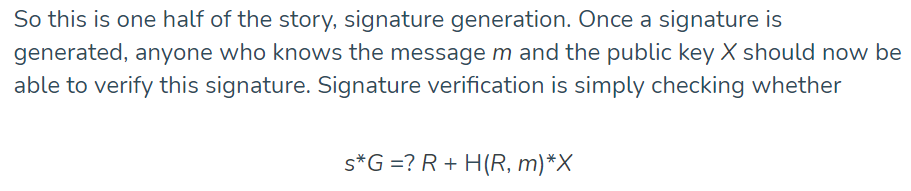
## 实验目的

1. 理解schnorr签名的过程
2. 掌握OPENSSL关于椭圆曲线的函数
3. 了解bitcoin源码中schnorr签名的实现方式及其原理
4. 了解密码学工程中一些技术

## 1.2 实验原理

### 1.2.1 ec shcnorr签名的定义：

Definition of Schnorr Signatures 
If you have a key pair (x, X = x*G), then a Schnorr signature Of a message m is the pair (R, s) where k is a 
random private key and 
R = k*G 



*图1：schnorr签名的定义[1]*

根据老师的ppt《零知识签名》一章的循环群上的schnorr签名的定义，和图1，可以延伸出EC上的schnorr签名定义：

**Key generation：**

随机生成私钥：x

计算出公钥X = x\*G

**Sign：**

随机生成k

签名方取得私钥x和待签名的消息m

R = k\*G（R通常称为nonce*[1]*）

e = H(R||m)

s = k+e\*x = k+H(R||m)\*x

输出签名为（R，s），—— 这是bitcoin所使用的签名方式

In Bitcoin, this pair (*R*, *s*) is encoded using 64 bytes where the first 32 bytes represent *R* and the last 32 bytes represent *s*. *[1]*

**Verification：**

接收方取得公钥X = x\*G和签名（R,s）,以及原消息m

计算

e = H(R||m)

如果s\*G = R+e\*X，（R = s\*G+（-e）\*X）输出1，表示验签成功

否则输出0，表示验签失败

### 1.2.2 Batch Verification的原理：

为了提高效率，还可以进行Batch Verification.这是schnorr签名比起ECDSA的一大优点.

*”standardized cannot be verified more efficiently in batch compared to individually, unless additional witness data is added.  “[2]（具体的批验证的必要性和实例也可以参考该文献）。*

1. **实现原理：**

对于一系列的签名tuples（假设总共为n个）：

Xi为公钥

mi为原文消息

（Ri,si）如上面EC schnorr的定义，是一个签名

则输入的样式为：[(X1,m1,(R1,s1)),…(Xn,mn,(Rn,sn))]

如果按照单个来验证，即把上述的Verification对每个元组都运行一次，总共是n次，代价很大。所以我们采用批验证的方式：

1.the verifier generates *n* random numbers *a1*, …, *an*, computes the *n* challenge hashes *ei* = H(*Xi*,*Ri*, *mi*)

2. The verifier checks if

(*a1s1* + *a2s2* + … + *ansn*)\**G* ?= *a1*\**R1* + *a2*\**R2* + … + *an*\**Rn* + (*a1e1*)\**X1* + (*a2e2*)\* *X 2* + … + (*anen*)\* *X n*

[2]

1. **正确性：**

之所以要随机生成ai，是为了避免invalid signature 也被加进来比如： if (*R*, *s*) and (*R’*, *s’*) are both valid signatures and (*R*, *s’*) and (*R’*, *s*) are not, then this latter pair of signatures would still pass batch validation.

1. **进一步优化：**

Batch Verification 的第二步还可以进一步优化：

*2. The verifier checks if 0* ?= *a1*\**R1* + … + *an*\**Rn* + (*a1e1*)\**X 1* + … + (*anen*)\*X*n* – (*a1s1* + … + *ansn*)\**G*

