blog.openacid.com /storage/ec-2/

Erasure-Code-擦除码-2-实现篇

: 2020/2/4

本文链接: https://blog.openacid.com/storage/ec-2/

书接上回Permalink

上一篇 第一篇:原理 中, 我们介绍了EC的基本原理, 实际上EC的存储跟恢复过程可以理解为: 一条k-1次曲线可以通过k个系数或曲线上的点来确定.

我们也提到:

> 但这套理论还不能直接应用到线上产品中. 因为计算机中还要考虑数字大小限制, 例如k个32位整数作为数据, 通过Vandermonde矩阵生成校验块, 那校验块的数值几乎确定会溢出.

本文我们来解决这个问题, 看如何将EC的理论应用到计算机中, 保证计算不会溢出.

- 第一篇:原理 上一篇 🤔
- 第二篇:实现 我们在这 😁
- 第三篇:极限 🤔

思路Permalink

既然EC的存储过程就是对x取多个不同的值来计算y:

$$y=d1+d2x+d3x2+...+dkxk-1$$

恢复的过程是通过已知点的坐标来确定曲线方程:

$$y1=d1+1d2+12d3+…1k-1dky2=d1+2d2+22d3+…2k-1dky3=d1+3d2+32d3+…3k-1dk...$$

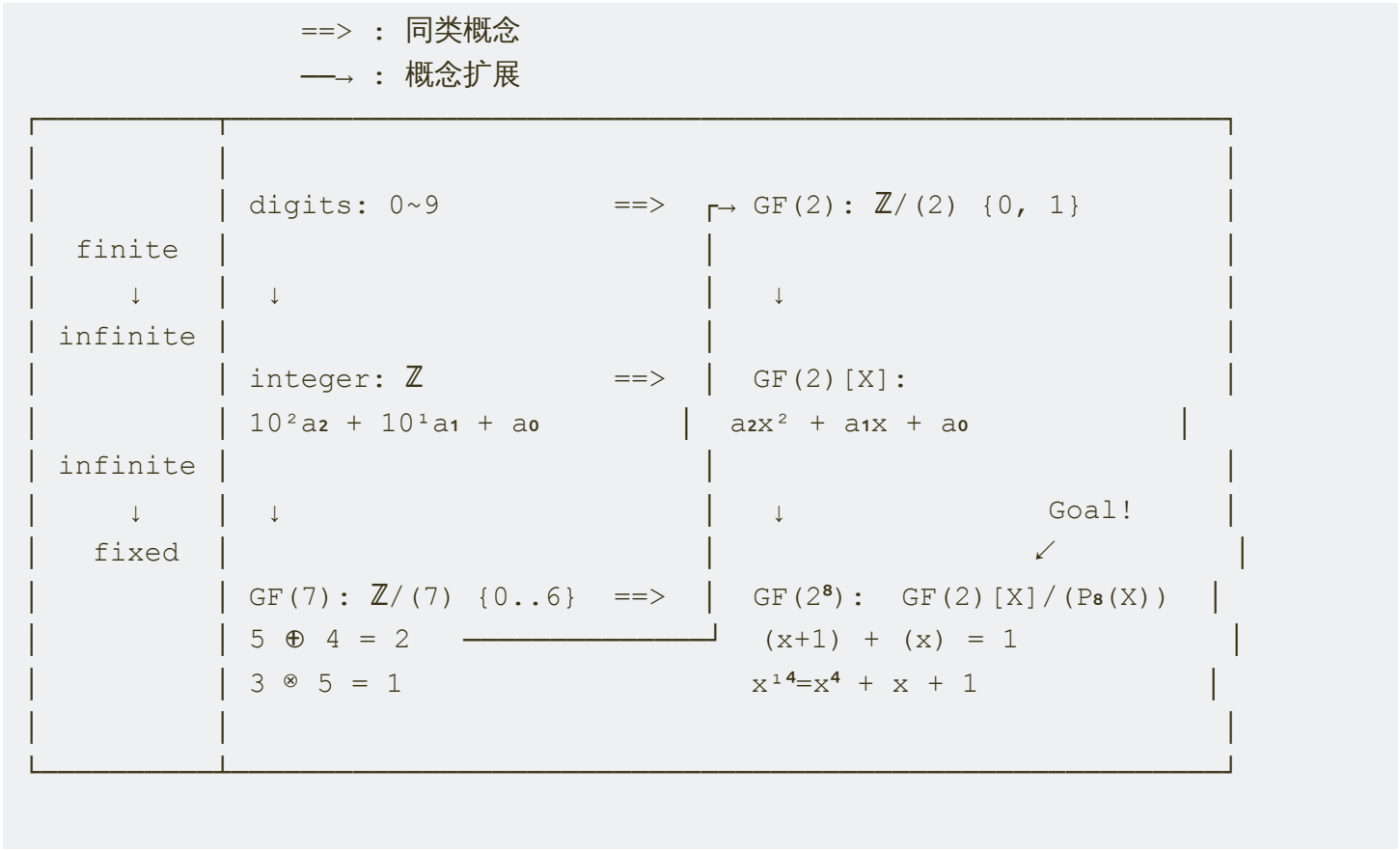
而编码和解码的过程都只需要加减乘除的四则运算(不需要开方), 那么, 除了数字之外, 任何对象, 只要能对其进行四则运算, 则都可以使用这种方法来实现冗余存储和备份.

