

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 $\mathbf{a} = a_1a_2\dots a_n$ and $\mathbf{b} = b_1b_2\dots b_n$ be elements of Z_2^n . Let

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

We define

- $\neg \mathbf{a} = c_1c_2\dots c_n$, where $c_i = 1$ exactly when $a_i = 0$.
- $\mathbf{a} \vee \mathbf{b} = s_1s_2\dots s_n$, where $s_i = 1$ if $a_i = 1$ **or** $b_i = 1$ (or both).
- $\mathbf{a} \wedge \mathbf{b} = p_1p_2\dots p_n$, where $p_i = 1$ if $a_i = 1$ **and** $b_i = 1$.
- $\mathbf{a} \oplus \mathbf{b} = x_1x_2\dots x_n$, 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*:

- \oplus is commutative, i.e., $\mathbf{a} \oplus \mathbf{b} = \mathbf{b} \oplus \mathbf{a}$.
- \oplus is associative, i.e., $(\mathbf{a} \oplus \mathbf{b}) \oplus \mathbf{c} = \mathbf{a} \oplus (\mathbf{b} \oplus \mathbf{c})$.
In view of (ii), we can define $\mathbf{a}_1 \oplus \mathbf{a}_2 \oplus \dots \oplus \mathbf{a}_n$ unambiguously.
- $\mathbf{a}_1 \oplus \mathbf{a}_2 \oplus \dots \oplus \mathbf{a}_n$ has a 1 in bit i exactly when an odd number of $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n$ have 1s in bit i .
- $\mathbf{a} \oplus \mathbf{0} = \mathbf{a}$.
- $\mathbf{a} \oplus \mathbf{a} = \mathbf{0}$.
- $\mathbf{a} \oplus \mathbf{1} = \neg \mathbf{a}$.
- $\mathbf{a} \oplus \neg \mathbf{a} = \mathbf{1}$.
- $\mathbf{x} \oplus \mathbf{a} = \mathbf{x} \oplus \mathbf{b} \Rightarrow \mathbf{a} = \mathbf{b}$.
 $\mathbf{a} \oplus \mathbf{x} = \mathbf{b} \oplus \mathbf{x} \Rightarrow \mathbf{a} = \mathbf{b}$.
- The following are equivalent:

$$\begin{aligned}\mathbf{a} \oplus \mathbf{b} &= \mathbf{c}, \\ \mathbf{a} \oplus \mathbf{c} &= \mathbf{b}, \\ \mathbf{b} \oplus \mathbf{c} &= \mathbf{a}.\end{aligned}$$

We will say that $\mathbf{r} = r_1 r_2 \dots r_n$ is a random sequence of bits if

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

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

- x) If \mathbf{r} is a random sequence of bits, and \mathbf{a} is any sequence of bits (of the same length), then $\mathbf{a} \oplus \mathbf{r}$ is a random.

Note that \mathbf{a} need not be random.

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

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

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

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

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

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

Since $q \neq 0,1$, $2|q - 0.5| < 1$, and

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

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

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

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

Examples:

$\text{prob}(a=1)$	$\text{prob}(b=1)$	$\text{prob}(a \oplus b = 1)$
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 \mathbf{a} and \mathbf{b} (independent of each other), each somewhat close to random, then $\mathbf{a} \oplus \mathbf{b}$ is an independent bit sequence that is far closer to random than either \mathbf{a} or \mathbf{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 p (a sequence of bits), we choose the key as random bit sequence r of the same length as p . We encrypt by

$$c = E_r(p) = p \oplus r.$$

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

Once we have encrypted a message with r , we never use it again for encryption. (This is why it is called a one-time pad.)

Why does this work?

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