这样更便于高效的连续乘计算

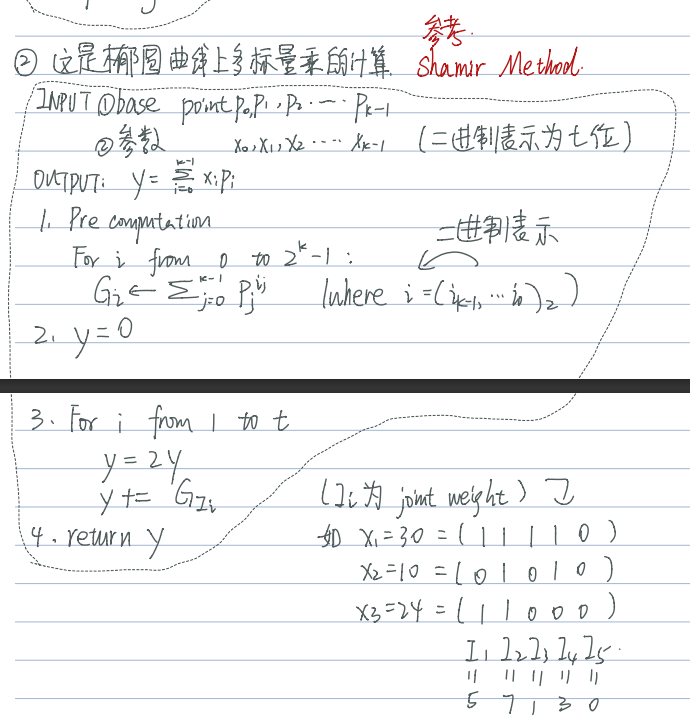
### 1.2.3 ec上的连续同时乘

在群上：根据课程PPT《公钥密码数学基础》，ge0\*ge1\*…\*gek-1这样的计算是非常耗时的。

为了进行高效的ge0\*ge1\*…\*gek-1的计算，我们可以使用（**simultaneous multiple exponentiation**）算法。

在椭圆曲线上该算法称为多标量乘法。可以参考这些[3]，[4]文献进行学习。

在本课程的作业中，也给出了这个描述：



图：课程作业

### 1.2.4 ec上的点的压缩

Fp群上，椭圆曲线上一个横坐标对应2个纵坐标，因此我们在传输的时候可以传输x坐标，然后再传输1个bit（0或1）分别表示y为偶数和奇数——对应椭圆曲线上的两个点。这样可以对签名（R,s）中的R进行压缩，也可以在传输公钥的时候对公钥进行压缩。

在bitcoin中，采用的不是这个压缩方法，它采用的是X-only public key[5]，即只使用点的x坐标，一个公钥可以对应2个私钥，这样效率更加高。

我开始根据这个方法来设计，发现结果只有在公钥的y值为even的时候，验签的结果才正确，这其中应该还有对应的其他的设计，由于时间关系我就直接改用加上1bit值来表示y值的even或者odd进行算法的设计了。

### 1.2.5 密码学工程

测信道攻击（side channel attack）核心思想是对加密软件硬件系统运行时产生的信息进行分析以获取密文信息。if分支这样的结构会导致执行时间不一样，在实际机器中可能体现为泄露的辐射强度不一样，敌手根据这个信息就可能取得密文信息。

于是在本次pj里面我的密钥生成，签名函数和验签函数等关键函数都没有使用if结构。

# 实验内容

1. 实现Schnoor签名，包含密钥生成、签名、验签、批验签四个函数。

2. 代码实现必须是constant-time的，不要出现安全问题，如分支结构这类无法抗侧信道攻击的实现；

3. 可使用OpenSSL提供的函数，进行C/C++实现。

# 实验环境

windows10 操作系统

visual studio2019

openssl的开发环境参考此文献[6]

# 实验步骤

## 4.1密钥生成

### 4.1.1 结构体

* 说明

采用OPENSSL的EC\_KEY结构体来表示密钥。

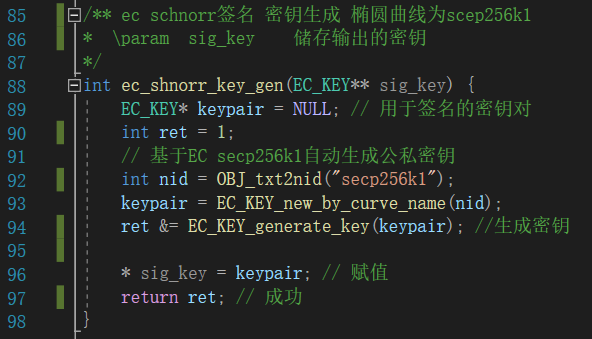
* 截图

287 
288 
289 
291 
292 
293 
294 
295 
296 
297 
298 
299 
3ee 
3e2 
3e3 
3e4 
ses 
zes 
se7 
v struct ec_key_st { 
const EC KEY METHOD *meth; 
ENGINE •engine; 
int version; 
EC_CROUP *group; 
EC_POINT 
BIGNUM 
unsigned int enc _ flag; 
conv_form; 
CRYPTO REF COUNT references; 
int flags; 
v #ifndef FIPS MODULE 
ex_data; 
#endif 
*lock; 
OSSL LIB CTX *libctx; 
char *propq; 
Provider data 
size_t dirty_cnt; If any key material changes, 
increment this s/ 

[openssl-master\crypto\ec\ec\_local.h]

### 密钥生成函数

* 截图



[ec\_schnorr.cpp]