本文中我们就将通过将数字替换成另一种满足四则运算的东西, 来实现计算机上的EC.

我们先从简单的例子开始, 打开新世界的大门.

感谢19世纪伟大数学家 伽罗华.

本文结构Permalink



GF(7): 伽罗华域Galois-Field GF(7)Permalink

上面我们提到的几个数学公式, 高次曲线, 多元一次方程组等, 他们之所以能正确的工作, 是因为他们依赖于一套底层的基础运算规则, 这就是四则运算: `+ - * /`

这听起来有点废话, 不用四则运算用什么?

其实我们平时熟知的四则运算, 并不是唯一的四则运算法则 例如在有1种四则运算可能是: 5 + 5 = 3 而不是10, 5 * 3 = 1 而不是15.

栗子4: 只有7个数字的新世界: GF(7)Permalink

🌰🌰🌰🌰

我们来尝试定义一个新的加法规则, 在这个新的世界里只有0~6这7个数字:

其他整数进来时都要被模7, 变成0~6的数字. 在这个模7的新世界里, 四则运算也可以工作:

模7新世界中的 加法Permalink

新的加法被表示为 ⊕ (这里原始的加法还是用+来表示):

$$a \oplus b \rightarrow (a+b) (mod 7)$$

它定义为: a ⊕ b的结果是 a + b后结果再对7取模. 例如:

1 ⊕ 1 = 2
5 ⊕ 2 = 0 (7 % 7 = 0)
5 ⊕ 5 = 3 (10 % 7 = 3)

在这个新世界里, 0 还是和以前的0很相像, 任何数跟0相加都不变:

0 ⊕ 3 = 3
2 ⊕ 0 = 2

0 在新世界 GF(7) 里被称为加法的**单位元**.

模7新世界中的 减法Permalink

然后我们再在模7的世界里定义减法. 减法的定义也很直接, 就是加法的逆运算了.

自然数里, -2 + 2 = 0, 我们称呼-2是2在加法上的逆元(通常称为相反数). 在模7的世界里,我们也很容易找到每个数的加法逆元,例如: 3 ⊕ 4 = 0 所以 4 和 3 就互为加法的逆元.

减法定义就是: $a \ominus b \to a \oplus (-b)$.

例如:

$3 \ominus 4 = 3 \oplus (-4) = 3 \oplus 3 = 6$
$2 \ominus 6 = 2 \oplus (-6) = 2 \oplus 1 = 3$

模7新世界中的 乘法 和 除法Permalink

在模7的新世界里, 我们也可以类似地定义1个乘法:

$$a \otimes b \to (a \times b) \bmod 7$$

例如:

$1 \otimes 5 = 5$ (5 % 7 = 5)
$3 \otimes 4 = 5$ (12 % 7 = 5)
$2 \otimes 5 = 3$ (10 % 7 = 3)
$0 \otimes 6 = 6 \otimes 0 = 0$

对于模7新世界的乘法$\otimes$来说, 1 是乘法的**单位元**, 也就是说$1 \otimes$ 任何数都是它本身.

我们也可以用类似的方法定义每个数字在乘法$\otimes$的逆元:

a的乘法逆元 $a^{-1} = b$, iff $a \otimes b = 1$.

