CS3381-Cryptography

Lecture 3: Pseudorandom number generators and stream ciphers.

January 31, 2017

Stream ciphers are simulated one-time pads that try to get around the restriction that the key must be as long as the message. A short truly random key k is expanded to a long 'random-looking' string $G(k)$, and the message is encrypted by setting

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

1 Pseudorandom generators

A pseudorandom generator (PRG) is a function

$$G : \{0,1\}^n \to \{0,1\}^\ell,$$

where $\ell >> n$. We require that the function G be easy to compute: in practice, this means that there is a fast, efficient algorithm for extracting the ℓ bits of $G(k)$ from k. (We call k the 'seed' value used for the generator.)

We also require that if k is chosen uniformly at random from $\{0,1\}^n$, then $G(k)$ *looks* as if it had been chosen uniformly at random from the much larger set $\{0,1\}^\ell$. What 'looking random' means in this context is that the generator be *cryptographically secure.* We will give precise definitions later on. For now, we will give two (equivalent, as it turns out) rough formulations of this notion: The first is that there is no efficient, reliable method for distinguishing a random output of G from a string that is chosen uniformly at random from $\{0,1\}^\ell$. 'Reliable' means that the method succeeds with more than negligible probability. The second is that there is no efficient, reliable way, to predict bit $i+1$ of the output from bit i of the output.

Even with these imprecise definitions, we can still get a feel for what they mean. Let us suppose, to be concrete, that $m = 128$, $\ell = 256$. We can in principle

compute all 2^{128} values $G(k)$ for $k \in \{0,1\}^1 28$. Then if we are given a bit string $m \in \{0,1\}^{256}$, we would guess that m was chosen at random from the message space if m is not among these 2^{128} precomputed values, and that it was computed by G if and only if it was among these values. The probability of guessing wrong (that is, guessing that the string is the output of the generator when it was chosen randomly from the large message space) is no more than $\frac{2^{128}}{2^{256}} = 2^{-128}$, which is minuscule, so this method is more than reliable: it is almost certainly correct. However, it is highly inefficient, as it requires us to precompute an infeasible number of values of G, and to search through this list of values.

In particular, this means that in order for the PRG to be cryptographically secure, the seed length n cannot be *too* small, otherwise the above method becomes efficient.

2 Stream ciphers

The PRG gives you a cryptosystem by setting

$$\mathcal{K} = \{0,1\}^n, \mathcal{M} = \{0,1\}^\ell,$$

and

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

This is identical to the one-time pad with pseudorandom $G(k)$ replacing the truly random one-time key. In practice, Alice and Bob share the short key k, and use it to separately regenerate the 'keystream' $G(k)$ without having to share this very long string in advance.

The underlying PRG must be secure. If it is not, then a little bit of known plaintext gives us a little bit of known keystream: if we could use this to predict more of the key, we could decrypt more of the message.

The same caveats (malleability, reuse of key) that apply to the one-time pad apply here as well.

3 Examples

3.1 Repeating-byte XOR

Strictly speaking, the Vigenère-like system that was used to encrypt the MS Word documents on Assignment 1 is a stream cipher. If the message length is, say,

1024 characters, which is 64 16-byte blocks, then the underlying PRG maps $k \in \{0,1\}^{128}$ to

$$\underbrace{k||k||\cdots||k \in \{0,1\}^{8192}}_{\text{64 times}}.$$

Obviously, there is a quick and easy method for telling whether a string is a possible output of the generator: just verify that each 16-byte block matches the previous 16-byte block. So this not a cryptographically secure PRG.

3.2 Standard system random-number generators

The methods in `java.util.Random` in Java, or the Python module `random`, are designed to produce 'random-looking' output for purposes of simulation, but are not cryptographically secure. For example, the Java generator works by maintaining a 48-bit state s. To generate a new random value, the state is updated by the simple formula

$$s \longleftarrow (25214903917 \times s + 11) \bmod 2^{48}.$$

If the generator is used (as is often the case) to generate its output bits in 32-bit chunks, then we use the high-order 32 bits of the state, namely

$$\lfloor s/2^{16} \rfloor.$$

Random-number generators that update their state by formulas like the one above are called *linear congruential generators*. Let's see why this is insecure. Suppose we are given a string u of at least 64 bits, and want to know if it is the output of the generator, or a randomly-chosen string. We write $u = v||w$, where v and w are each 32 bits. The string v consists of the high-order 32 bits of a state s of the generator, and w is the high-order 32 bits of the next state. We can try to guess the low-order 16 bits t of $s = v||t$, and then compute the next state by the formula above and see if its high-order 32 bits matches w. It is not too much work to try out all $2^{16} = 65536$ possible values for t. There is only a remote probability (about 2^{-32}) that more than one value of t will yield a state whose high-order bits match w. So this is a reasonably efficient way to 'break' the random-number generator.

