椭圆曲线公钥恢复算法与SM2编程实现

💡 报告分成两大部分，第一部分调研椭圆曲线公钥恢复算法，第二部分进行SM2公钥恢复算法编程实现，具体目录结构如下：

椭圆曲线公钥恢复算法

在以太坊中，交易消息中不包含“from”字段（即交易发起者的地址），这是因为交易发起者的公钥可以直接从ECDSA签名中计算出来。而一旦你有公钥，就可以很容易地计算出对应的地址。恢复签名者公钥的过程称为公钥恢复，对应的算法即为公钥恢复算法。

接下来描述以太坊中公钥恢复算法的过程。

首先根据给定 ECDSA签名算法 中计算的值 **r** 和 **s**，我们可以计算得到两个可能的公钥。我们根据签名中的 **x** 坐标 **r** 值计算两个椭圆曲线点 **R** 和 **R'**。对于 r 值，计算它关于 n 的逆元 r^{-1} ，其中 n 是椭圆曲线的阶数。最后计算 e ，它是消息的散列值。然后可以得到两个可能的公钥：

$$K_1 = r^{-1}(sR - eG)$$

$$K_2 = r^{-1}(sR' - eG)$$

其中:

- K_1 和 K_2 是签名者公钥的两种可能性
- r^{-1} 是签名的 r 值的逆元
- s 是签名的 s 值
- R 和 R' 是临时公钥 Q 的两种可能性
- e 是消息散列的最低位

- G 是椭圆曲线生成点

为了使计算更有效率，在以太坊交易签名里包括一个前缀值 v，它告诉我们两个可能的R值中哪一个是真正临时的公钥。如果 v 是偶数，那么R是正确的值。如果 v 是奇数，那么选择R'。这样，我们只需要计算R的一个值。这也就是以太坊交易中签名数据的（v, r, s）。

椭圆曲线上点的压缩

在上述过程，有一个地方可能还没有解释清楚，那就是为什么一个 x 坐标会对应有两个椭圆曲线上的点？其实这里用到了椭圆曲线上点的压缩和解压缩方法。

根据椭圆曲线方程，我们只需要知道 x 坐标，就可以通过方程计算出 y 坐标，这样就不用同时保存x，y的值，减少了存储和带宽。但是如果只知道x，带入方程会求出两个y，一正一负，对应两个不同的点，所以还必须有一个标志来区别实际使用的是哪个。在以太坊中就采用了压缩公钥格式，具体格式为：

- 前缀02 + x （当y为偶数）
- 前缀03 + x （当y为奇数）

为什么y一定是一奇一偶呢，刚刚不是说一正一负吗？

假设 y 是方程的一个解，那么 -y 也是方程的一个解，但在模运算的规则下，`-y ≡ p - y (mod p)`，所以 p-y 也是方程的解，y-p 也是方程的解，但是在mod p 的有限域中，取值范围是[0, p-1]，没有负数，所以 -y 和 y-p 在椭圆曲线上取不到，最终得到就是两个解 y 和 p-y，因为p是大素数，所以 y 和 p-y 一定是一奇一偶。

在《SM2椭圆曲线公钥密码算法》中也谈及了椭圆曲线上点的压缩和解压缩方法，如下图所示：

A.5.2　F_p上椭圆曲线点的压缩与解压缩方法

设$P=(x_P, y_P)$是定义在F_p上椭圆曲线E：$y^2 = x^3 + ax + b$上的一个点，$\tilde{y}_P$为y_P的最右边的一个比特，则点P可由x_P和比特$\tilde{y}_P$ 表示。

由x_P和$\tilde{y}_P$恢复y_P的方法如下：

a) 计算域元素$\alpha=(x_P{}^3 + ax_P + b) \bmod p$；

b) 计算$\alpha \bmod p$的平方根β(参见附录B.1.4)，若输出是“不存在平方根”，则报错；

c) 若β的最右边比特等于$\tilde{y}_P$，则置$y_P=\beta$；否则置$y_P = p - \beta$。

椭圆曲线签名过程

在推导椭圆曲线公钥算法之前，还需要先回顾了解下椭圆曲线的签名过程。

Given domain parameters (p, a, b, G, n, h), private key d, message m.