例如:

$$3^{-1} = 5 (3 \times 5 \% 7 = 1) \quad 4^{-1} = 2 (4 \times 2 \% 7 = 1)$$

除法的定义就是: 乘以它的乘法逆元

栗子5: 模7新世界直线方程-1Permalink

🌰🌰🌰🌰🌰

现在我们有了新的加法和减法$\oplus$, $\ominus$ 我们可以像使用旧世界的加减法一样来使用$\oplus$, $\ominus$. 例如我们可以建立一个简单的, 斜率为1的直线方程:

$$y = x \oplus 3$$

新世界里这个直线上的点是: (x,y) ∈ [(0,3), (1,4), (2,5), (3,6), (4,0), (5,1), (6,2)] 只有7个.

如果把这条直线画到坐标系里, 它应该是这个样子的:

```
y = x ⊕ 3


  y
  ^
6 |      •
5 |    •
4 | •
3 •
2 |            •
1 |          •
0 +--------•-----> x
  0 1 2 3 4 5 6
```

栗子6: 模7新世界直线方程-2Permalink

🌰🌰🌰🌰🌰🌰

再加上新世界加减乘除四则运算, 我们可以在新世界里进行基本的代数运算了, 例如我们可以设定1个斜率为2的直线方程:

$$y=2\otimes x\oplus 3$$

新世界里这个直线上的点是: (x,y) ∈ [(0,3), (1,5), (2,0), (3,2), (4,4), (5,6), (6,1)] 这7个.

如果把这条直线画到坐标系里, 它应该是这个样子的:

```
y = 2 ⊗ x ⊕ 3


  y
  ^
6 |           •
5 | •
4 |        •
3 •
2 |      •
1 |            •
```

```
0 +---•---------> x
  0 1 2 3 4 5 6
```

栗子7: 模7新世界中的二次曲线方程Permalink

🌰🌰🌰🌰🌰🌰🌰

下面我们来建立1个稍微复杂1点的, 二次曲线的方程:

$$y=x2⊕x⊕2$$

这里 x^2 表示 $x \otimes x$

新世界里这个抛物线上的点集合是: $(x,y) \in [(0, 2) (1, 4) (2, 1) (3, 0) (4, 1) (5, 4) (6, 2)]$

如果把这条抛物线画到坐标系里, 它应该是这个样子的:

```
y = x^2 ⊕ x ⊕ 2


  y
  ^
6 |
5 |
4 | •       •
3 |
2 •           •
1 |   •   •
0 +-----•-------> x
  0 1 2 3 4 5 6
```

可以看出它的图像也遵循了旧世界抛物线的特性: **这条抛物线是以3为轴对称的**: 因为类似旧世界的多项式分解一样, 原方程也可以分解成:

$$y=(x⊕(−3))2=x2⊕(−6)x⊕9=x2⊕x⊕2$$

GF(7) 的EC实现Permalink

在这个模7的新世界里, 它满足我们旧世界里的四则运算法则, 我们已经可以使用前面(第一篇:原理)提到的 EC 的算法来编码或解码了:

假设模7新世界里我们的数据块 $d_1 = 3$, $d_2 = 2$, 对应上面的直线方程: $y = 2 \otimes x \oplus 3$

我们只要记住直线上2个点的位置, 就能把直线的方程恢复出来, 例如:

我们先记录直线上2个点: (1,5) 和 (3,2)

假设丢失的数据是 d_1, d_2 用 u_1, u_2 表示, 带入2个点的坐标, 得到一个二元一次方程组:

$$\{5=u2 \oplus u12=u2 \otimes 3 \oplus u1$$

2个方程左右分别相减消元:

$$5 \oplus (-2)=u2 \otimes (1 \oplus (-3)) \oplus u1 \oplus (-u1)5 \oplus 5=u2 \otimes (1 \oplus 4)3=u2 \otimes 5$$

最后得到 $u_2 = 3 \otimes 5^{-1} = 3 \otimes 3 = 2$.

将 $u_2 = 2$ 带入第1个方程: $5 = 2 \otimes 1 \oplus u_1$

得到 u_1: $u_1 = 5 \oplus (-2) = 3$

是不是跟普通的一次方程组解法完全一样!

至此, 我们用模7新世界的四则运算实现了之前的 EC . 并且我们保证了校验数据的大小是可控的: 不会大于7! 距离我们的目标又接近了1步.

