ECDSA 学习笔记

孙永康　11911409

理论部分

椭圆曲线数字签名算法（ECDSA）是使用椭圆曲线密码（ECC）对数字签名算法（DSA）的模拟。

- ECC : 基于椭圆曲线数学理论实现的一种非对称加密算法。 相比RSA，ECC优势是可以使用更短的密钥，来实现与RSA相当或更高的安全。 160位ECC加密安全性相当于1024位RSA加密，210位ECC加密安全性相当于2048位RSA加密（有待考证）。椭圆曲线加密算法（ECC） - 知乎 (zhihu.com)

  （后续会对其有更详细的介绍）
- DSA : 数字签名算法，Schnorr和ElGamal签名算法的变形， 该算法的安全性依赖于计算模数的离散对数的难度。DSA-数据签名算法（理论）_aaqian1的博客-CSDN博客dsa算法

详细介绍ECDSA：

- ### What is `ECDSA` ?

  `ECDSA`是一个签名算法，用于发布者Alice对自己的文件进行签名，与现实生活中的签名十分类似。接收者Bob有能力验证接收文件上的签名是否是发布者Alice亲手签的，但是它也能保证恶意的接收者Bob不存在伪造Alice签名的能力。

  `ECDSA`不会对数据进行加密、或阻止别人看到或访问你的数据，它可以防止的是确保数据没有被篡改。

- ### Basic Understanding

  选一条椭圆曲线，选一个原点，选一个随机数作为你的`私钥(Private key)`， 用原点和私钥计算出一个`公钥(Public key)`。

  一个数字签名包含两个数字，`R`和`S`：使用一个私钥和原文哈希值带入一个数学方程来产生`R`和`S`，如果将公钥和`S`代入另一个数学方程给出`R`的话，这个签名就是有效的。仅仅知道公钥是无法知道私钥或者创建出数字签名。

- ### The Hash

  `ECDSA`与消息的`SHA-1加密哈希`一起使用来对文件进行签名。哈希是你作用于数据的每一个字节然后给你一个代替该数据的整数的一个数学函数。SHA1算法将给出一个非常巨大的数（160位或者比特，如果用十进制表示的话将由49个数字组成），并且随着文件的一点细微的小变化，它也能够产生显著的变化。这个不可预测的特性让SHA1算法成为一个非常好的哈希算法，非常安全且产生"碰撞(collision)"（两个不同文件有相同的哈希）的可能性非常低，使得通过伪造数据获得特定的哈希的变得不可能。

