2. Stream Ciphers

Stream ciphers constitute very practical encryption schemes that can be easily be implemented in hardware. They are widely used in prominent protocols such as SSL/TLS and 802.11b WEP.

2.1 Definition of Stream Ciphers

We have seen that the One-time Pad provides very strong security guarantees, but is not usable for most practical applications as keys need to be as long as the messages. One idea to overcome this problem is to not use keys that are fully random but keys that only look random. A relatively short string which is truly random is used to compute a larger string which, while of course not being truly random, is as good as being random in a sense formally defined below. This large string is called a *pseudorandom* string, and it can be used to replace the random key in the One-time Pad.

Such a cipher is called a synchronous stream cipher, where synchronous means that the pseudorandom string is created independently of the messages. Recall that we proved in the last chapter that such a cipher cannot provide perfect secrecy since its key space is much smaller than the message space; however, we will see that it still provides strong security guarantees.

Definition 2.1 (Synchronous Stream Cipher) $(\mathsf{E}, \mathsf{D}, \mathsf{G})$ *is a* synchronous stream cipher *over* $(\mathcal{M}, \mathcal{C}, \mathcal{K}, \mathcal{L})$ *iff* (E, D) *is a symmetric encryption scheme over* $(\mathcal{M}, \mathcal{C}, \mathcal{L})$*, and* $\mathsf{G} : \mathcal{K} \to \mathcal{L}^*$ *is an efficient function called the* pseudorandom generator *of the stream cipher.* $\diamond$

Elements of $\mathcal{K}$ are called *seeds*, $\mathcal{L}$ is called the *key-stream alphabet*, and output of G *pseudorandom strings*. Of course, the security of a stream cipher now crucially depends on the function G which creates long, almost random strings out of short, random strings. The central property this function should satisfy is called *unpredictability*.

2.2 Unpredictability of Pseudorandom Generators (PRGs)

Why do we need unpredictability? Assume an email is transmitted in an encrypted manner. Then one knows that large parts of the plaintext, e.g., the header, have a fixed format. Thus an attacker can obtain some bytes of the pseudorandom string at the beginning by computing the xor of the initial bits of the encryption and the known parts of the header. Perhaps he needs to guess where exactly the header is placed, but this is possible in reasonable time. If he was able to predict the behavior of the pseudorandom generator, then he would be able to decrypt the remaining parts of the message.

Slightly more formally, this means that if an attacker sees an initial prefix of the pseudorandom string generated by G, then he is not able to predict the next bit with probability better than $\frac{1}{2}$, i.e., the probability of purely guessing the bit, apart from a very small, so-called *negligible* advantage

ϵ. Before we give the rigorous definition of (next-bit) unpredictability, we first rigorously define the notions of efficient algorithms and negligible functions. Both constitute central concepts in modern cryptography.

Definition 2.2 (Efficient Algorithm) *Let $k \in \mathbb{N}$ be the security parameter. An algorithm A is* efficient *or* efficiently computable *iff it is computable in probabilistic polynomial-time (in k). We write $\mathsf{A} \in PPT$ if A is efficiently computable.* $\diamond$

A function is negligible if it decreases faster than the inverse of any polynomial. Formally we define it as follows:

Definition 2.3 (Negligible Function) *A function $f : \mathbb{N} \to \mathbb{R}_0^+$ is* negligible *iff for all $c \in \mathbb{N}$ there exists an $n_c \in \mathbb{N}$ such that for all $n \geq n_c$ we have*

$$f(n) \leq \frac{1}{n^c}.$$

$\diamond$

We are now ready to give the definition of next-bit unpredictability. For the sake of readability, we only address the case that the keystream alphabet is the set of all strings of a certain length k; similarly, we expand these strings to strings of length $p(k)$ where p is an arbitrary but fixed polynomial.