If we only used the high-order 8 bits of the state as the generator output, this attack becomes less feasible, since we would have to try out all 2^{40} possible values of t. But there are ways to predict the output of linear congruential generators that do not require so much work. The generator in Python's `random` module works differently, but is likewise insecure.

Moral: Do not build stream ciphers with random-number generators that were not specifically designed for cryptographic use.

4 RC4 stream cipher

This stream cipher was in wide use for many years. Subsequent security problems were discovered, which has led to its being abandoned. It is actually difficult to exploit these problems if the cipher is used properly, but improper use (see below) has led to problems.

The underlying generator is described in detail code posted on the course website. The seed value k can be between 40 and 128 bits. This is used to generate a 256-byte state that is some permutation of the 256 possible byte values values $0, \ldots, 255$. Thus there are $256!$ possible states. To generate an n-byte keystream, the state is updated n times through a series of swaps, and a new byte is emitted after each swap. We get a pseudorandom string $G(k)$ in $\{0,1\}^{8n}$ from the short seed value.

4.1 WEP 802.11 wireless encryption

WEP stands for 'Wired Equivalent Privacy', but as we'll see, it is anything but!

In the original version, the wireless router and the user share a long-term key which was 40 bits long, to comply with US government export restrictions on cryptography. 40 bits is already troublesome, and vulnerable to brute-force attacks. Later versions used a long-term key of 104 bits.

WEP uses the RC4 stream cipher. Since we cannot reuse a key with a stream cipher, how do we make a long-term shared key work? The idea is to combine the long-term key with a one-time string, called an *initialization vector*, and denoted IV. The IV together with the key k is used to generate a fresh seed value

$$s = g(IV, k)$$

for the RC4 generator. The IV is sent in the clear, concatenated to the ciphertext. That is, if Alice wants to send the plaintext m, she generates a random IV, then computes $s = g(IV, k)$, and sends Bob

$$IV||(m \oplus G(s)).$$

Bob on his end strips IV from the received message, computes $s = g(IV, k)$ using the shared key k, and regenerates the keystream $G(s)$, then XORs to recover m.

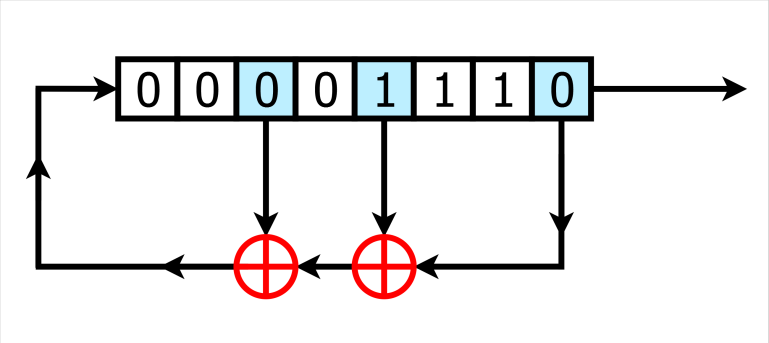

Figure 1: LFSR (Image copied from Wikipedia)

WEP used only 24 bits for the IV, which means that there are at most $2^{24} \approx 16$ million different values of s. Thus in a large volume of traffic, the same key will be reused several times. Furthermore, the way in which the key and IV were combined to produce the seed was simply concatenation:

$$s = g(IV, k) = IV||k,$$

The IV was incremented by 1 with each successive packet, rather than regenerated at random. Known weaknesses in RC4 in combination with these practices resulted in an attack that could recover the key k after interception of several million packets. Attacks only get stronger: The cryptanalysis was eventually strengthened to the point where k could be recovered after interception of tens of thousands of packets.

4.2 LFSR and Content Scramble System

There is a simple method for generating a long apparently random-looking stream of bits from a short seed, called a *linear feedback shift register*, or LFSR. These are very fast when implemented in hardware. The idea is that several bits of an n-bit register are 'tapped'. At each clock cycle, the XOR of the tapped bits is shifted into the high-order bit of the register, and the bit shifted out of the low-order bit is the output bit. The initial seed value must be chosen different from 0^n. (See Figure 1.)