1. Randomly select $k \in [1, n\text{-}1]$.

2. Compute $kG = (x, y)$.

3. Compute $r = x \bmod n$, if $r = 0$ then goto step 1.

4. Compute $e = H(m)$. $H()$ is a hash function.

5. Compute $s = k^{-1}(e + dr) \bmod n$, if $s = 0$ then goto step 1

Return signature (r, s)

1. 随机选择一个临时私钥 k 值，范围在 [1, n-1]，其中 n 为椭圆曲线的阶数
2. 计算临时公钥 kG = (x , y)
3. 计算 r 值， r = x mod n，如果 r 为0则返回第1步重新选择私钥 k
4. 计算消息散列值，e = H(m)
5. 计算 s 值， $s = k^{-1}(e + dr)\ mod\ n$ ，其中 d 是签名者的私钥，如果算得 s 为0则返回第一步重新选择 k
6. 最终得到签名 (r, s)

对应签名的验证过程如下：

Given domain parameters (p, a, b, G, n, h), public key Q, message m, signature (r, s)

1. Verify that $r, s \in [1, n\text{-}1]$, otherwise return "*failed*"

2. Compute $e = H(m)$. $H()$ is the hash function.

3. Compute $w = s^{-1} \bmod n$.

4. Compute $u_1 = ew \bmod n$, $u_2 = rw \bmod n$.

5. Compute $X = (x, y) = u_1G + u_2Q$. If $X = O$ then return "*failed*".

6. Compute $v = x \bmod n$.

If $v = r$ then return "*success*", else return "*failed*"

1. 检验 r, s 是否满足取值范围 [1, n-1]
2. 计算消息散列值， e = H(m)
3. 计算 s 的逆元， $w = s^{-1}\ mod\ n$
4. 计算 $u_1 = ew\ mod\ n$，$u_2 = rw\ mod\ n$

5. 计算 $X=(x,y)=u_1G+u_2Q$ 。 其中G是生成元，Q是签名者的公钥。如果X为无穷远点则验签失败。
6. 计算 v = x mod n
7. 如果 v = r 则验签成功，否则失败

椭圆曲线签名和验证的正确性证明如下所示：

$$k \equiv s^{-1}(e+dr) \equiv s^{-1}e+s^{-1}rd \equiv we+wrd \equiv u_1+u_2d(mod\ n)$$

$$X=u_1G+u_2Q=(u_1+u_2d)G=kG$$

即验签时计算的X与签名时的临时公钥是相等的，对应的 x 坐标也相等，所以验签时判断 v 值是否等于 r 值即可。

椭圆曲线公钥恢复算法推导

好的，前序准备都已经完成，现在来推导公钥恢复算法，这里主要用到了签名过程中的相关计算。

设临时公钥为R，即 $R=kG$

$$s=k^{-1}(e+dr)\ mod\ n \quad \Rightarrow \quad k=s^{-1}(e+dr)\ mod\ n$$

$$kG=s^{-1}(e+dr)\ G \quad \Rightarrow \quad R=s^{-1}(eG+rQ) \quad \Rightarrow \quad Q=r^{-1}(sR-eG)$$

因为通过 r 值可以计算逆元 r^{-1} ，s 值已知，消息散列e和生成元G已知，临时公钥 R 可以通过 r（对应公钥的x坐标）利用椭圆曲线上点的解压缩方法求解，因此公钥 Q 是可以被计算出来的。

在 椭圆曲线标准 中也描述了普通椭圆曲线公钥恢复算法的实现，如下图所示。

Output: An elliptic curve public key Q for which (r,s) is a valid signature on message M.

Actions: Find public key Q as follows.

1. For j from 0 to h do the following.
   - 1.1. Let $x=r+jn$.
   - 1.2. Convert the integer x to an octet string X of length $mlen$ using the conversion routine specified in Section 2.3.7, where $mlen=\lceil(\log_2 p)/8\rceil$ or $mlen=\lceil m/8\rceil$.
   - 1.3. Convert the octet string $02_{16}\|X$ to an elliptic curve point R using the conversion routine specified in Section 2.3.4. If this conversion routine outputs “invalid”, then do another iteration of Step 1.
   - 1.4. If $nR\neq\mathcal{O}$, then do another iteration of Step 1.
   - 1.5. Compute e from M using Steps 2 and 3 of ECDSA signature verification.
   - 1.6. For k from 1 to 2 do the following.
     - 1.6.1. Compute a candidate public key as:
       $$Q=r^{-1}(sR-eG).$$
     - 1.6.2. Verify that Q is the authentic public key. (For example, verify the signature of a certification authority in a certificate which has been truncated by the omission of Q from the certificate.) If Q is authenticated, stop and output Q.
     - 1.6.3. Change R to $-R$.
2. Output “invalid”.

