CS3381-Cryptography

Lecture 4: Bits and bytes; One-time pad revisited; Stream ciphers

September 12, 2014

1 Binary encoding

Computers, of course, do not use a 26-letter alphabet to represent the data they manipulate. They employ the 2-letter alphabet $\{0, 1\}$ of *bits,* and these bits are typically grouped into 8-bit *bytes*. We now view our encrypted data as consisting of sequences of bits. Addition mod 26 is now replaced by addition mod 2, which we denote $\oplus$:

$$0 \oplus 0 = 1 \oplus 1 = 0, 1 \oplus 0 = 0 \oplus 1 = 1.$$

'Addition mod 2' reflects a number theory view of this operation. If you think of it in terms of boolean logic, $\oplus$ is the *exclusive-or* (XOR) operation.

When we have two bit sequences of the same length, the $\oplus$ operation is applied bit by bit, for example:

$$011011 \oplus 101101 = 110110.$$

This operation is commutative and associative, and also satisfies

$$v \oplus v = 00 \cdots 0,$$

for any bit string v.

It is also helpful to have notation for concatenating sequences of bits. It is somewhat standard in the subject to use $||$ for this, although I don't think our textbook uses this notation.

$$011011||1011 = 0110111011.$$

There are lots of ways to write sequences of bytes. We can view each byte as an integer in the range 0..255, for instance:

```
20, 169, 146, 156, 39, 131, 19, 176, 44, 85, 215
```

Often we are less interested in these numerical values than in the actual pattern of bits, so we can represent the same sequence directly in binary form, here grouped into to 8-bit blocks:

```
00010100 10101001 10010010 10011100 00100111 10000011
00010011 10110000 00101100 01010101 11010111
```

Such bit strings are very difficult to read, so it is more convenient to represent each byte as two hexadecimal digits. (Each of the hex digits $0, 1, \ldots, 9, a, b, \ldots f$ represents one of the 16 different sequences of 4 bits.) The sequence above is represented in hex as:

```
14 a9 92 9c 27 83 13 b0 2c 55 d7
```

It is useful to have a more compact text representation of bit strings. Every printable character has a one-byte ASCII code, but not every byte value represents a printable character, so a typical string of bytes cannot be represented naturally in text. Instead, we use the following scheme: The lower-and upper-case letters along with the digits give us 62 different characters. If we throw in two more (+ and /) we get $64 = 2^6$, so we can represent each 6-bit block by a single character. A problem arises because the number of bits in a sequence of bytes is typically not divisible by 6. In this case either two or four zero bits of padding are added before encoding to make the number of bits divisible by 6, and '=' or '==' is appended to the string, depending on how much padding was added. Here is the base 64 representation of our example—since it contains eleven bytes, there are 88 bits. Two zero bits were added to make the total divisible by 6, so a single '=' appears.

```
FKmSnCeDE7AsVdc=
```

2 The One-time Pad, revisited

Both the key and the plaintext are sequences of bytes of the same length. Encryption is no longer addition mod 26, but addition mod 2:

$$E(k, m) = k \oplus m.$$

Since addition and subtraction mod 2 are the same operation, decryption is *identical* to decryption:

$$D(k, c) = k \oplus c.$$

The one-time-only principle looks like this: If we encrypt two different plaintexts with the same key,

$$c_1 = k \oplus m_1, c_2 = k \oplus m_2,$$

then

$$c_1 \oplus c_2 = m_1 \oplus m_2,$$

since the term $k \oplus k$ is all zeros. If at some later date we learn the value of one of the plaintexts, say m_1, then we have

$$c_1 \oplus m_1 = k,$$

and we recover the second plaintext

$$c_2 \oplus k = m_2.$$

Even without complete recovery of m_1, the re-use of the key leaks a lot of information—we know exactly where the two plaintexts are the same, which could tip us off to the nature of the message.