* 说明

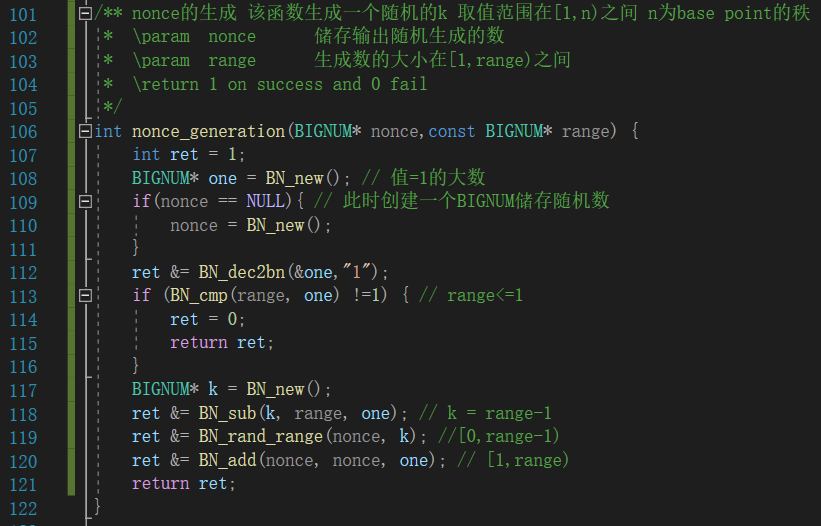
使用了OPENSSL库，将结果写入sig\_key中。

截图的测试使用的是secp256k1椭圆曲线，即bitcoin所用的曲线，之后还会对其他类型的椭圆曲线进行测试。

## 4.2 签名

### 4.2.1 nonce的生成

* 函数截图



[ec\_schnorr.cpp]

* 说明

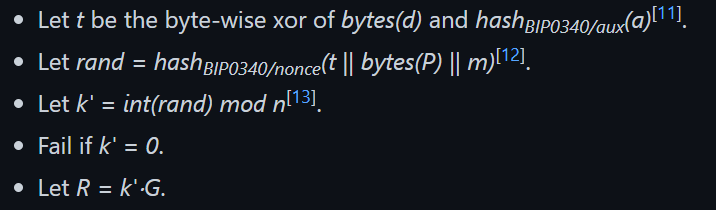
按照schnorr算法的原理，理论上我们应该随机选取一个k，然后计算R= k\*G 作为commitment。

我参考了一下bitcoin中的nonce生成算法和openssl中ecdsa签名的nonce,发现这两者都不是直接使用伪随机函数生成一个k，而是使得这个k与message、key有关联。

schnorr算法给出了一个指导性的对于nonce的要求：

*the value must be a fresh uniformly random string which is not even partially predictable for the attacker[5].*

bitcoin中的实现：

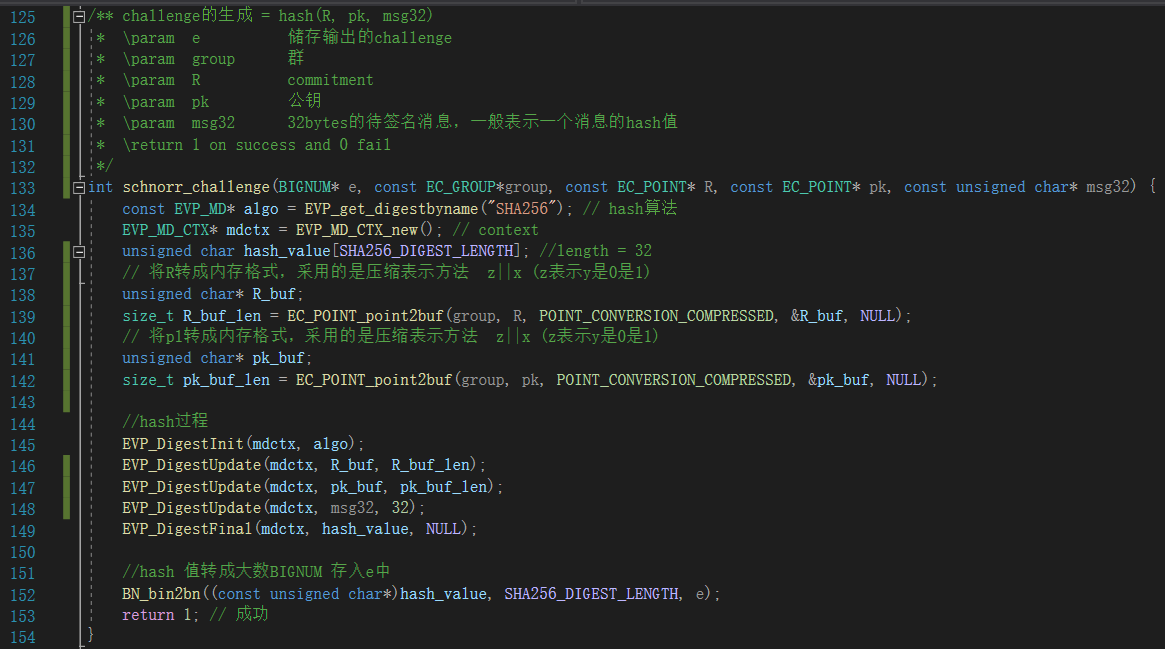
图：bitcoin中的nonce实现原理[5]

但这里我直接使用了最简单的机制随机生成一个k。

由于OPENSSL给的函数是从[0,order)这个范围内的随机数，于是我先生成[0,order-1）的随机数再对该随机数+1，则生成一个[1,order）的随机数

