CRC背后的故事

> 本文为翻译文章，笔者翻译水平有限，原文详见[Understanding CRC]

介绍

去年的时候，在一些CTF赛事中出现了与`CRC`相关的问题。问题的关键就是找到一个`x`使得`flag{x}`的`CRC`值为`x`。在解题的过程中，我们发现大多数人并不了解`CRC`的性质。实际在网络应用中，通常使用`CRC`作为冗余校验码，但是也仅限于使用现有的函数接口计算文件或数据的`CRC`值，而没有深入了解`CRC`的性质。

我对`CRC`的本质也不太理解，于是我重新学习并写下了这篇文章。

背景知识

Finite Field

在理解CRC之前，必须要了解有限域，这是理解CRC的数学基础。

有限域是指满足域条件的有限集合，那什么是域呢？简单来说，域是加减乘除都有定义且满足特定规则的集合。有限域最常见的例子是当 p 为素数时，整数对 p 取模。其所定义的集合为$\{0,1,...,p-1\}$,容易看出加减乘除四个运算在这个集合上都得到了满足。

关于有限域，有很多有趣的性质

- 有限域的阶(有限域中元素的个数)是一个素数的幂，对于任意一个素数`p`和任意一个正整数`k`，在同构意义下存在唯一的p^k阶的有限域，并且所有元素都是方程$x^{p^k}-x=0$的根，该有限域的特征为`p`。
- 相同大小的有限域是同构的(阶相同)。两个同构的有限域之间存在一一对应关系，同时加减乘除的关系保持不变

> 有限域的更多性质详见 [有限域]

对于阶为`p`的有限域，我们可以将其表示为F_p，或者$GF(p)$，本文将采用第二种表达方式。对于$GF(p)$我们很容理解为是 $\bmod\ p$的有限域，但是对于$GF(p^k)$我们该如何定义呢?

> 如果单纯的理解为 $\bmod\ p^k$的集合，则无法满足除法运算，比如计算(p+1)/p 是，需要对p求逆元，但是p 在 p^k 下无逆元

为了解决这个问题，我们要对$GF(p)$进行扩展。$GF(p)[X]$是以$GF(p)$中元素为系数关于变量`X`的多项式集合。例如，$GF(2)[X]$的系数为0或者1,X^2+1,X^3+X^2+1为$GF(2)[X]$的元素。注意如果系数出现2的倍数，会被约去。如X^2+2X+1等价于X^2+1。

现在考虑$GF(p^k)$,首先我们需要一个k阶的不可约多项式P来定义$GF(p^k)$。不可约多项式指的是无法分解的多项式。例如在$GF(2)[X]$上X^3+X^2+1是一个不可约多项式，而X^2+1则不是，因为其可以表示为$(X+1)^2$,显然是可以被分解为$(X+1)(X+1)$。对于一个确定的不可约多项式P，如果某个多项式的一项aX^l的次数l大于k,我们可以通过加上一个$-aX^{l-k}P$消去高次项，消去所有高次项后仅保留小于等于$k-1$次的项，且所有项的系数为$\{0,1,...p-1\}$。以这样方式定义的集合，记作$GF(p)[X]/(P)$，即$GF(p^k)$，其中P为选定的一个k次不可约多项式。

例如，令$GF(2^3)=GF(2)[X]/(X^3+X+1)$,在这个有限域上$X^2+X+1$和$X^2+1$的相乘的过程如下:

$$(X^2+X+1)(X^2+1)=X^4+X^3+X+1=X^4+X^3+X+1-X(X^3+X+1)=X^3+X^2+X+1=X^3+X$$

CRC

Cyclic redundancy check，简单的来说CRC就是将给定的字符串表示为$GF(2^k)$上的值的过程。查看[Wikipedia]可以看到对于不同的CRC，使用的不可约多项式(irreducible polynomial)也是不一样的。但是如果仅仅看下面的C代码很难理解将字符串映射到$GF(2^k)$这个变化的过程。

```
unsigned int crc32b(unsigned char *message) {
   int i, j;
   unsigned int byte, crc, mask;

   i = 0;
   crc = 0xFFFFFFFF;
```

```
    while (message[i] != 0) {
       byte = message[i];            // Get next byte.
       crc = crc ^ byte;
       for (j = 7; j >= 0; j--) {    // Do eight times.
          mask = -(crc & 1);
          crc = (crc >> 1) ^ (0xEDB88320 & mask);
       }
       i = i + 1;
    }
    return ~crc;
}
```

