Operations on Bit Strings

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

Let \( \mathbb{Z}_2^n \) denote the set of bit strings of length \( n \).

- We may think of a bit string in \( \mathbb{Z}_2^n \) 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 \( a = a_1a_2...a_n \) and \( b = b_1b_2...b_n \) be elements of \( \mathbb{Z}_2^n \). Let

\[
0 = 00...0 \in \mathbb{Z}_2^n \quad \text{and} \quad 1 = 11...1 \in \mathbb{Z}_2^n .
\]

We define

\[
\neg a = c_1c_2...c_n , \quad \text{where} \quad c_i = 1 \text{ exactly when } a_i = 0 .
\]

\[
a \lor b = s_1s_2...s_n , \quad \text{where} \quad s_i = 1 \text{ if } a_i = 1 \text{ or } b_i = 1 \text{ (or both)} .
\]

\[
a \land b = p_1p_2...p_n , \quad \text{where} \quad p_i = 1 \text{ if } a_i = 1 \text{ and } b_i = 1 .
\]

\[
a \oplus b = x_1x_2...x_n , \quad \text{where} \quad x_i = 1 \text{ if } a_i = 1 \text{ or } b_i = 1 , \text{ but not both}.
\]

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., \( a \oplus b = b \oplus a \).

ii) \( \oplus \) is associative, i.e., \( (a \oplus b) \oplus c = a \oplus (b \oplus c) \).

  In view of (ii), we can define \( a_1 \oplus a_2 \oplus ... \oplus a_n \) unambiguously.

iii) \( a_1 \oplus a_2 \oplus ... \oplus a_n \) has a 1 in bit \( i \) exactly when an odd number of \( a_1, a_2, ..., a_n \) have 1s in bit \( i \).

iv) \( a \oplus 0 = a \).

v) \( a \oplus a = 0 \).

vi) \( a \oplus 1 = \neg a \).

vii) \( a \oplus \neg a = 1 \).

viii) \( x \oplus a = x \oplus b \Rightarrow a = b \).

\[
a \oplus x = b \oplus x \Rightarrow a = b .
\]

ix) The following are equivalent:

\[
a \oplus b = c ,
\]

\[
a \oplus c = b ,
\]

\[
b \oplus c = a .
\]
We will say that \( r = r_1 r_2 ... r_n \) is a random sequence of bits if

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

The following property of xor is especially important for cryptography.

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

Note that \( a \) need not be random.

Note property (x) doesn’t hold for and (\( \land \)) or for or (\( \lor \)).

- Consider \( a \land r \), where \( r \) is random. \( a \land r \) is random in those bits in which \( a \) is 1, but is completely determined in those bits in which \( a \) is 0.
- Even if \( a \) is also random, \( a \land r \) 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

\[ \text{prob}(a = 1) = p \quad \text{and} \quad \text{prob}(b = 1) = q. \]

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

\[
0.5 - (p + q - 2pq) = 2|pq - p - q + 0.25| + 2|p - 0.5| \cdot |q - 0.5|
\]

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:

<table>
<thead>
<tr>
<th>prob(a=1)</th>
<th>prob(b=1)</th>
<th>prob(a\oplus b=1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.6</td>
<td>0.4</td>
<td>0.52</td>
</tr>
<tr>
<td>0.6</td>
<td>0.7</td>
<td>0.46</td>
</tr>
<tr>
<td>0.52</td>
<td>0.52</td>
<td>0.4992</td>
</tr>
<tr>
<td>0.52</td>
<td>0.498</td>
<td>0.50008</td>
</tr>
</tbody>
</table>

If we have two independent bit sequences \( a \) and \( b \) (independent of each other), each somewhat close to random, then \( a \oplus b \) 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 $p$ (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.