- **Something About `ECC`**
  - **有限空间中的椭圆曲线：**

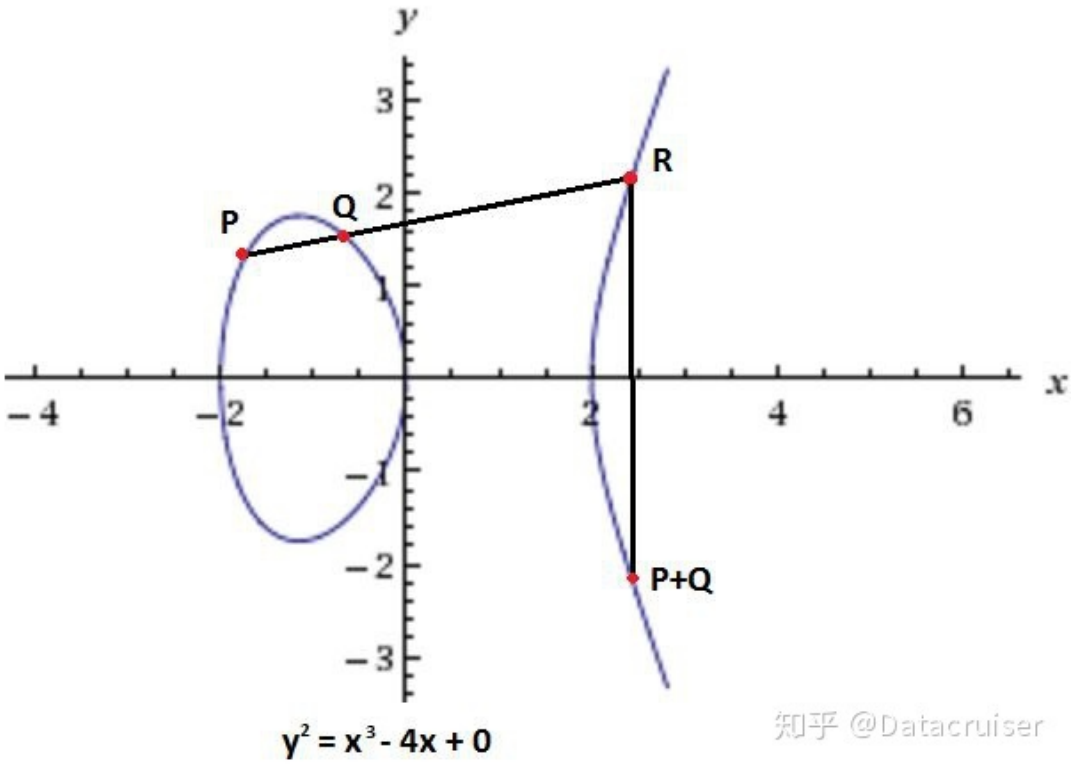
$$y^2 = (x^3 + a \times x + b) \mod p$$

取模运算的底p是一个素数且确保所有得到的数值在160比特所能够表示的范围之内，允许采用模平方根和模的乘法逆元来简化运算。

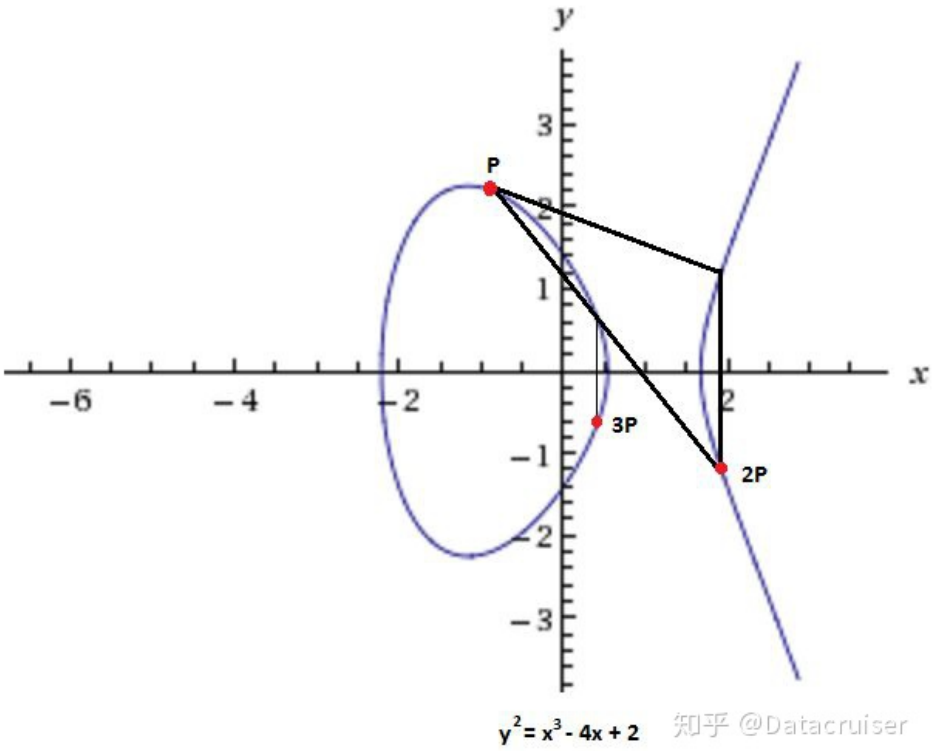
`ECDSA`方程给出了一条曲线，这条曲线上面一共有N个有效的点，因为Y轴的取值区间由模底p来确定，并且需要满足完美平方（Y^2）并关于X轴对称。我们一共有$N/2$个有效的x坐标。