下面分析一下这段代码是如何写出来的。首先考虑$GF(2)$的加法运算，可以发现实际上$GF(2)$加法运算和XOR运算没有区别。比如$0+0=0, 1+0=0, 0+1=0, 1+1=0$。如果我们扩展次范围并考虑$GF(2)[X]$上的加法，需要将次数相同项的系数异或作为新的系数。在这里，我们使用二进制的形式表示多项式的系数，于是将表示两个多项式的整数进行XOR运算，等价于在$GF(2)[X]$两个多项式想加。

在上述代码中,`unsigned int`代表了$GF(2^{32})$，在`while`循环中，`message[i]`的值被连续添加到以`0xFFFFFFFF`为初值的`crc`中。

再看看for循环，如果`crc&1`为`0`，则mask为`0`，反之mask为`0xffffffff`。而mask是要和`0xEDB88320`想与然后和右移后的crc异或的，那么为什么只有在crc&1非零的时候才会对crc的值进行改变呢?

> 手撸过多项式乘法的代码的小伙伴可能会比较熟悉，但是没写过的问题也不大

实际上，此过程中crc的`LSB`(最低位)是X^{31}的系数，而`MSB`(最高位)是一个常数项。换言之，位数越低，对应项次数越高。for循环将message[i]的8位添加到了crc中，每一右移运算相当于将每一位的系数增大了1，等价于将多项式乘了个`X`，那么当右移前的`LSB`为1时(X^{31}系数不为零)，右移后则变为X^{32}，为了消除系数大于等于的32的项，需要减去一个$X \cdot P, P$为生成多项式，而代码中则`XP`的值为`0xEDB88320`。到此相信读者已经理解了这段`CRC`代码的原理。

重新梳理一个整个算法的流程，设需要计算`CRC`的`message`长度为`l`，其二进制形式为`msg`,第`i`个字节对应的二进制形式为msg_i,由于`message[i]`是和`crc`低8位异或,相当于将msg_i乘X^{24}然后和`crc`相加，循环里则是相当于乘X^8(在有限域上的乘法)，那么最后需要计算的`CRC`为$msg \times X^{32} + \text{0xFFFFFFFF} \times X^{8l} + \text{0xFFFFFFFF}$，这里面第一个`0xFFFFFFFF`为crc的初始值，由于每次循环右移8位，一共`l`次，等价于$\text{0xFFFFFFFF} \times X^{8l}$,第二个`0xFFFFFFFF`是最后的`~crc`表示俺位翻转。

但是有的`CRC`代码，它们的实现方式和这种方式不一定相同。例如，crc的初始值不是`0xffffffff`而是`0`，`MSB`表示X^{31}，`LSB`表示X^0(这种方式称为Normal form)，但是通常使用的还是本文所描述的方式，初始值为`0xffffffff`，`lsb`对应高次系数，最后再异或`0xfffffff`。