模7下的四则运算构成了1个 伽罗华域 Galois-Field: GF(7). 简单来说, Galois-Field 是一个集合, 集合里的元素满足某种四则运算. 摘自wikipedia上的解释:

> A finite field is a set on which the operations of multiplication, addition, subtraction and division are defined
>
> 7是1个可选的数来通过取模的方式构造一个Galois-Field, 也可以选择模11或其他质数来构造1个 Galois-Field, 但是不能选择模一个合数来建立新的四则运算规则. 假设使用模6, 模6世界里面的2是6的一个因子, 它没有乘法逆元, 也即是说2 乘以 1~5任何一个数在模6的世界里都不是1.
>
> 没有乘法逆元就说明模6的世界里没有和旧世界里一样的除法, 不能构成一个完整的四则运算体系.

为了简化, 四则里还有几个方面没有提到, 例如乘法加法的分配率. 乘法和加法的结合律也必须满足, 才能在新世界里实现上面例子中的曲线方程等元素. 这部分也很容验证,在上面的模7新世界里是可以满足的.

现在我们有了 EC 的算法, 以及很多个可以选择的四则运算来限定数值的范围. 接下来要在计算机上实现,还有1步,就是: 模7虽然可取,但是它没有办法对计算机里的数字有效利用,因为计算机里的数是二进制的. 如果把数值限定到7或其他质数上,没有办法实现256或65536这样的区间的有效利用.

所以接下来我们需要在所有四则运算里选择一个符合计算机的二进制的四则运算, 作为实现 EC 计算的基础代数结构.

从现在开始, 我们要构造一个现实中可用的伽罗华域, 它比上面模7新世界稍微复杂一点, 得到这个域分为2步:

- 我们首先选择1个基础的, 只包含2个元素的 Galois-Field GF(2): {0, 1}.
- 再在这个 GF(2) 的基础上建立1个有256个元素的 Galois-Field GF(2^8).

GF(2): 模2的新世界: Galois-Field GF(2)Permalink

首先选择了最小的Galois-Field GF(2), 类似于前面模7的例子, GF(2) 里的四则运算的定义为结果模2. 它里面只有2个元素{0,1}:

在这个GF(2)里, 运算的规则也非常简单:

- 加法(刚好和位运算的异或等价):

```
0 ⊕ 0 = 0
0 ⊕ 1 = 1
1 ⊕ 0 = 1
1 ⊕ 1 = 0
```

  1的加法逆元就是1 本身.

- 乘法(刚好和位运算的与等价):

```
0 ⊗ 0 = 0
0 ⊗ 1 = 0
1 ⊗ 0 = 0
1 ⊗ 1 = 1
```

  1的乘法逆元就是1 本身. 0 没有乘法逆元.

以这个GF(2)为基础, 可以构建一个1-bit的 EC 算法了:)

后面的讨论全部是依赖于GF(2)中的加法和乘法, 为了看起来清楚, 不再使用⊕和⊗了, 直接使用传统的+, *来表示.

GF(2) 的EC实现Permalink

假设要存储的数据是: $d_1 = 1, d_2 = 1$

对应直线方程是: $y = 1 + x$

取2个点: $x_1 = 0, x_2 = 1$ 得到2个校验数据的值:

$y_1 = 1 + 0 = 1$
$y_2 = 1 + 1 = 0$

数据丢失后, 通过2个点的坐标找回直线方程系数:

$$1=u1+0u20=u1+1u2$$

解得 $u_1 = 1, u_2 = 1$.

下一步, 我们希望构建1个1 byte大小(2^8 个元素)的 Galois-Field GF(2^8), 在这个 GF(2^8) 里的 EC 中, 每个 d_j 和 y_i 的取值范围可以是0~255.

但首先我们需要能表示比GF(2)更多的值.

GF(2)[X] : GF(2) 作为系数的多项式Permalink

类似于我们只需要0~9这10个自然数, 就可以通过增加进位这个概念后, 扩展成能表示任意大小的10进制整数一样(或用0~7表示任意大小的8进制数), 我们通过类似的方法扩展{0,1}这2个数字, 表示更多的信息.

引入多项式:

使用 GF(2) 的元素作为系数, 定义1个多项式:

$$anxn+...+a2x2+a1x1+a0ai∈GF(2)={0,1}$$

系数 a_i 的四则运算还是遵循 GF(2) 的规则的, 而多项式的四则运算, 显然是基于它的系数的四则运算建立的. 例如多项式的加法:

- 因为 1 + 1 = 0, 所以: (x + 1) + (1) = x

- x的同指数幂的系数相加遵循系数的Field的加法规则, 1 + 1 = 0:

  $(x^2 + x + 1) + (x) = x^2 + 1$

- 2个相同的多项式相加肯定是0:

  $(x^2 + x + 1) + (x^2 + x + 1) = 0$

多项式的乘法和旧世界的多项式乘法类似, 仍然是通过乘法的分配率展开多项式:

$$(x+1)(x+1)=x(x+1)+(x+1)=x2+x+x+1=x2+1$$

多项式的除法依旧使用旧世界的多项式长除法法则, 唯一不同仍旧是系数的四则运算是基于GF(2)的, 例如:

$$x3+1x+1=x2+x+1$$

多项式的除法的取余计算也类似, 因为 $x^2 + x + 1 = x (x+1) + 1$, 所以: $(x^2 + x + 1) = 1 \bmod (x+1)$.