SM2公钥恢复算法编程实现

因为SM2公钥签名算法和以太坊上椭圆曲线的签名算法不太一样，所以需要先了解SM2的签名过程，然后再推导SM2对应的公钥恢复算法。

SM2签名与验证过程

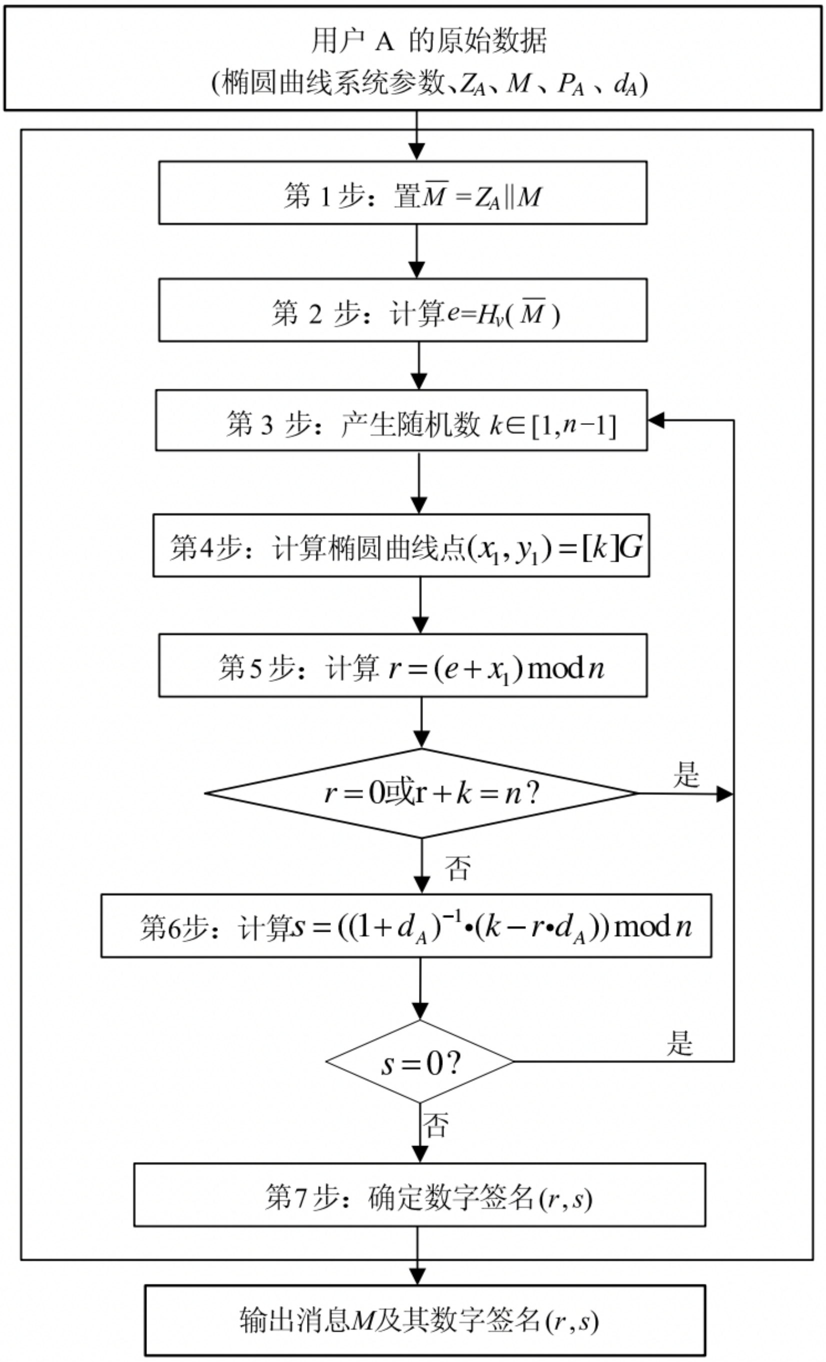
根据官方的 SM2 椭圆曲线公钥密码算法 可以找到SM2算法的签名过程。

> 设待签名的消息为M，为了获取消息M的数字签名(r,s)，作为签名者的用户A应实现以下运算步骤：
>
> A1：置$\overline{M}=Z_A \parallel M$；
>
> A2：计算$e = H_v(\overline{M})$，按本文本第1部分4.2.3和4.2.2给出的细节将e的数据类型转换为整数；
>
> A3：用随机数发生器产生随机数$k \in[1,n\text{-}1]$；
>
> A4：计算椭圆曲线点$(x_1,y_1)=[k]G$，按本文本第1部分4.2.7给出的细节将x_1的数据类型转换为整数；
>
> A5：计算$r=(e+x_1) \bmod n$，若$r=0$或$r+k=n$则返回A3；
>
> A6：计算$s = ((1 + d_A)^{-1} \cdot (k - r \cdot d_A)) \bmod n$，若$s=0$则返回A3；
>
> A7：按本文本第1部分4.2.1给出的细节将r、s的数据类型转换为字节串，消息M 的签名为(r,s)。

