Operations on Bit Strings

A bit string is merely a sequence of bits (0s and 1s).

Let Z_2^n denote the set of bit strings of length n.

- We may think of a bit string in Z_2^n as a single integer in the range $[0, 2^n-1]$, and perform integer operations with it.
    - Many public-key algorithms do this.
- There are also operations that apply directly to bit strings.
    - Secret-key algorithms make extensive use of some of these operations.

Definitions of operations: Let $\boldsymbol{a} = a_1 a_2 \ldots a_n$ and $\boldsymbol{b} = b_1 b_2 \ldots b_n$ be elements of Z_2^n. Let

$$\mathbf{0} = 00\ldots0 \in Z_2^n \quad \text{and} \quad \mathbf{1} = 11\ldots1 \in Z_2^n.$$

We define

$\neg\boldsymbol{a} = c_1 c_2 \ldots c_n$, where $c_i = 1$ exactly when $a_i = 0$.

$\boldsymbol{a} \vee \boldsymbol{b} = s_1 s_2 \ldots s_n$, where $s_i = 1$ if $a_i = 1$ **or** $b_i = 1$ (or both).

$\boldsymbol{a} \wedge \boldsymbol{b} = p_1 p_2 \ldots p_n$, where $p_i = 1$ if $a_i = 1$ **and** $b_i = 1$.

$\boldsymbol{a} \oplus \boldsymbol{b} = x_1 x_2 \ldots x_n$, where $x_i = 1$ if $a_i = 1$ **or** $b_i = 1$, but **not both**.

Of these operations, $\oplus$ (exclusive or, or *xor*) is by far the most useful in cryptography. We concentrate on *xor*.

Properties of *xor*:

i) $\oplus$ is commutative, i.e., $\boldsymbol{a} \oplus \boldsymbol{b} = \boldsymbol{b} \oplus \boldsymbol{a}$.

ii) $\oplus$ is associative, i.e., $(\boldsymbol{a} \oplus \boldsymbol{b}) \oplus \boldsymbol{c} = \boldsymbol{a} \oplus (\boldsymbol{b} \oplus \boldsymbol{c})$.

In view of (ii), we can define $\boldsymbol{a}_1 \oplus \boldsymbol{a}_2 \oplus \ldots \oplus \boldsymbol{a}_n$ unambiguously.

iii) $\boldsymbol{a}_1 \oplus \boldsymbol{a}_2 \oplus \ldots \oplus \boldsymbol{a}_n$ has a 1 in bit i exactly when an <u>odd</u> number of $\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_n$ have 1s in bit i.

iv) $\boldsymbol{a} \oplus \mathbf{0} = \boldsymbol{a}$.

v) $\boldsymbol{a} \oplus \boldsymbol{a} = \mathbf{0}$.

vi) $\boldsymbol{a} \oplus \mathbf{1} = \neg\boldsymbol{a}$.

vii) $\boldsymbol{a} \oplus \neg\boldsymbol{a} = \mathbf{1}$.

viii) $\boldsymbol{x} \oplus \boldsymbol{a} = \boldsymbol{x} \oplus \boldsymbol{b} \Rightarrow \boldsymbol{a} = \boldsymbol{b}$.
$\boldsymbol{a} \oplus \boldsymbol{x} = \boldsymbol{b} \oplus \boldsymbol{x} \Rightarrow \boldsymbol{a} = \boldsymbol{b}$.

ix) The following are equivalent:
$\boldsymbol{a} \oplus \boldsymbol{b} = \boldsymbol{c}$,
$\boldsymbol{a} \oplus \boldsymbol{c} = \boldsymbol{b}$,
$\boldsymbol{b} \oplus \boldsymbol{c} = \boldsymbol{a}$.

We will say that $\boldsymbol{r} = r_1 r_2 \ldots r_n$ is a <u>random sequence</u> of bits if

- $prob(r_i = 1) = 2$ for all i, and
- r_i is independent of $r_1 \ldots r_{i-1} r_{i+1} \ldots r_n$, for all i.

The following property of *xor* is especially important for cryptography.

x) If $\boldsymbol{r}$ is a random sequence of bits, and $\boldsymbol{a}$ is <u>any</u> sequence of bits (of the same length), then $\boldsymbol{a} \oplus \boldsymbol{r}$ is a random.

Note that $\boldsymbol{a}$ need not be random.

Note property (x) doesn't hold for *and* ($\wedge$) or for *or* ($\vee$).