我们看到, **多项式之间也是满足加减乘除四则运算的**!

现在我们已经接触到了3种不同的四则运算:

GF(2)中元素的四则运算. GF(7)中元素的四则运算. GF(2) 多项式的四则运算.

也就是说我们可以建议一个备份和恢复多项式的EC:

GF(2)[X]的EC实现Permalink

通过一个简单的例子来构建一个存储2个多项式的EC, 将要存储的2个**多项式**作为**系数**建议一个**直线**:

$$Y=p1+p2X$$

例如要存储的2个多项式 $p_1 = x^2 + 1$, $p_2 = x$, 取直线上2个点, 例如X_1 X_2 分别取1, x, 代入直线方程得到2个Y的值:

$$\{Y1=p1+p2(1)=x2+1+x(1)=x2+x+1Y2=p1+p2(x)=x2+1+x(x)=1$$

- 存储过程: 存储 p_1, p_2, Y_1 Y_2; 也就是: ($x^2 + 1$, x, $x^2 + x + 1$, 1)
- 恢复过程: 假设 p_1, p_2 都丢了, 我们可以把Y_1, Y_2代入, 把p_1, p_2作为未知数建立一个方程组:

$$x2+x+1=p1+p21=p1+xp2$$

通过消元解方程, 两个方程相减, 恢复出p_2:

$$x2+x+1−1=p1−p1+p2−xp2x2+x=(1−x)p2x2+x=(1+x)p2x2+x1+x=p2x=p2$$

这样我们就实现了一个基于多项式的EC了.

但多项式的乘法依旧是有越界的问题的, 如果自然数到模7的方法一样, 我们需要把多项式的四则运算通过取模, 约束到一个可控的范围内.

还是取模!

GF(2^8)Permalink

GF(2)为系数的多项式的集合还不是一个伽罗华域, 因为缺少除法逆元. 就像整数全集也不是一个伽罗华域, 它也缺少除法逆元.

> 但在上面我们使用GF(2)的多项式来实现EC时, 没有触碰到没有乘法逆元这个问题. 就像可以用2个整数作为分子分母来表示分数一样, 用这种复合形式表示一个不存在的元素进行中间步骤的计算.
>
> 例如½不在整数集合, 但可以用1和2两个整数来表示出来.
>
> 而计算的最终结果都是恢复已存在的值, 所以分数形式的多项式最终都会被消去. 但这种分数形式的表示方法在实际使用中会造成很大不便.

现在我们需要找到1个质的多项式(Prime-Polynomial), 来替代GF(7)中**7**的角色, 最终得到1个有256个元素的多项式的伽罗华域 GF(2^8).

GF(2^8) 类似于模7的方式, 将多项式模一个质的多项式(Prime-Polynomial)来得到: GF(2)[X]/(P_8(X))

质的多项式(Prime-Polynomial) 简单说就是不可分解的多项式, 例如 $x^2 + 1$ 在**实数域**下就是 **质多项式**, 它无法分解成2个多项式乘积.

GF(2) 下的质多项式Permalink

- 1 是1个质多项式.
- x + 1 是1个质多项式. 因为它最高次幂是1, 肯定不能再拆分成2个多项式乘积了. 把所有GF(2)下的多项式对x + 1 取模, 只有2个可能的值: 0, 1.
- $x^2 + 1$ 不是一个质多项式, 它可以分解成$(x + 1)^2$.
- 2次的质多项式是: $P_2(x) = x^2 + x + 1$. 它在GF(2)的域中不能被拆分成2个1次多项式的乘积.

  我们可以像使用7对所有整数取模那样, 用它对所有多项式取模, 模它而产生的所有 **余多项式**, 是所有最高次幂小于2的多项式, 共有4个: 0, 1, x, (x + 1).

  模$P_2(x)$的多项式集合里, 同样满足多项式的四则运算.