- 设待签名的消息为M，为了获取消息M的数字签名(r,s)，作为签名者的用户A应实现以下运算步骤：

1. 置 $\overline{M} = Z_A \mid\mid M$
2. 计算消息散列值， $e = H(\overline{M})$ ，并将e的数据类型转换为整数
3. 用随机数发生器产生随机数 k，取值范围 [1, n-1]
4. 计算椭圆曲线上的点（也就是临时公钥）， $(x_1, y_1) = kG$ ，并将 x_1 数据类型转换为整数
5. 计算 $r = (e + x_1)\ mod\ n$ ，若 r = 0 或 r+k = n 则 返回第3步，重新选择k
6. 计算 $s = ((1 + d_A)^{-1} \cdot (k - r \cdot d_A))\ mod\ n$ ，若s = 0 则返回第3步
7. 将r、s类型转换为字符串，得到消息M的签名(r, s)

SM2数字签名的算法流程图如下所示。

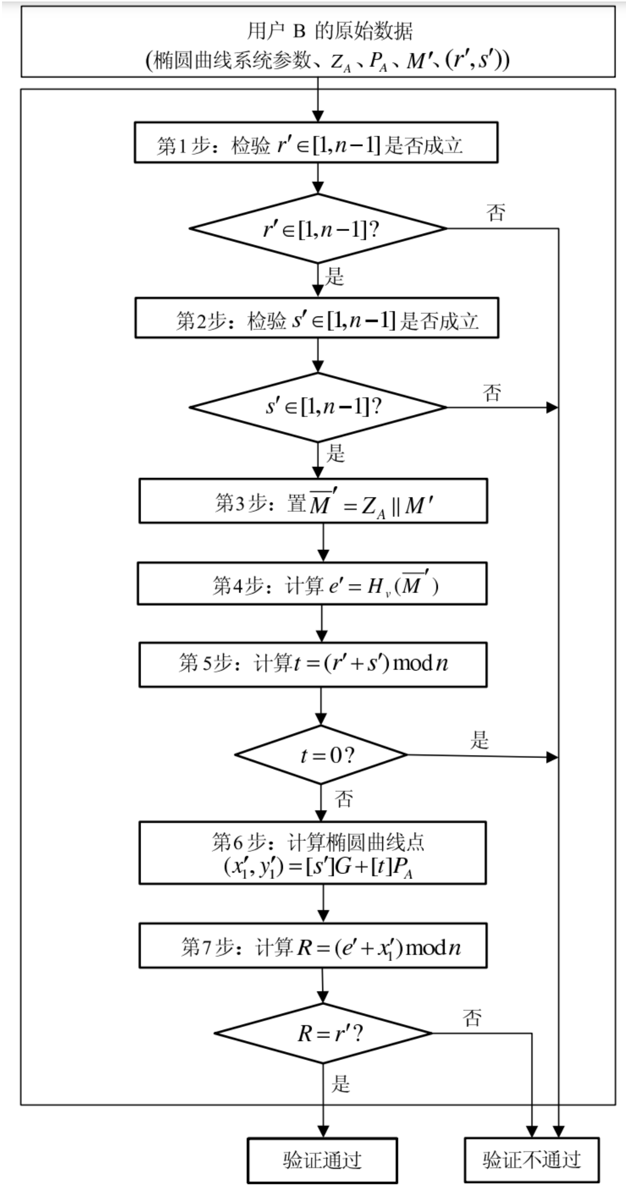


SM2验证过程

- 为了检验收到消息M'及其数字签名(r',s')，作为验证者的用户B应实现以下运算步骤：

1. 检验 r' 是否在 [1, n-1] 范围内
2. 检验 s' 是否在 [1, n-1] 范围内
3. 置 $\overline{M}' = Z_A \mid\mid M'$
4. 计算消息散列值， $e = H(\overline{M}')$ ，并将e的数据类型转换为整数
5. 将 r' , s' 数据类型转换为整数，计算 $t = (r' + s')\ mod\ n$
6. 计算椭圆曲线点 $(x_1', y_1') = [s']G + [t]P_A$
7. 计算 $R = (e' + x_1')\ mod\ n$ ，检验 R = r' 是否成立，如果成立则验证通过，否则验证不通过