  - **椭圆曲线上的加法与数乘：**

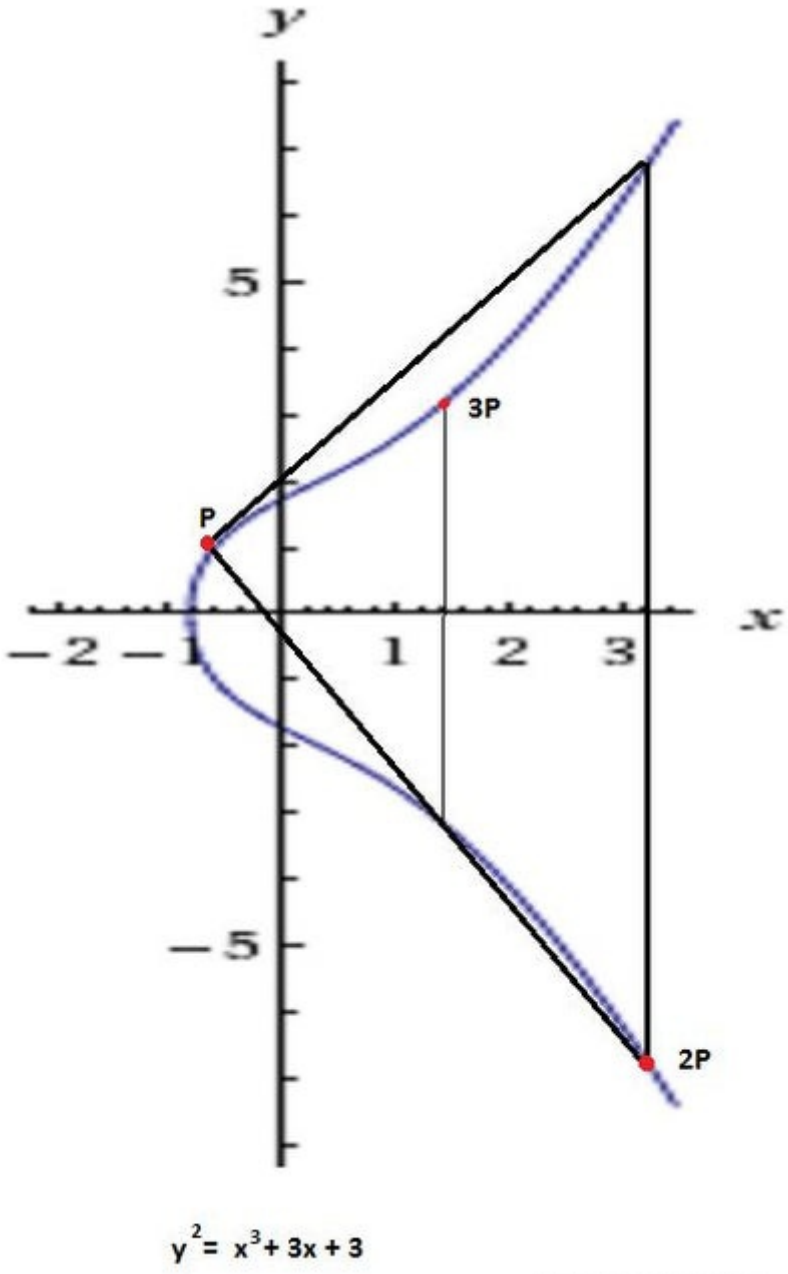
加法：



数乘：



$y^2 = x^3 - 4x + 2$



- **单向陷门函数：**

  一个椭圆曲线乘法的特性是你有一个点$R = k \times P$，你知道R和 P，但是你无法据此求出k，因为这里并没有椭圆曲线减法或者椭圆曲线除法可用，你并不能通过$k = R/P$得到k。并且，因为你可以做成千上万次的加法，最终你只是知道在曲线上面结束的点，但是具体是如何到达这个点你也并不知道。你无法进行反向操作，得到与点P相乘以后给你点R的 k。