  对于其他j次幂的质多项式 $P_j(x)$, 模$P_j(x)$的多项式集合里, 也刚好有2^j 个元素.

GF(2) 扩张成 GF(2^8)Permalink

为了扩张到 GF(2^8) 我们选择的8次幂的质多项式是: $P_8(x) = x^8 + x^4 + x^3 + x^2 + 1$

这个8次幂的质多项式,模它的所有余多项式,是所有最高次幂不超过7的多项式, 共256个, 它就是 GF(2) 到 GF(2^8) 的扩张.

> Field-Extension 域的扩张, 简单来说就是通过把一个域(例如GF(2)), 作为系数构建多项式, 再去模一个质多项式(如$P_8(x)$), 得到的**余多项式**集合(例如GF(2^8)).
>
> 这里从一个2个数字的集合GF(2), 扩张之后得到的却是一个多项式的集合, 因为在伽罗瓦的理论里, 数字跟多项式是没有区别的, 因为他们都可以进行四则运算. 就像我们前面也可以把多项式作为直线方程的系数一样.

然后我们还发现, 因为多项式的系数是GF(2)下的元素, 只能是0或1. 于是 **这些多项式和二进制数是有一一对应关系的**, 多项式中指数为i的项的系数就是二进制数第i位的值:

$$1 \rightarrow 1 \quad x^2+1 \rightarrow 101 \quad x^3+x \rightarrow 1010$$

扩张后的元素对应0~255这256个二进制数, $P_8(x)$ 对应:

- 二进制: 1 0001 1101
- 16进制: 0x11d

而GF(2^8)中的四则运算如下:

- 加法: a $\oplus$ b 对应多项式加法, 同时它表示的二进制数的加法对应: a ^ b
- 乘法: a $\otimes$ b 对应多项式的乘法(模$P_8(x)$):

总结一下GF(2^8)能够满足EC运算的几个性质:

- 加法单位元: 0
- 乘法单位元: 1
- 每个元素对加法都有逆元(可以实现减法): 逆元就是它本身((x+1) + (x+1) = 0)
- 每个元素对乘法都有逆元(除了0)(可以实现除法):$P_8(x)$是不可约的, 因此不存在a和b都不是0但ab=0; 又因为GF(2^8)只有255个非0元素, 因此对a,总能找到1个x使得 $a^x = a$. 所以 $a^{x-2}\ a = 1$. a^{x-2} 是a的乘法逆元.

  PS, 看到 $a^x = a$ 是不是想起了费马小定理? :) 费马小定理的群论的证明

- 乘法和加法满足分配率: 基于多项式乘法和加法的定义.

满足这些性质的四则运算, 就可以用GF(2^8)来建立高次曲线, 进而在GF(2^8)上实现EC.

标准EC的实现Permalink

以上讨论的是标准的EC的原理, 现在我们将以上的内容总结, 应用到实践上面.

- 标准的EC实现是基于GF(2)[X]/P(x), 一般基于 GF(2^8) 或 GF(2^{16}), GF(2^{32}), 分别对应1字节, 2字节或4字节. 最常见的是GF(2^8).
- GF(2^8) 下的加减法直接用异或计算, 不需要其他的工作.
- GF(2^8) 下的乘法和除法用查表的方式实现.

  首先生成 GF(2^8) 下对2的指数表和对数表, 然后把乘除法转换成取对数和取幂的操作:

  以 GF(2^8) 为例:

  - 生成指数表 2^0, 2^1, 2^2... 的表, 表中元素 p_i = `2^i`.
  - 生成对数表, 表中元素 $l_i = \log_2 i$.

  生成2个表的代码很简单, 用python表示如下:

```
power, log = [0] * 256, [0] * 256
n = 1
for i in range(0, 256):
```

```
    power[i] = n
    log[n] = i

    n *= 2

    # modular by the prime polynomial: P8(x) = x^8 + x^4 + x^3 + x^2 + 1
    if n >= 256:
        n = n ^ 0x11d

log[1] = 0 # log[1] is 255, but it should be 0
```