Though the resulting stream may look random, this is obviously insecure, since once we have n successive bits of output, the remaining bits are completely

determined. However, by combining the output of several different stream ciphers in a nonlinear way, it is in principle possible to build a reasonably secure PRG, and use it in a stream cipher. This is the idea behind the Content Scramble System (CSS) used beginning in the 1990's to encrypt video on commercial DVDs. The implementation of the idea was very weak, and the system was easily cracked. (The details of CSS were a trade secret, but they were quickly discovered and released by hackers, and a complete break of the system was distributed soon after.)

CSS uses a 40-bit key and two LFSRs, one of 17 bits and the other of 25 bits. Initially, 16 bits of the key are loaded into the 17-bit LFSR, along with an additional 1 bit; the remaining 24 bits of the key are loaded into the 25-bit register, along with an additional 1 bit.

In each step of the stream generation, 8 bits of output are produced from each LFSR. The 8 low-order bits of the sum form the next byte of the keystream, and any carry is added into the next step.

Let's see why this is insecure: Sectors of the encrypted video contain a header in a known format, so it is possible for the attacker to recover the first few bytes of the keystream. The attacker can then guess the 16 bits of the key used to seed the first LFSR, and produce the first few bytes of output. As a result, the attacker then obtains, by subtraction from the known keystream, the first few bytes of output of the second LFSR, and uses this conjectured values from the two registers to generate a few bytes more of the keystream. If they match the known keystream bytes, then the guess was correct. You only need to guess 2^{16} different initial values for the first LFSR.

It should be added that this is an oversimplified description. The 40-bit key for the stream cipher is itself encrypted, and must be recovered by means of another secret key, which was distributed to manufacturers of DVD players. However, this phase of the encryption was also cracked.

4.3 Good stream ciphers?

So as not to leave you with the impression that there are no secure stream ciphers, there are really some good alternatives. In the next unit we will look at 'block ciphers in CTR mode', which provides one way to build a secure stream cipher.

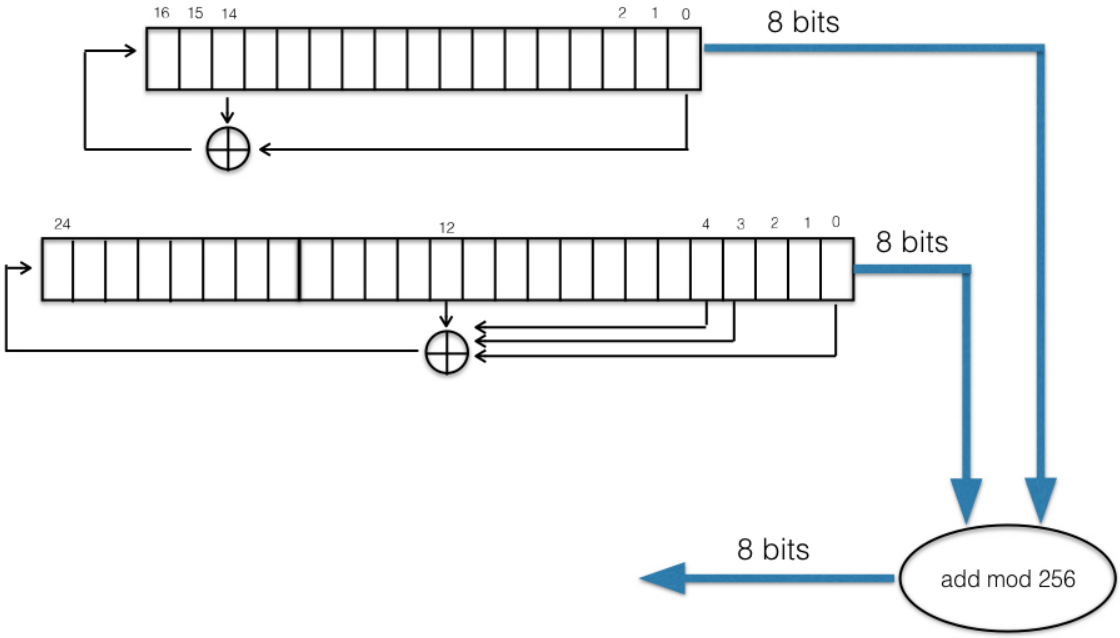

Figure 2: The keystream generator in CSS. At each step, the two LFSR are clocked 8 times, and the two 8-bit outputs are added to produce the next output byte of the keystream. The attack, requiring just a few bytes of known plaintext, guesses the 17-bit LFSR, infers the initial state of the 25-bit LFSR, and checks of the result agrees with the known plaintext. This requires checking just 2^{16} guesses.