Bitwise Operators (in code)

Monday, September 28, 2020 11:44 AM

Background

The first part of the class is over - this dealt with crypto before computers.

Now, we'll talk about the huge leap forward with the use of computers.

Computers store information in binary (0s and 1s)

Next set of ciphers we are going to look at are the

Modern Symmetric Encryption Systems

In detail, we'll look at DES (Data Encryption Standard), the US Government symmetric encryption standard from the mid 1970s through 1998 or so.

Then, we'll look AES (Advanced Encryption Standard), which is the current symmetric encrytion standard (for the US), which took over for DES.

Modern symmetric systems are typically block ciphers. A block cipher breaks up the data into n bit blocks. (Bits are 0 or 1.) Plaintext is assumed to be some bitstring. The bitstring is then broken up into chunks, each of which is the same size, in bits.

DES used 64 bit blocks

AES uses 128 or 192 or 256 bit blocks

The term symmetric means that there is a secret key that both the sender and recipient share, and that same secret key can be used to both encrypt and decrypt.

DES used a 56 bit key

AES uses either 128, 192 or 256 bit key (same size as the block).

Most of the operations in modern day ciphers are usually either look up tables, some sort of mathematics, or using standard bitwise operators.

This lecture will cover the standard bitwise operators and how to use them in C.

Bitwise Operators:

AND	&
OR	|
XOR	^
NOT	~

All integer type variables (int/short/byte/long, etc.)

are stored in binary. Two types binary (unsigned and two's complement). By default, things are in two's complement, but if you put the keyword unsigned in front of a type (at least in some languages), you'll get the unsigned version.

In the computer, for example, the number 23 is stored as

0000…10111

Each bit is worth 2^i, where I represents how far from the right end that bit is. So the number above is $2^4 + 2^2 + 2^1 + 2^0 = 23$.

Plaintext: 00000101 01101011

In practice, we must have a method of converting our original message (maybe in ascii) to a bitstring…there are standard ways to do this, but this won't be a focus of our class.

Use of bitwise and:

X: 1010 0111 128 + 32 + 4 + 2 + 1 = 167

```
Y: 0011 0010 &        32 + 16 + 2 = 50
---------------
    0010 0010         32 + 2 = 34
```

bitwise or

```
X: 1010 0111          128 + 32 + 4 + 2 + 1 = 167
Y: 0011 0010 |        32 + 16 + 2 = 50
---------------
    1011 0111         128 + 32 + 16 + 4 + 2 + 1 = 183
```

bitwise xor (it's 1 if and only if exactly one input is 1.)

```
X: 1010 0111          128 + 32 + 4 + 2 + 1 = 167
Y: 0011 0010 |        32 + 16 + 2 = 50
---------------
    1001 0101         128 + 16 + 4 + 1 = 149
```

Properties of XOR: # of bits set to 1 in the xor represents the number of different bits in the two inputs.

if x ^ y = z, then x ^ z = y and y ^ z = x. (any two things in an xor equation, always xor to the third thing.

To undo an xor, just do the same exact xor twice:

(x ^ y) ^ y = x, (also commutative)

A one-time pad in cryptography, is nothing but an XOR with a secret key that only gets used once. Theoretically, these are unbreakable, since there are no possible patterns. The US military does use these…

Just XOR the plaintext with the key. The key must be of equal length (or greater) to the plaintext. To decode, just XOR with the key again…

Next operators

---

a >> b, this is a right shift of a by b bits
a << b, this is a left shift of a by b bits

For example

108 >> 3 = 1101100 chop off last 3 bits -> 1101 (13)

11 << 4 = 1011 add 4 bits to the right 10110000 (176)

Cyclic right shift

Here instead of chopping off the bits, we put them back at the end of the number.

11101010 cyclicly shifted by 3 spots to the right:

11101 010 (would normally get chopped off)

But in a cyclic shift, it shows up on the left side:

010 11101 --> 0101 1101 is the final result. (If the block size was 8)

1<<8 is 100000000
if we subtract 1 from it, we get:
11111111

Basically ((1<<k) - 1) is a bitstring k 1s.