指数表:

```
01 02 04 08 10 20 40 80 1d 3a 74 e8 cd 87 13 26
4c 98 2d 5a b4 75 ea c9 8f 03 06 0c 18 30 60 c0
9d 27 4e 9c 25 4a 94 35 6a d4 b5 77 ee c1 9f 23
46 8c 05 0a 14 28 50 a0 5d ba 69 d2 b9 6f de a1
5f be 61 c2 99 2f 5e bc 65 ca 89 0f 1e 3c 78 f0
fd e7 d3 bb 6b d6 b1 7f fe e1 df a3 5b b6 71 e2
d9 af 43 86 11 22 44 88 0d 1a 34 68 d0 bd 67 ce
81 1f 3e 7c f8 ed c7 93 3b 76 ec c5 97 33 66 cc
85 17 2e 5c b8 6d da a9 4f 9e 21 42 84 15 2a 54
a8 4d 9a 29 52 a4 55 aa 49 92 39 72 e4 d5 b7 73
e6 d1 bf 63 c6 91 3f 7e fc e5 d7 b3 7b f6 f1 ff
e3 db ab 4b 96 31 62 c4 95 37 6e dc a5 57 ae 41
82 19 32 64 c8 8d 07 0e 1c 38 70 e0 dd a7 53 a6
51 a2 59 b2 79 f2 f9 ef c3 9b 2b 56 ac 45 8a 09
12 24 48 90 3d 7a f4 f5 f7 f3 fb eb cb 8b 0b 16
2c 58 b0 7d fa e9 cf 83 1b 36 6c d8 ad 47 8e 01
```

对数表(0没有以2为底的对数):

```
00 00 01 19 02 32 1a c6 03 df 33 ee 1b 68 c7 4b
04 64 e0 0e 34 8d ef 81 1c c1 69 f8 c8 08 4c 71
05 8a 65 2f e1 24 0f 21 35 93 8e da f0 12 82 45
1d b5 c2 7d 6a 27 f9 b9 c9 9a 09 78 4d e4 72 a6
06 bf 8b 62 66 dd 30 fd e2 98 25 b3 10 91 22 88
36 d0 94 ce 8f 96 db bd f1 d2 13 5c 83 38 46 40
1e 42 b6 a3 c3 48 7e 6e 6b 3a 28 54 fa 85 ba 3d
ca 5e 9b 9f 0a 15 79 2b 4e d4 e5 ac 73 f3 a7 57
07 70 c0 f7 8c 80 63 0d 67 4a de ed 31 c5 fe 18
e3 a5 99 77 26 b8 b4 7c 11 44 92 d9 23 20 89 2e
```

```
37 3f d1 5b 95 bc cf cd 90 87 97 b2 dc fc be 61
f2 56 d3 ab 14 2a 5d 9e 84 3c 39 53 47 6d 41 a2
1f 2d 43 d8 b7 7b a4 76 c4 17 49 ec 7f 0c 6f f6
6c a1 3b 52 29 9d 55 aa fb 60 86 b1 bb cc 3e 5a
cb 59 5f b0 9c a9 a0 51 0b f5 16 eb 7a 75 2c d7
4f ae d5 e9 e6 e7 ad e8 74 d6 f4 ea a8 50 58 af
```

在计算 GF(2^8) 中的乘法 将 a, b 通过查对数表和指数表实现:

$$a*b=2log2a+log2ba/b=2log2a−log2b$$

Galois-Field 的计算目前实现都是基于查表的, 所以选择大的域虽然可以一次计算多个字节, 但内存中随机访问一个大表也可能会造成cache miss太多而影响性能.

一般CPU都没有支持GF乘除法的指令, 但有些专用的硬件卡专门加速GF的乘除法. 最新的CPU 逐渐开始加入GF乘除法支持了.

有了加减乘除的计算支持, 下一步就是实现EC的编解码.

EC编码: 校验数据生成Permalink

通常使用1个矩阵来表示输入和输出的关系 (而不是像上文中只使用校验块的生成矩阵), 这里选择1, 2, 3..生成的 Vandermonde 矩阵:

[100…0010…0001…0⋰⋱000…1111…11222…2k−11332…3k−1⋰⋱1mm2…mk−1]×[d1d2d3...dk]=[d1d2d3...dky1y2y3...ym]

这个矩阵里上面是1个大小为k的单位矩阵, 表示 d_j 的输入和输出不变.

下面一部分是1个 m * k 的矩阵表示校验块的计算.

对要存储的k组数据, 逐字节读入, 形成 d_1, d_2… , 进行矩阵乘法运算, 得到最后要存储的 k 个数据块和 m 个校验块.