### 4.2.2 chanllenge

* 函数截图



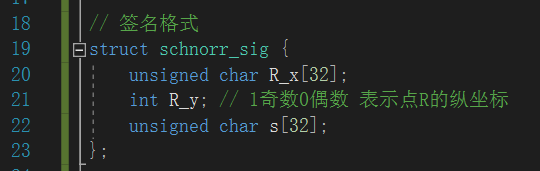
* 说明

challenge 的值 e = hash(R,pk,msg32)即对R、pk、msg的hash，采用的算法是sha256，

先对R，pk进行序列化，转成能作为hash的输入，然后依次将三者拼接起来，进行hash，再将hash值转成大数，储存在e里面

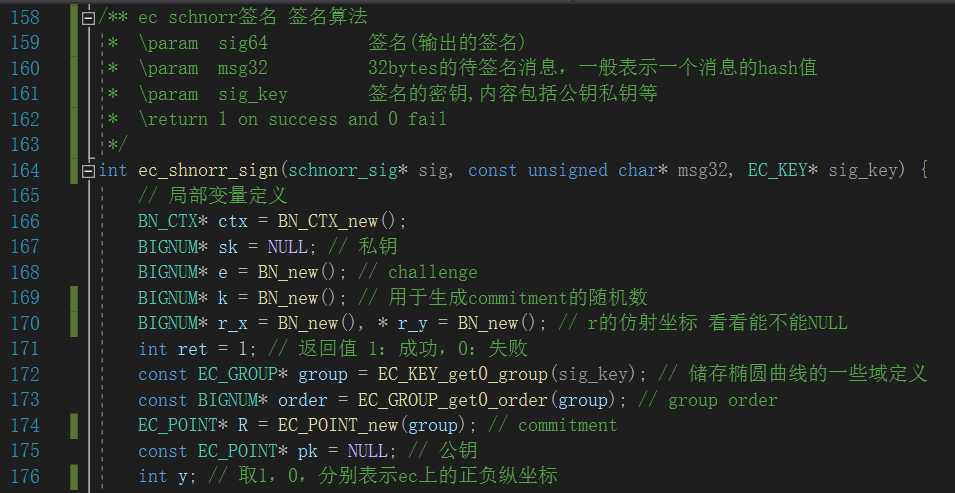
### 4.2.3 sign

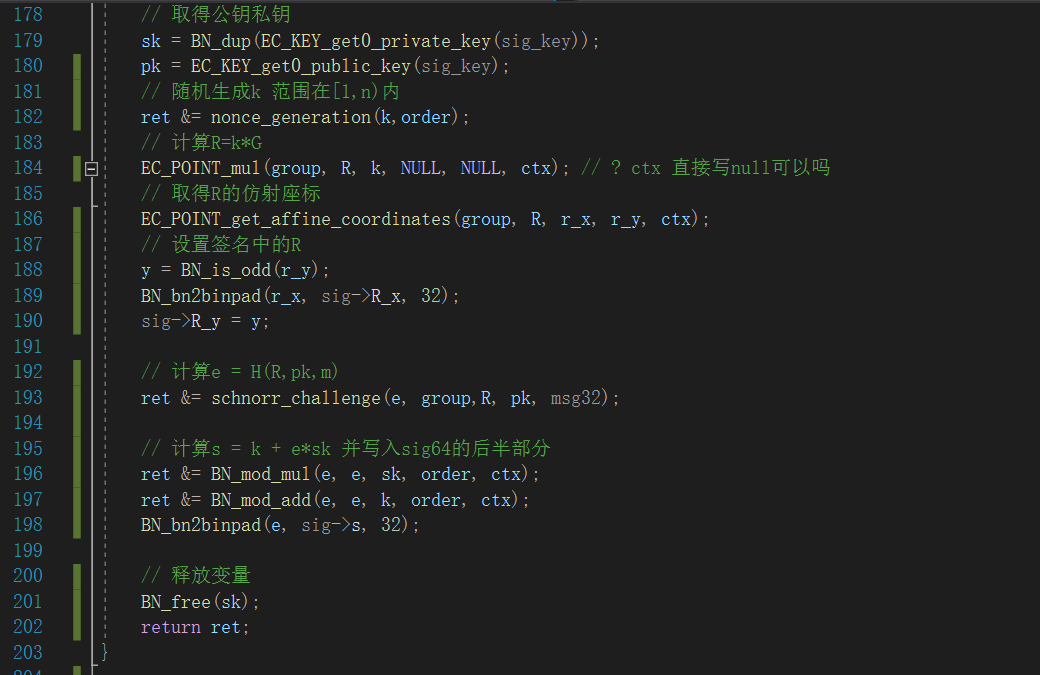
* 签名结构设计



[ec\_schnorr.h]