The Beginning of `ECDSA` Algorithm

- **A Curve:**

$$y^2 = (x^3 + a \times x + b) \mod p$$

一条曲线及其相关参数：a, b, p, N, G（基点，参考点）

一些相关机构NIST(National Institute of Standards and Technology)和SECG(Standards for Efficient Cryptography Group)已经提供了预处理的已知高效和安全的标准化曲线参数。

- **公钥和私钥：**

  首先，你有一对密钥：公钥和私钥，私钥是一个随机数，也是160比特大小，公钥是将曲线上的点G与私钥相乘以后的曲线上的点。令dA表示私钥，一个随机数，Qa表示公钥，曲线上面的一个点，我们有$Qa = dA \times G$，其中G是曲线上面的参考点。

- **Creating a Signature**

  什么是一个signature：签名本身是40字节，由各20字节的两个值来进行表示，第一个值叫作R，第二个叫作S。值对(R, S)放到一起就是你的 ECDSA 签名。

  生成过程：

  - 产生一个随机数k，20字节
  - 利用点乘法计算$P = k \times G$
  - 点P的 x坐标即为R
  - 利用SHA1计算信息的哈希，得到一个20字节的巨大的整数z
  - 利用方程$S = k^{-1}(z + dA \times R) \mod p$计算$S$

- **Verifying the Signature**

  将公钥和签名还有hash值代入下式中即可完成验证。

  $$P = S^{-1} \times z \times G + S^{-1} \times R \times Qa$$

  数学推导：

  首先，我们有：

  $$P = S^{-1} \times z \times G + S^{-1} \times R \times Qa$$

  由于$Qa = dA \times G$，代入上式，则有：

  $$P = S^{-1} \times z \times G + S^{-1} \times R \times dA \times G = S^{-1}(z + dA \times R) \times G$$

  再由点P的 x坐标必须与R匹配，且R是点$k \times P$的 x坐标，即 $P = k \times G$，于是有：

  $$k \times G = S^{-1}(z + dA \times R) \times G$$

  两边将G拿掉，有：

  $$k = S^{-1}(z + dA \times R)$$

  两边求逆，即：

  $$S = k^{-1}(z + dA \times R)$$

- **The Security of ECDSA**

  需要同时知道随机数k和私钥dA才能够计算出S，但是需要R和公钥Qa来对签名进行确认和验证。并且由于$R = k \times G$以及$Qa = dA \times G$再加上 ECDSA 点乘法当中的单向陷门函数的特性，我们无法通过Qa和 R来计算dA或 k，这使得 ECDSA 算法非常安全，没有办法找到私钥，也无法在不知道私钥的情况下伪造签名。

  但是 随机数 k 的随机性是尤其重要的， 假如一直使用一个确定的数据 k，我们可以很轻易地破解到 ECDSA 的私钥。

  由于两次加密使用同一随机数k， 可得:

  $$S - S' = k^{-1}(z + dA \times R) - k^{-1}(z' + dA \times R) = k^{-1}(z + dA \times R - z' - dA \times R) = k^{-1}(z - z')$$

  于是有:

  $$k = \frac{z - z'}{S - S'}$$

  有了随机数k被破解，可以轻松计算出私钥dA:

  $$dA = （S \times k - z）/R$$

reference :

- Understanding how ECDSA protects your data
- 上文的中文翻译版：一文读懂ECDSA算法如何保护数据
- 一个很好的可以跟着学习并python实现的ECDSA教程：Lecture14.pdf (purdue.edu)