- Consider $\boldsymbol{a} \wedge \boldsymbol{r}$, where $\boldsymbol{r}$ is random. $\boldsymbol{a} \wedge \boldsymbol{r}$ is random in those bits in which $\boldsymbol{a}$ is 1, but is completely determined in those bits in which $\boldsymbol{a}$ is 0.

- Even if $\boldsymbol{a}$ is also random, $\boldsymbol{a} \wedge \boldsymbol{r}$ is not random. It is an independent sequence of bits in which each bit is 1 with probability 1/4.

This explains in part why *xor* is so useful in cryptography.

Suppose a and b are independent bits with

$$prob(a=1) = p \text{ and } prob(b=1) = q.$$

Then $prob(a \oplus b = 1) = p(1-q) + (1-p)q = p + q - 2pq$.

As long as $p \neq 0{,}1$ and $q \neq 0{,}1$, $p + q - 2pq$ is closer to 0.5 than either p or q. To see this, note

$$\begin{aligned} \left|0.5 - (p + q - 2pq)\right| &= 2\left|pq - p - q + 0.25\right| \\ & 2\left|p - 0.5\right| \cdot \left|q - 0.5\right| \end{aligned}$$

Since $q \neq 0{,}1$, $2\left|q - 0.5\right| < 1$, and

$$\left|0.5 - (p + q - 2pq)\right| < \left|p - 0.5\right|.$$

Likewise, $\left|0.5 - (p + q - 2pq)\right| < \left|q - 0.5\right|$.

So $\left|0.5 - (p + q - 2pq)\right| < min\left(\left|p - 0.5\right|, \left|q - 0.5\right|\right)$.

In other words, $prob(a \oplus b = 1)$ is closer to 0.5 than either $prob(a=1)$ or $prob(b=1)$. In fact, if $prob(a=1)$ and $prob(b=1)$ are close to 0.5, then $prob(a \oplus b = 1)$ is <u>much</u> closer to 0.5.

Examples:

$prob(a=1)$	$prob(b=1)$	$prob(a \oplus b=1)$
0.6	0.4	0.52
0.6	0.7	0.46
0.52	0.52	0.4992
0.52	0.498	0.50008

If we have two independent bit sequences $\boldsymbol{a}$ and $\boldsymbol{b}$ (independent of each other), each somewhat close to random, then $\boldsymbol{a} \oplus \boldsymbol{b}$ is an independent bit sequence that is far closer to random than either **a** or **b**.

One-Time Pad: This is essentially a perfect method of encryption.

- It cannot be broken as long as the key remains secret.
- But its practicality is limited by the fact that the key is as long as the message.

To encrypt a plaintext $\boldsymbol{p}$ (a sequence of bits), we choose the key as random bit sequence $\boldsymbol{r}$ of the same length as $\boldsymbol{p}$. We encrypt by

$$\boldsymbol{c} = \mathrm{E}_{\boldsymbol{r}}(\boldsymbol{p}) = \boldsymbol{p} \oplus \boldsymbol{r}.$$

$\boldsymbol{c} = \boldsymbol{p} \oplus \boldsymbol{r}$ implies $\boldsymbol{p} = \boldsymbol{c} \oplus \boldsymbol{r}$, so the decryption function is the same as the encryption function (xor with $\boldsymbol{r}$).

Once we have encrypted a message with $\boldsymbol{r}$, we never use it again for encryption. (This is why it is called a <u>one-time</u> pad.)

Why does this work?

- Since $\boldsymbol{r}$ is random, $\boldsymbol{c} = \boldsymbol{p} \oplus \boldsymbol{r}$ is a random sequence of bits. It contains no information about $\boldsymbol{p}$.
- But note $\boldsymbol{c} = \boldsymbol{p} \oplus \boldsymbol{r}$ implies $\boldsymbol{r} = \boldsymbol{p} \oplus \boldsymbol{c}$, so if an intruder intercepts the ciphertext and somehow manages to discover the plaintext, he can compute the key $\boldsymbol{r}$. If the same key $\boldsymbol{r}$ were used a second time, he could decrypt the ciphertext.
- Even if the intruder never discovers any plaintext, if two plaintexts $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$ were encrypted with the same key $\boldsymbol{r}$, and both ciphertexts $\boldsymbol{p_1} \oplus \boldsymbol{r}$ and $\boldsymbol{p_2} \oplus \boldsymbol{r}$ were intercepted, the intruder could compute $(\boldsymbol{p_1} \oplus \boldsymbol{r}) \oplus (\boldsymbol{p_2} \oplus \boldsymbol{r}) = \boldsymbol{p_1} \oplus \boldsymbol{p_2}$, and determine if the two plaintexts were identical, or highly similar.