* 函数截图





* 说明

输入密钥，消息

输出签名

首先取得公钥私钥

然后利用nonce\_generation()函数生成k

计算R = k\*G

为了将R写入签名，先取得R的放射坐标，然后如果y坐标为偶数，则签名中R\_y=1，否则为0 ，这就是用1bit来表示R的y值。另外再把R的x坐标也序列化后（转成内存形式）存入签名的R\_x中

然后利用schnorr\_challenge()函数计算e

利用e计算s = k+e\*sk

写入签名中

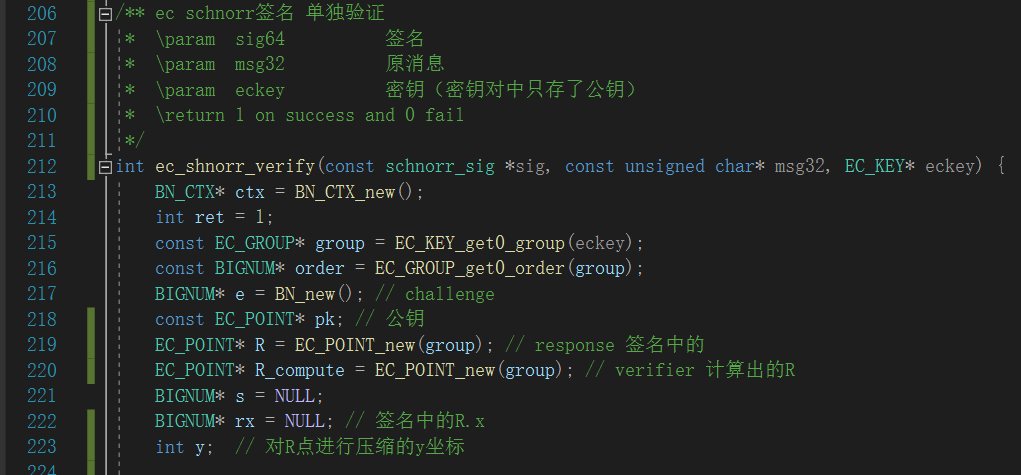
lem 椭圆曲线上的mod该如何计算？

int BN\_mul(BIGNUM \*r, BIGNUM \*a, BIGNUM \*b, BN\_CTX \*ctx);