代码部分

签名:

```rust
/// Try to sign the given prehashed message using ECDSA.
///
/// This trait is intended to be implemented on a type with access to the
/// secret scalar via `&self`, such as particular curve's `Scalar` type.
#[cfg(feature = "arithmetic")]
#[cfg_attr(docsrs, doc(cfg(feature = "arithmetic")))]
pub trait SignPrimitive<C>: Field + Into<FieldBytes<C>> + Reduce<C::UInt> + Sized
where
    C: PrimeCurve + ProjectiveArithmetic + ScalarArithmetic<Scalar = Self>,
    SignatureSize<C>: ArrayLength<u8>,
{
    /// Try to sign the prehashed message.
    ///
    /// Accepts the following arguments:
    ///
    /// - `k`: ephemeral scalar value. MUST BE UNIFORMLY RANDOM!!!
    /// - `z`: scalar computed from a hashed message digest to be signed.
    ///   MUST BE OUTPUT OF A CRYPTOGRAPHICALLY SECURE DIGEST ALGORITHM!!!
    ///
    /// # Computing the `hashed_msg` scalar
    ///
    /// To compute a [`Scalar`] from a message digest, use the [`Reduce`] trait
    /// on the computed digest, e.g. `Scalar::from_be_bytes_reduced`.
    ///
    /// # Returns
    ///
    /// ECDSA [`Signature`] and, when possible/desired, a [`RecoveryId`]
    /// which can be used to recover the verifying key for a given signature.
    #[allow(non_snake_case)]
    fn try_sign_prehashed<K>(
        &self,
        k: K,
        z: Scalar<C>,
    ) -> Result<(Signature<C>, Option<RecoveryId>)>
    where
        K: Borrow<Self> + Invert<Output = Self>,
    {
        if k.borrow().is_zero().into() {
            return Err(Error::new());
        }

        // Compute scalar inversion of k
        let k_inv = Option::<Scalar<C>>::from(k.invert()).ok_or_else(Error::new)?;

        // Compute R = k×G
        let R = (C::ProjectivePoint::generator() * k.borrow()).to_affine();

        // Lift x-coordinate of R (element of base field) into a serialized big
        // integer, then reduce it into an element of the scalar field
        let r = Self::from_be_bytes_reduced(R.x());

        // Compute `s` as a signature over `r` and `z`.
        let s = k_inv * (z + (r * self));

        if s.is_zero().into() {
            return Err(Error::new());
        }

        // TODO(tarcieri): support for computing recovery ID
        Ok((Signature::from_scalars(r, s)?, None))
    }
}
```

验证：

```rust
/// Verify the given prehashed message using ECDSA.
///
/// This trait is intended to be implemented on type which can access
/// the affine point represeting the public key via `&self`, such as a
/// particular curve's `AffinePoint` type.
#[cfg(feature = "arithmetic")]
#[cfg_attr(docsrs, doc(cfg(feature = "arithmetic")))]
pub trait VerifyPrimitive<C>: AffineXCoordinate<C> + Copy + Sized
where
    C: PrimeCurve + AffineArithmetic<AffinePoint = Self> + ProjectiveArithmetic,
    Scalar<C>: Reduce<C::UInt>,
    SignatureSize<C>: ArrayLength<u8>,
{
    /// Verify the prehashed message against the provided signature
    ///
    /// Accepts the following arguments:
    ///
    /// - `z`: prehashed message to be verified
    /// - `sig`: signature to be verified against the key and message
    fn verify_prehashed(&self, z: Scalar<C>, sig: &Signature<C>) -> Result<()> {
        let (r, s) = sig.split_scalars();
        let s_inv = Option::<Scalar<C>>::from(s.invert()).ok_or_else(Error::new)?;
        let u1 = z * s_inv;
        let u2 = *r * s_inv;

        let x = ((C::ProjectivePoint::generator() * u1) + (C::ProjectivePoint::from(*self) * u2))
            .to_affine()
            .x();

        if Scalar::<C>::from_be_bytes_reduced(x) == *r {
            Ok(())
        } else {
            Err(Error::new())
        }
    }
}
```