Definition 2.4 (Next-bit Unpredictability) *A deterministic efficient function $\mathsf{G} : \{0,1\}^k \to \{0,1\}^{p(k)}$ for a polynomial p is* (next-bit) unpredictable *iff for every $0 \leq i \leq p(k)$ and every attacker $\mathsf{A} \in PPT$, there exists a negligible function ϵ such that*

$$\begin{array}{lll} P[& b = b_i; & \# \textit{ The probability that } \mathsf{A}\textit{'s guess is correct} \\ & x \leftarrow_{\mathcal{R}} \{0,1\}^k, & \# \textit{ if the seed } x \textit{ is drawn uniformly at random} \\ & b_1 \ldots b_{p(k)} \leftarrow \mathsf{G}(x), & \# \textit{ if } \mathsf{G} \textit{ generated } p(k) \textit{ bits using the seed } x \\ & b \leftarrow \mathsf{A}(b_1 \ldots b_{i-1}) \,] & \# \textit{ and if } \mathsf{A} \textit{ guesses } b_i \textit{ knowing bits } b_1, \ldots, b_{i-1} \\ & \leq \frac{1}{2} + \epsilon(k). & \# \textit{ is only negligibly larger than pure guessing.} \end{array}$$

$\diamond$

2.3 Getting True Randomness in Practice

The seed for a pseudorandom generator, and keys for symmetric encryption schemes in general, should be as random as possible. One uses for example physical random number generators to get good randomness.

There are some physical sources that are supposed to produce good randomness, but the resulting bits may have a certain bias or some correlation. One usually circumvents this by taking the xor of bits obtained from different such sources, and by subsequently applying a randomness extractor further improving the quality of the output. An example for such a randomness extractor is the Von Neumann Correction.

Typical physical sources of randomness include:

- Thermal noise in various electric circuits, e.g., Zener-Diodes, optical sensors,

- atmospheric noise, captured by antennas,
- radioactive radiation, and
- more playfully, lava lamps. There is even a patent for creating random bits with lava lamps (US patent #5,732,138).

There is also a number of random sources that are more easily available, at the cost of producing weaker randomness. What is used also in practice is

- measurement of times between user key-strokes, and
- time needed to access different sectors on the hard-disk drive (the air turbulences caused by the spinning disk is supposed to be random).

2.4 Attacks Against the One-time Pad and Stream Ciphers, and How to Defend Against Them

Recall that the One-time Pad has perfect secrecy and thus is not vulnerable to a ciphertext-only attack. However, it turns out that the One-time Pad as well as stream ciphers are vulnerable to stronger attacks in which the attacker does more than passively easvesdropping on the wire; moreover, they are vulnerable against improper use of the scheme.

2.4.1 The Two-time Pad

One of the most important improper usages of the One-time Pad and stream ciphers is to re-use the random tape, i.e., to encrypt several messages with the same key. This may be called the "Two-time Pad", which is fully insecure. Assume that the adversary is given two encryptions $c_i = K \oplus m_i$ for $i \in \{1, 2\}$ for two messages m_1, m_2 with the same key K. Then the adversary simply computes the xor of c_1 and c_2 yielding:

$$c_1 \oplus c_2 = (m_1 \oplus K) \oplus (m_2 \oplus K) = m_1 \oplus m_2$$

Thus re-using the key leaks the XOR of the actual plaintexts. Assuming that both messages contain ordinary text or have some other form of exploitable redundancy, we can again use frequency analysis to recover the plaintexts m_1 and m_2 from $m_1 \oplus m_2$. Thus one has to be careful not to re-use a key when using the One-time Pad or stream ciphers. There are mainly two methods to realize this:

- One might use successive parts of the output stream to encrypt successive messages. This requires synchronization of the senders and the receivers streams by some means, usually by transmitting its position along with the encrypted message. This has disadvantages if the order of messages is changed in the transmission line or by the protocol.
- One might create a new seed for each message or file that needs to be encrypted. Then one additionally transmits the seed along with the message. Of course, the seed has to be transmitted secretly somehow. This can be done by combining the stream cipher with another cipher E^* to transmit the seed x as follows:

$$\mathsf{E}^*(x) || \mathsf{G}(x) \oplus m.$$

This has the advantage that a fast stream cipher can be used to encrypt the potentially very long message m, while a slower encryption scheme E^* is only needed to encrypt the shorter seed.

2.4.2 Active Attacks against the One-time Pad and Stream Ciphers

If an adversary not only reads messages but also is able to change messages, then attacks such as the following become possible. Consider a simple voting system, where each participant votes 0 or 1. This vote is encrypted using the OTP, i.e., each participant i shares a random bit $K_i \leftarrow_{\mathcal{R}} \{0,1\}$ with a central voting authority. Every participant sends the ciphertext $c_i \leftarrow K_i \oplus m_i$ to the central authority. The voting authority then computes $c_i \oplus K_i = m_i$ and counts all votes. Note that this is a very weak form of voting scheme since the voting authority can see how everybody voted, and it is provided for the sake of demonstration only.