Playing With CRC

在此为了深入的理解`CRC`,我们采用python实现的CRC32,即$GF(2^{32})[X]/P$。

```python
poly = 0x104C11DB7

def normalize(x):
    while x.bit_length() > 32:
        x ^= poly << (x.bit_length() - 33)
    return x

def mult(x, y):
    res = 0
    for i in range(32):
        if y & (1 << i) != 0:
            res ^= (x << i)
    return normalize(res)

def bytes_to_gf32(s):
    val = 0
    for c in s:
        rev = int(format(c, '08b')[::-1], 2)
        val = (val << 8) | rev
```

```
    return normalize(val)

def crc32(s):
    l = len(s)
    m = bytes_to_gf32(s)
    return normalize((m << 32) ^ (0xFFFFFFFF << (8 * l)) ^ 0xFFFFFFFF)

def crc32b(message):
    crc = 0xFFFFFFFF
    for i in range(len(message)):
        byte = message[i]
        crc = crc ^ byte
        for j in range(8):
            if crc & 1:
                crc = (crc >> 1) ^ 0xEDB88320
            else:
                crc >>= 1
    return crc ^ 0xFFFFFFFF

message = b"test"

crc1 = crc32(message)
crc2 = crc32b(message)

print(format(crc1, '032b'))
print(format(crc2, '032b')[::-1])
```

这里的`crc32`是根据$msg \times X^{32} + \mathrm{0xFFFFFFFF} \times X^{8l} + \mathrm{0xFFFFFFFF}$表达式写的代码，而`crc32b`则是根据c代码重写的，运行此脚本，得到如下的结果。

```
> python .\test.py
00110000011111101111111000011011
00110000011111101111111000011011
```

现在我们回顾一下，在文章开始时引入的问题，找到一个`x`使得`flag{x}`的`CRC`值为`x`。在`CRC32`下，不妨令`x`为4个字节,根据$msg \times X^{32} + \mathrm{0xFFFFFFFF} \times X^{8l} + \mathrm{0xFFFFFFFF}$，于是我们得到"$flag\{x\}$"$\times X^{32} + \mathrm{0xFFFFFFFF} \times X^{8l} + \mathrm{0xFFFFFFFF} = x$，那么需要解这个方程，便能够得到预期的`x`,但是需要定义除法。

幸运的是在有限域$GF(p^k)$上，非零元素`a`有$a^{p^k} - a = 0$,于是有$a \cdot a^{p^k - 2} = 1$，所以`a`的逆元为$a^{p^k - 2}$。

于是，可以写出下面的代码:

```
def pow(x, y):
    if y == 0:
        return 1
    elif y == 1:
        return x
    else:
        res = pow(x, y // 2)
        res = mult(res, res)
        if y & 1:
            res = mult(res, x)
        return res

def inverse(x):
    return pow(x, 2 ** 32 - 2)

const = crc32(b"flag{\0\0\0\0}")
coef = normalize((1 << 40) ^ 1)
x = mult(const, inverse(coef))

print(format(x, '032b')[::-1])

# 01110011 10011011 01000101 00000111
```

```python
#     0x73     0x9b     0x45     0x07

print(hex(x))
print(hex(bytes_to_gf32(b"\x07\x45\x9b\x73")))
print(hex(crc32(b"flag{\x07\x45\x9b\x73}")))
print(hex(crc32b(b"flag{\x07\x45\x9b\x73}")))
```

输出结果为:

```
python .\test.py
01110011100110110100010100000111
0xe0a2d9ce
0xe0a2d9ce
0xe0a2d9ce
0x739b4507
```

于是预期的`x`为`\x07\x45\x9b\x73`

总结

我们花了一些时间来了解`CRC`的数学基础，并深入了解了`CRC`,并找出了某些字符串的`CRC`值等于字符串自身情况。除了`CRC`以外，有限域本身也是一个经常在其他地方使用的概念，因此希望在以后出现混淆的时候可以参考这篇文章。

参考资料

1. https://en.wikipedia.org/wiki/Finite_field
2. https://en.wikipedia.org/wiki/Cyclic_redundancy_check
3. https://stackoverflow.com/questions/21001659/crc32-algorithm-implementation-in-c-without-a-look-up-table-and-with-a-public-li
4. https://zh.wikipedia.org/wiki/%E6%9C%89%E9%99%90%E5%9F%9F