3 Stream ciphers

The random-number generator built into Python (and nearly every other programming environment) provides a method for producing an arbitrarily long stream of seemingly random bits by 'seeding' the random number generator with a short key. The code below carries this out. The function `getrandbits` is a standard library function in the module `random` that generates a desired number of bits from the built-in random number generator. The result is given as a Python long integer. The module `bytestuff` contains functions for converting sequences of bytes between different representations and carrying out the xor operation.

```
import bytestuff
import random

def otp_encrypt(k,s):
    random.seed(k)
    v=random.getrandbits(8*len(s))
    u=bytestuff.long_to_bytes(v)
    return bytestuff.xor(u,s)
```

The general idea is that the random number generator is an algorithm that takes reasonable-length keys $k \in \mathcal{K}$ and produces 'random' strings of bits some large length L. That is, the algorithm computes a function

$$g : \mathcal{K} \to \{0,1\}^L.$$

Encryption and decryption are then done just as with the one-time pad. If m is a string of L bits then

$$E(k,m) = D(k,m) = g(k) \oplus m.$$

Such a setup is called a *stream cipher,* and the function g is called a *pseudo-random number generator.* Here k of course is the key, and the long string $g(k)$ is called the *keystream*.

We have proved that such a system cannot have perfect secrecy, because we are using relatively short keys to encrypt long messages. But it is possible that if the key k is long enough to resist brute-force searching, such a scheme will be computationally secure. (However, see below concerning the security of this particular example!)

In a typical stream cipher, a large *state vector* is maintained, There is an algorithm that updates the state and emits one bit of output. Prior to encryption, the secret key k shared by Alice and Bob, is concatenated to another sequence of bits IV called the *initalization vector,* and the result $IV||k$ is used to initialize the state. Let us suppose the plaintext m is N bits long. The state is then updated N times, producing the keystream s of N bits. Alice computes $c = s \oplus m$ and sends Bob $IV||c$. Note that the IV is sent in the clear: it is used to help randomize the initial state so that the same keystream does not appear twice.

Bob on his end takes the IV, concatenates it to the secret key k, and initializes the state. He then updates the state N times, which provides him with a copy of the keystream s. He now computes

$$s \oplus c = s \oplus (s \oplus m) = (s \oplus s) \oplus m = 0 \oplus m = m$$

to recover the plaintext. In short, this is a mechanism for simulating a one-time pad using a short key k.

In our example above, initializing the state is handled by the function `random.seed`, and updating the state and obtaining keystream bits by `random.getrandbits()`. No initialization vector appears in the example, but it is easy enough to modify it so that the value used to reseed the generator is split into two components.

The resulting stream cipher, however, is highly insecure, and in fact you should **never** design a stream cipher using a system random number generator! Such random number generators are designed to produce output that is random-looking to various statistical tests of randomness, and therefore useful for simulation and modeling. But the problem is that such generators are also predictable: If you have a modest number of bits of the keystream, it is possible to recover the internal state of the generator, and predict all future bits of the keystream. (In the Python random number generator, the state is about 80 bytes long. As it turns out, if you know about 80 bytes of plaintext, you can xor with the ciphertext to obtain 80 bytes of keystream. It is possible to use this to determine the initial state of the generator and thus all subsequent bits of the keystream.) Much effort has gone into the design of *cryptographically secure random number generators* that cannot be predicted in this way.

Stream ciphers are typically very fast, so they have been incorporated in systems like wireless communication requiring rapid on-the-fly encryption and decryption. The design of a good stream cipher is a challenging task. Recent standards using stream ciphers for DVD copy-protection (Content Scrambling System), cellphone encryption (ORYX and A5), and wireless network security (WEP encryption) have all been shown to be insecure. There are ongoing efforts to produce secure stream ciphers. (See Project 2.)

4 LFSR Stream Ciphers

LFSR stands for 'Linear Feedback Shift Register'. This is a scheme for creating a shift cipher that can be implemented very rapidly in hardware. As we'll see, stream ciphers based on LFSRs are very insecure, but they are a core element of more secure designs.