SM2数字签名的验证过程如下图所示。



SM2公钥恢复算法

跟椭圆曲线中方法类似，从SM2签名算法的相关计算中，可以推导出对应的公钥恢复算法 。

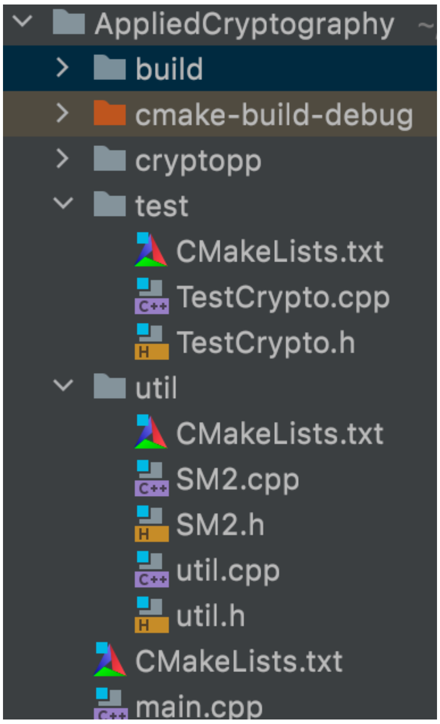
$$s = ((1 + d_A)^{-1} \cdot (k - r \cdot d_A))\ mod\ n \quad \Rightarrow \quad k = (1 + d_A)s + rd_A = (s + (s + r)d_A)\ mod\ n$$

设 $R = kG \qquad t = r + s$ ，

$$kG = (s + (s + r)d_A)G \quad \Rightarrow \quad R = sG + tP_A \quad \Rightarrow \quad P_A = t^{-1}(R - sG)$$

因为r，s值可以计算得到t，进而得到t的逆元 t^{-1} ，R 可以通过 $x_1 = r - e\ mod\ n$ ，并利用椭圆曲线的压缩方法计算得到。s和G也都已知，因此，是可以计算得到公钥 P_A

代码结构说明



附件代码结构如上图所示：

- cryptopp：该目录是开源库CryptoPP 的文件目录
- test：测试文件目录，存放测试类TestCrypto，主要测试CryptoPP的相关API，模拟椭圆曲线的签名、验证以及公钥恢复算法；还有就是测试自定义实现的SM2 的签名、验证以及公钥恢复算法
- util：该目录下，util类是cryptopp方法的封装，SM2是自定义实现类
- main.cpp：实现测试函数

编程实现说明

当了解了算法原理之后，实现起来就比较容易了，只要根据公式进行编程即可。在实现上借助了开源的CryptoPP库，来支持密码学里中原语操作，比如有限域上的计算，椭圆曲线上点的运算等。代码上主要是实现了一个`SM2`类，定义了签名，验证和公钥恢复算法，具体的代码实现这里不再展示，可见代码附件。

C++

```cpp
// 借助CryptoPP开源库 实现 SM2算法
class SM2 {
public:
    // 签名：输入的消息和输出的签名数据  均为字节格式，签名是由(r,s)各32字节拼接得到的
    void Sign(const std::string &message, std::string &signature);
    // 验证：输入的消息和签名均为字节格式，输出为bool值
    bool Verify(const std::string &message, const std::string &signature);
    // 公钥恢复算法，输入消息，签名 (r,s) ，y值是否为奇数，  输出pubKey 字节格式，由(x,y)各32字节拼接而成
    void RecoverPublicKey(const std::string &message, bool isOdd,
                                 const Integer &r, const Integer &s, std::string &pubKey)
private:
    std::shared_ptr<ECP> ecp;                       // SM2 对应的椭圆曲线
    std::shared_ptr<Integer> p;
    std::shared_ptr<Integer> a;
    std::shared_ptr<Integer> b;
    std::shared_ptr<Integer> privateKey;          // 私钥
    std::shared_ptr<ECPPoint> publicKey;          // 公钥，以Point表示
    std::shared_ptr<ECPPoint> generator;          // 生成元 点
    std::shared_ptr<Integer> n;                   // 生成元 对应的阶数
    std::string ZA;                               // 杂凑值，字节形式 ： ZA = H256(ENTL_A ∥ ID_A ∥ a ∥ b ∥ xG ∥ yG ∥ xA ∥ yA)

    // 默认构造函数，使用官方参数
    SM2();
    // 初始化曲线参数，使用官方参数
    void init();
    // 计算杂凑值 ZA
    void ComputeZA();
};
```