## 4.3 验签

### 4.3.1 单独验签

* 函数截图





* 说明

输入签名，消息，密钥

输出验签成功与否消息（成功1，失败0）

先从EC\_KEY结构中取得公钥，类型为EC\_POINT；

然后取得签名中的R\_x,R\_y，并由此计算处EC\_POINT类型的R点；

计算e = Hash（R,pk,m）

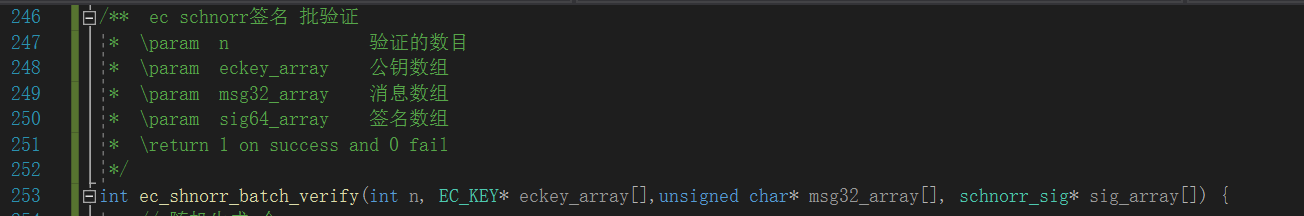
从签名中取出s

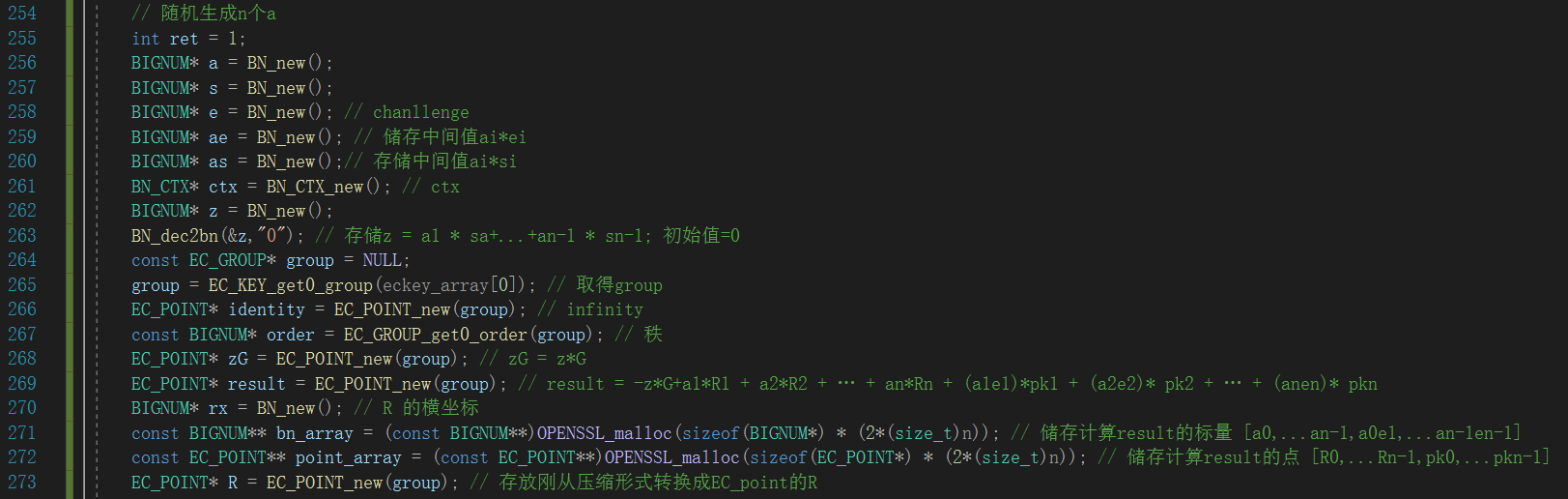
根据e,s计算R\_compute = s\*G+(-e)\*pk

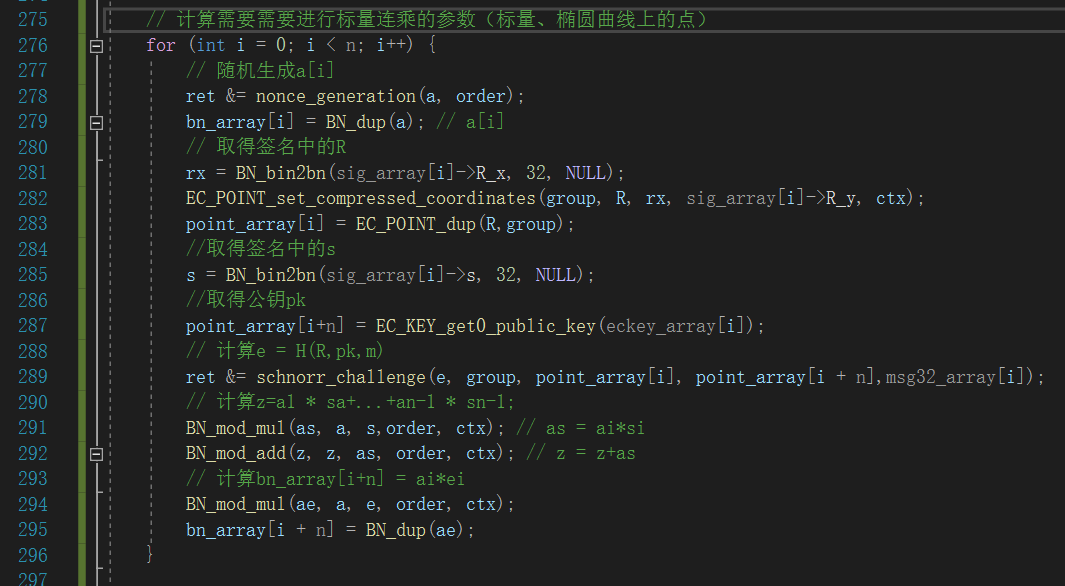
比较R和R\_compute是否相等，如果一样就返回1，否则返回0；

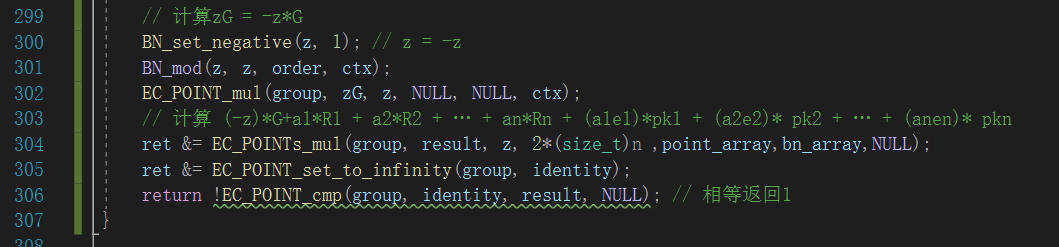
### 4.3.2 批验签（batch verification）

* 函数截图









* 说明

输入：密钥数组，消息数组，签名数组

输出：验签成功与否

我们要计算

infinity ？= -z\*G+a1\*R1 + a2\*R2 + … + an\*Rn + (a1e1)\*pk1 + (a2e2)\* pk2 + … + (anen)\* pkn

其中z = a0 \* s0+...+an-1 \* sn-1

由于要进行多标量乘法，于是我们要输入对应的标量的数组和EC上点的数组。

先根据输入计算出这些值

标量数组的值：[a0,...an-1,a0e1,...an-1en-1]

点数组的值：[R0,...Rn-1,pk0,...pkn-1]

利用nonce\_generation()函数随机生成[1,order)之间的数作为a[i]