The state consists of a sequence of m bits

$$s_{m-1}s_{m-2}\cdots s_0.$$

Each time the state is updated, a new bit is computed by XORing several of the bits of the present state. This new bit s is the output bit—the next bit of the keystream. The new state is

$$ss_{m-1}\cdots s_2s_1.$$

That is, all the bits of the original state are shifted right, and the new bit s is shifted in from the left. (It is probably more common in discussions of LFSR to treat s_0,

rather than the new bit s, as the next output of the register. Bit s will then appear as the $(m+1)^{th}$ output, after it has been shifted all the way to the right. By treating things in this slightly different fashion, we are just skipping the first m bits of the output.)

Here is an example with 5 bits. The updating function is given by

$$s = s_2 \oplus s_0.$$

If we start in the state 01100, the subsequent states are

```
10110
11011
11101
01110
10111
01011
10101
01010
00101
00010
00001
10000
01000
00100
10010
01001
10100
11010
01101
00110
10011
11001
11100
11110
11111
01111
00111
00011
10001
```

```
11000
01100
```

If you inspect this list carefully, you'll see that all 31 nonzero 5-bit patterns appear in it, each one exactly once. The sequence of leftmost bits is the keystream generated by this LFSR.

In an LFSR, we always require that at least one of the 'tapped' bits (the bits that are being XORed together) is the rightmost bit s_0. This being the case, if a state has at least one nonzero bit, the next state cannot consist entirely of zeros (why?). Thus as long as we start in a nonzero state, every subsequent state will be nonzero. If we have m bits in the register, there are thus a total of $2^m - 1$ possible states. This means that after at most $2^m - 1$ updates, the states will repeat, so the LFSR keystream is periodic. It is desirable to have the period as long as possible, so as to have the least amount of repetition in the keystream. (There is a good deal of mathematics behind how to choose the tapped bits so as to obtain the maximum period.) In our example above, we get the maximum period $2^5 - 1 = 31$.

For such a generator, if we know m successive bits of the keystream, we can completely recover the original state. Essentially this involves solving a system of linear equations using addition mod 2, but the equations take on a particularly simple form. Let's do this with our 5-bit example. Suppose the state was $s_4 s_3 s_2 s_1 s_0$, and the next 5 bits of the keystream are $s_5, s_6, \ldots, s_9$. Then we can write

$$\begin{aligned} s_5 &= s_0 \oplus s_2 \\ s_6 &= s_1 \oplus s_3 \\ s_7 &= s_2 \oplus s_4 \\ s_8 &= s_3 \oplus s_5 \\ s_9 &= s_4 \oplus s_6. \end{aligned}$$

Now if we know the values of s_5 through s_9, then we can substitute these in the last two equations and find s_3 and s_4 immediately. We can substitute these new values in the second and third equations and find s_1 and s_2. Finally we can substitute s_2 in the first equation and find s_0. For example, if the next 5 bits of the keystream are 1,1,0,0,1 then we have

$$0 = s_3 \oplus 1, 1 = s_4 \oplus 1,$$

which gives $s_3 = 1, s_4 = 0$. Substitution into the second and third equations gives

$$1 = s_1 \oplus 1, 0 = s_2 \oplus 0,$$

so $s_1 = s_2 = 0$. The first equation is now

$$1 = s_0 \oplus 0,$$

so $s_0 = 1$. The entire state is thus recovered as 01001. You can see in the tabulation above that the first 5 output bits (leftmost bits) after 01001 are indeed 1,1,0,0,1.

What does this imply about using LFSRs as stream ciphers? Of course, 5 bits is not a reasonable size state, but suppose we built a generator with maximal period and 80 bits of state. Let us assume that an attacker Eve knows the recurrence relation used to generate the keystream from a state (Kerckhoffs' principle) and knows some bits of the keystream. (This can be determined if Eve knows some plaintext.) Here it is impossible to brute-force guess the state that leads to a sequence of keystream bits. But if Eve knows 80 consecutive keystream bits, then she can solve the underlying system of 80 equations quite easily and recover the state. So by itself, this is a very bad cipher.

On the other hand, a widely-used idea is to build stream ciphers by combining a few LFSRs in a nonlinear fashion, for instance through ANDs and ORs of several registers of different sizes.