测试代码如下所示：

C++

```cpp
// 测试 SM2 类
void TestCrypto::Test_SM2() {
    SM2 sm2;
    std::string message = "message digest";
    std::string signature;
    Integer k = Util::StringToInteger("59276E27D506861A16680F3AD9C02DCCEF3CC1FA3CDBE4CE6D54B80DEAC1BC21");
    sm2.Sign(message, signature, k);

    Integer r = Util::StringToInteger(Util::HexEncode(signature.substr(0,32)));
    Integer s = Util::StringToInteger(Util::HexEncode(signature.substr(32,32)));

    // 打印输出，对照真实数据
    cout << "————————————测试SM2 签名算法————————————" << endl;
    cout << "真实 r:" << "F5A03B0648D2C4630EEAC513E1BB81A15944DA3827D5B74143AC7EACEEE720B3" << endl;
    cout << "计算 r:" << Util::HexEncode(signature.substr(0,32)) << endl;
    cout << "真实 s:" << "B1B6AA29DF212FD8763182BC0D421CA1BB9038FD1F7F42D4840B69C485BBC1AA" << endl;
    cout << "计算 s:" << Util::HexEncode(signature.substr(32,32)) << endl;

    cout << "————————————测试SM2 验证签名————————————" << endl;
    cout << "message: " << message << endl;
    cout << "r: " << Util::IntegerToString(r) << endl;
    cout << "s: " << Util::IntegerToString(s) << endl;
    cout << "签名验证结果: " << boolalpha << sm2.Verify(message, signature ) << endl;


    cout << "————————————测试SM2 公钥恢复算法————————————" << endl;
    std::string pubKey;
    sm2.RecoverPublicKey(message,false, r, s,pubKey);
    ECP::Point P(Util::StringToInteger(Util::HexEncode(pubKey.substr(0,32))),
                 Util::StringToInteger(Util::HexEncode(pubKey.substr(32,32))));
    cout << "真实公钥 P.x:" << "09F9DF311E5421A150DD7D161E4BC5C672179FAD1833FC076BB08FF356F35020" << endl;
    cout << "恢复公钥 P.x:" << Util::IntegerToString(P.x) << endl;
    cout << "真实公钥 P.y:" << "CCEA490CE26775A52DC6EA718CC1AA600AED05FBF35E084A6632F6072DA9AD13" << endl;
    cout << "恢复公钥 P.y:" << Util::IntegerToString(P.y) << endl;
}
```

测试数据来自：SM2 自检数据

测试结果如下所示：

```
———————————————测试SM2 签名算法———————————————
真实 r:F5A03B0648D2C4630EEAC513E1BB81A15944DA3827D5B74143AC7EACEEE720B3
计算 r:F5A03B0648D2C4630EEAC513E1BB81A15944DA3827D5B74143AC7EACEEE720B3
真实 s:B1B6AA29DF212FD8763182BC0D421CA1BB9038FD1F7F42D4840B69C485BBC1AA
计算 s:B1B6AA29DF212FD8763182BC0D421CA1BB9038FD1F7F42D4840B69C485BBC1AA
———————————————测试SM2 验证签名———————————————
message: message digest
r: F5A03B0648D2C4630EEAC513E1BB81A15944DA3827D5B74143AC7EACEEE720B3
s: B1B6AA29DF212FD8763182BC0D421CA1BB9038FD1F7F42D4840B69C485BBC1AA
签名验证结果: true
———————————————测试SM2 公钥恢复算法———————————————
真实公钥 P.x:09F9DF311E5421A150DD7D161E4BC5C672179FAD1833FC076BB08FF356F35020
恢复公钥 P.x:09F9DF311E5421A150DD7D161E4BC5C672179FAD1833FC076BB08FF356F35020
真实公钥 P.y:CCEA490CE26775A52DC6EA718CC1AA600AED05FBF35E084A6632F6072DA9AD13
恢复公钥 P.y:CCEA490CE26775A52DC6EA718CC1AA600AED05FBF35E084A6632F6072DA9AD13
```

参考资料

SM2 椭圆曲线公钥密码算法

SM2 自检数据

SM2 官方参数定义

以太坊：签名前缀值（v）和公钥恢复

CryptoPP 官方文档