和单独验签采用的技巧一样，得到e和s，R

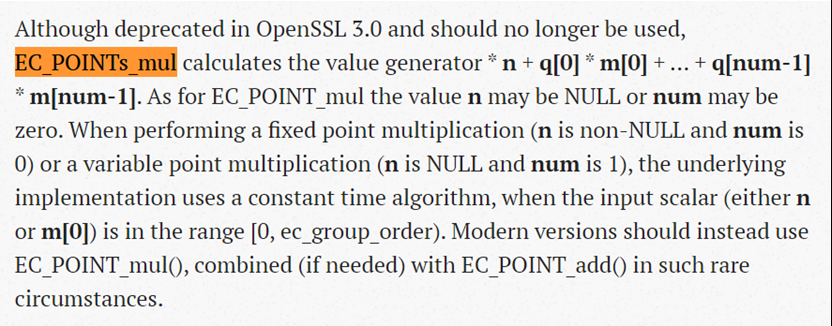
将a[i]存入数组

计算出标量a[i]e[i]的值存入数组对应位置

计算a[i]s[i]并求和

这个过程中也将点存入对应的数组

使用OPENSSL 的函数来进行多标量乘法计算



这是个constant time的算法

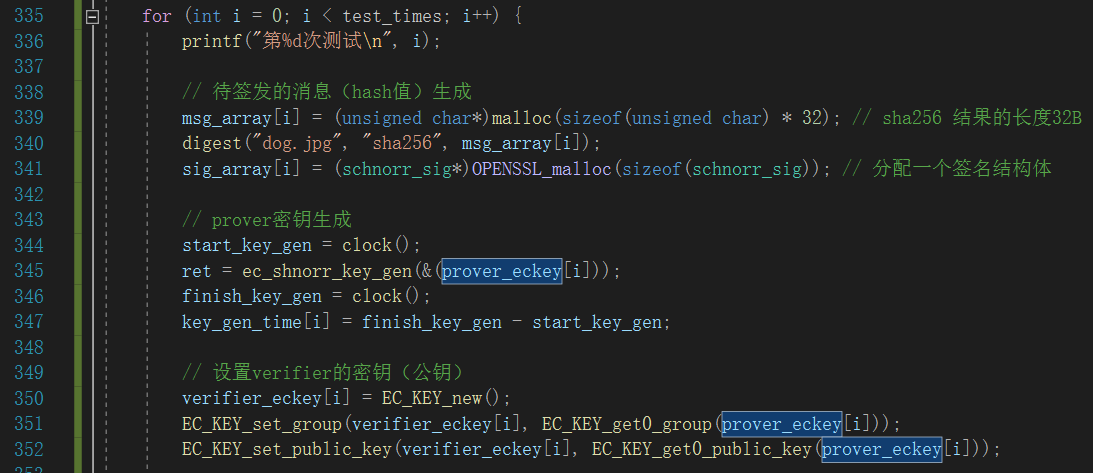
最后使用标量乘法验证椭圆曲线上的identity（infinity）是否和多标量乘法结果一样？