> 之所以把单位矩阵也放到编码矩阵上面, 看起来没有什么用, 只是把输入无变化的输出出来的这种风格, 原因在于在编码理论中, 并不是所有的生成的Code都是k个原始数据 和 m个校验数据的形式, 有些编码算法是将k个输入变成完全不1样的k+m个输出, 对这类编码算法, 需要1个k*(k+m)的编码矩阵来表示全部的转换过程. 例如著名的 Hamming-7-4 编码的编码矩阵(输入k=4, 输出k+m=7):

(1101101110000111010000100001)

EC解码Permalink

当数据损坏时, 通过生成解码矩阵来恢复数据:

对所有丢失的数据, 将它对应的第i行从编码矩阵中移除, 移除后, 保留编码矩阵的前k行, 构成1个k*k的矩阵.

例如第 2, 3个数据块丢失, 移除第2, 3行, 保留第k+1和k+2行: 这时矩阵中数据块和校验块的关系是:

[1000…00001…0⋰⋱0000…11111…1122223…2k−1]×[d1u2u3d4...dk]=[d1d4...dky1y2]

最后求逆矩阵, 和没有丢失的块相乘, 就可以恢复出丢失的数据块 u_2, u_3:

$$[d1u2u3d4...dk]=[1000…00001…0⋮⋮⋱⋮0000…11111…1122223…2k−1]−1×[d1d4...dky1y2]$$

因为只有 u_2, u_3 丢失了, 矩阵相乘时只需要计算逆矩阵的第2, 3行.

Vandermonde 矩阵的可逆性Permalink

在 第一篇:原理 中通提到:

> Vandermonde 矩阵的任意 `m * m` 的子矩阵, 是一个Generalized Vandermonde Matrix, 它在 x_i 都为正数时可以保证永远有唯一解.

但在GF(2^8) 中不成立.

举例来说, 以下矩阵是缺失 u_1, u_4 情况下的用来恢复数据的矩阵, 如果 $x^3 == 1$, 它就不可逆.

由于2是1个生成元, 容易看出, $x = 2^{85}$ 是1个不可逆的情况: $x^3 = 1$ 于是第1列和第4列完全一样.

$$[01000001000000111111xx2x3x4]$$

Cauchy 矩阵的可逆性Permalink

> Cauchy 矩阵的任意n行n列组成的矩阵都是可逆的, 因为任意子矩阵还是 Cauchy矩阵.

EC的实现到这里就结束了, 有了GF(2^8)的四则运算实现后, 再通过牛顿消元实现逆矩阵的求解, 就可以完整的实现出一套EC算法了.

在EC的计算中, 编解码是一个比较耗时的过程, 因此业界也在不断寻找优化的方法, 不论从理论算法上还是从计算机指令的优化上, 于是下一篇我们将介绍如何把EC实现为一个高效的实现.

EC擦除码系列:

- 第一篇:原理
- 第二篇:实现
- 第三篇:极限

本文链接: https://blog.openacid.com/storage/ec-2/

- [Vandermonde]: https://en.wikipedia.org/wiki/Vandermonde_matrix
- [Cauchy]: https://en.wikipedia.org/wiki/Cauchy_matrix
- [failure-rate]: https://www.backblaze.com/blog/hard-drive-reliability-stats-q1-2016/
- [RAID]: https://zh.wikipedia.org/wiki/RAID
- [RAID-5]: https://zh.wikipedia.org/wiki/RAID#RAID_5
- [RAID-6]: https://zh.wikipedia.org/wiki/RAID#RAID_6
- [Finite-Field]: https://en.wikipedia.org/wiki/Finite_field
- [Galois-Field]: https://en.wikipedia.org/wiki/Finite_field
- [Reed-Solomon]: https://en.wikipedia.org/wiki/Reed%E2%80%93Solomon_error_correction
- [Erasure-Code]: https://en.wikipedia.org/wiki/Erasure_code
- [Prime-Polynomial]: https://en.wikipedia.org/wiki/Irreducible_polynomial
- [Field-Extension]: https://en.wikipedia.org/wiki/Field_extension
- [Complex-Number]: https://en.wikipedia.org/wiki/Irreducible_polynomial#Field_extension
- [Hamming-7-4]: https://en.wikipedia.org/wiki/Hamming(7,4)
- [Generator-Matrix]: https://en.wikipedia.org/wiki/Generator_matrix
- [第一篇:原理]: https://blog.openacid.com/storage/ec-1
- [第二篇:实现]: https://blog.openacid.com/storage/ec-2
- [第三篇:极限]: https://blog.openacid.com/storage/ec-3
- [费马小定理的群论的证明]: https://en.wikipedia.org/wiki/Proofs_of_Fermat%27s_little_theorem#Proofs_using_group_theory