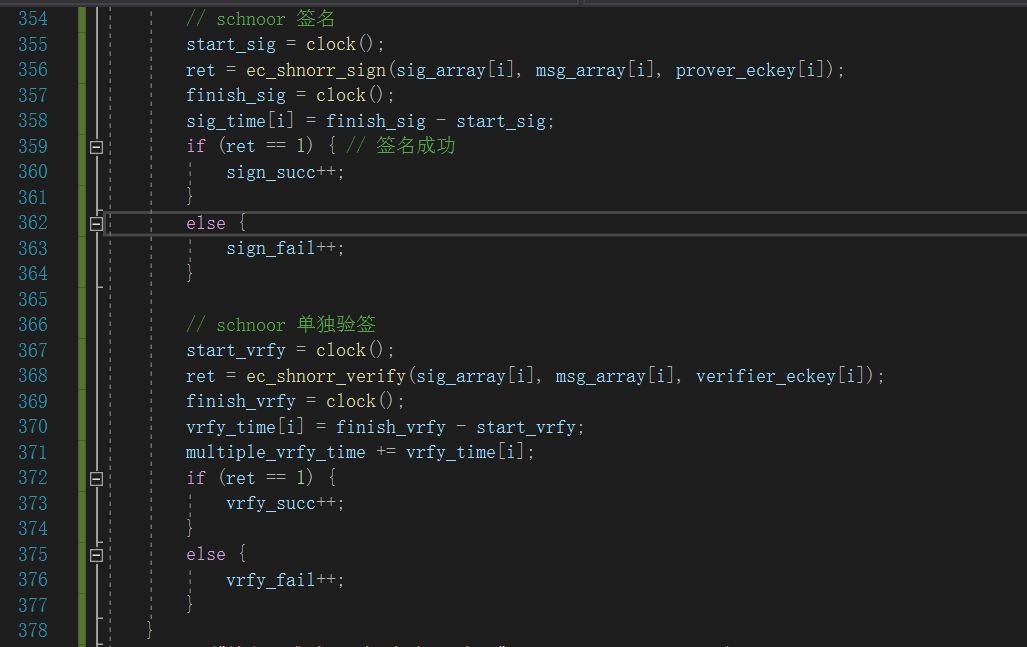
一样则输出1，否则输出0；

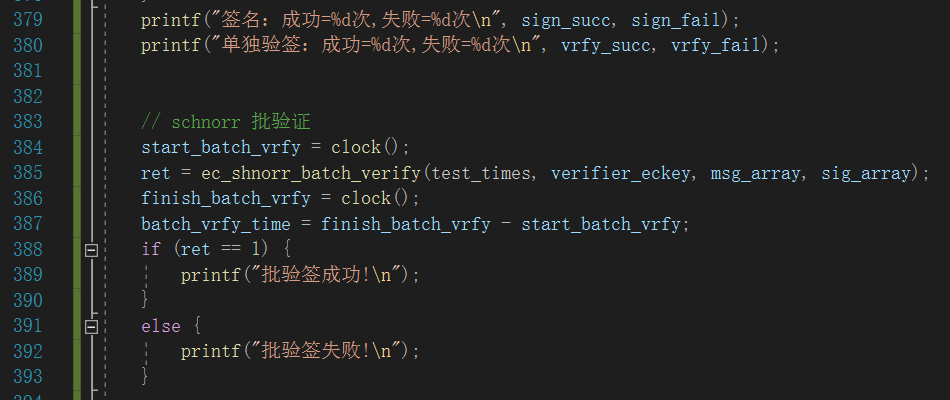
## 4.4 总体运行流程

* 函数截图











* 说明

这是运行的main函数

设置test\_times = 10000作为测试次数

使用clock（）计算各个函数运行的时钟周期。

在每次循环里面，先生成文件的hash值，作为待签名消息

然后利用密钥生成函数ec\_schnorr\_key\_gen()生成prover的密钥

把公钥部分传递给verifier的密钥

然后进行schnorr签名

进行单独验签

循环结束

输出签名成功次数、失败次数

输出验签成功次数、失败次数

结束后再对这些测试过程中生成的签名进行一次批验签。

输出验签成功与否消息

这个过程中测试的时间数据有：

10000次密钥生成的时间中值

10000次签名函数的时间中值

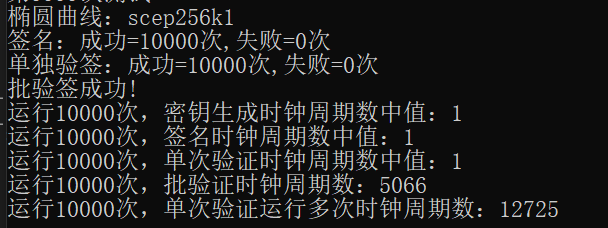
10000次验签函数的时间中值

1次验证10000个签名的批验签函数时间

10000次分别验签的总时间

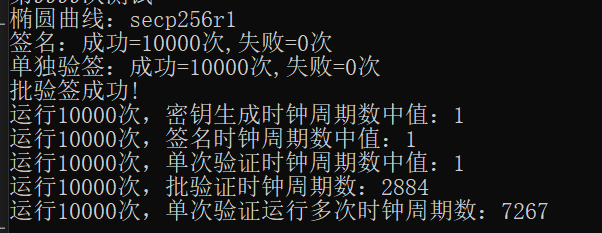
# 实验结果

* secp256k1



批验证明显比单独多次运行验证快很多

* secp256r1



可以看到在这个曲线上效率已经高了很多

* ed25519

在openssl1.1.1以上才能支持，这里就没有做出测试，但代码中实现了对应的部分。

schnorr签名在这个曲线上效率也很高。

# 讨论与分析

本实验实现了schnorr签名的密钥生成、签名、验签、批验签四大主要功能。

实现的代码时constant-time的

且避免使用if结构抗信道攻击

借鉴了bitcoin的源代码和openssl中ecdsa签名的实现。

但由于时间原因也还有很多没有深入探讨。比如bitcoin中x-only的密钥表示。密码学工程领域上效率、安全性的提高。

# 参考文献

1. *Introduction to Schnorr Signatures*. Available from: <https://suredbits.com/introduction-to-schnorr-signatures/>.

2. *Schnorr Applications: Batch Verification*. Available from: <https://suredbits.com/schnorr-applications-batch-verification/>.

3. Karthikeyan, D.E. *Survey of Elliptic Curve Scalar Multiplication Algorithms*. 2012.

4. Möller, B. *Algorithms for Multi-exponentiation*. 2001. Berlin, Heidelberg: Springer Berlin Heidelberg.

5. *BIP-340*. Available from: <https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki>.

6. *Windows安装配置C/C++（VS2017）-OpenSSL(1.1.0f)开发环境*. Available from: <https://blog.csdn.net/DaSo_CSDN/article/details/72952625>.