Now assume that the attacker knows (e.g., by means of an opinion poll) that a majority of all participants are going to vote 0. Then by intercepting a vote c_i and replacing it with $c_i^* := c_i \oplus 1$ the attacker is able to invert the outcome of the vote, as these ciphertexts decrypt to

$$c_i^* \oplus K_i = (m_i \oplus K_i \oplus 1) \oplus K_i = m_i \oplus 1.$$

This attack applies to both the One-time Pad and stream ciphers. One says that the One-time Pad and stream ciphers are highly *malleable*, i.e., they offer no form of integrity of the plaintexts. Thus both ciphers should only be used with suitable integrity protection mechanisms. These are called MACs (message authentication codes) and will be discussed later in class.

2.5 Examples of Pseudorandom Generators

2.5.1 RC4

RC4 is a stream-cipher invented by Ron Rivest in 1987 for RSA Security, which also holds the trademark for it. The source code was originally not released to the public because it was a trade secret, but was posted to a newsgroup some time ago; thus people referred to this version as *alleged RC4*. Today it is known that alleged RC4 indeed equals RC4. While RC4 does not satisfy next-bit unpredictability, it is considered secure from a practical point of view if one takes certain precautions. It is used in many protocols such as SSL/TLS and 802.11b WEP.

Its main data structure is an array S of 256 bytes. The array is initializes to the identity before any output is generated, i.e., the first cell is initialized with 0, the second with 1 and so on. Then the cells are permuted using certain operations that depend on the current state and the chosen key K. In pseudocode, the RC4 initialization phase works as follows:

```
for i from 0 to 255
     S[i] := i
j := 0
for i from 0 to 255
     j := (j + S[i] + K[i mod keylength]) mod 256
     swap(S[i],S[j])
```

After initialization has been completed, the following procedure computes the pseudorandom sequence. For each output byte, new values of the variables i, j are calculated, the corresponding

cells are swapped, and the content of a third cell is output. This described again in pseudocode as follows:

```
i := 0
j := 0
while GeneratingOutput:
     i := (i + 1) mod 256
     j := (j + S[i]) mod 256
     swap(S[i],S[j])
     output S[(S[i] + S[j]) mod 256]
```

One easily sees that the first byte which is output depends on the content of 3 cells only. This property can be used to launch attacks against the cipher, so one usually discards the first 256 bytes of output generated by this algorithm to prevent these attacks.

2.5.2 Linear Feedback Shift Register (LFSR)

Linear Feedback Shift Registers constitute a very fast way of generating pseudorandom strings. An LFSR consists of a shift register that shifts its content for one bit at each clock, and a function that determines the feedback, i.e., the next value that enters the shift register, as a function of its current state. For LFSRs this is a linear function, i.e., the xor of certain specified bits in the state. The initial state of the shift register comprises the seed. An schematic sketch of an LFSR is given in Figure 2.1. The security of an LFSR is determined by its feedback function and how it is interlinked

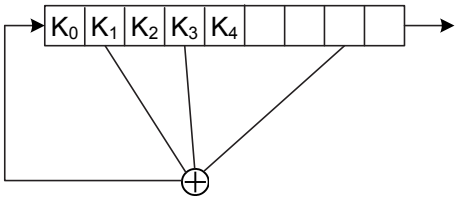

Figure 2.1: Linear Feedback Shift Register

with the surrounding operations. There also exist constructions with non-linear feedback. Note that LFSRs on their own cannot naively be used as stream ciphers because the first bits that are output are precisely the key bits which would make any stream cipher insecure. The solution taken here is that one throws away the first bits and subsequently combines LFSRs in some clever way either with other LFSRs or with other cryptographic primitives. For instance, one encrypts the output of an LFSR using a secure block cipher (see the next lecture), which yields a pseudorandom sequence provided as long as the bits output by an LFSR do not repeat themselves.

2.5.3 The Content Scrambling System (CSS)

We conclude with a brief description of the Content Scrambling System (CSS), which is a proprietary standard for encrypting the contents of DVDs. The key management is rather complex and we do not go into details here. The actual content is encrypted with a stream cipher whose corresponding

PRG, i.e., the generation of the pseudorandom sequence, will be described in the sequel. Figure 2.2 illustrate the PRG.

Each sector key consists of 5 bytes $K_0, \ldots, K_4$. The pseudo random generator comprises two LFSRs that operate as follows. The first LFSR has a length of 17 bits and is initialized with $1||K_0||K_1$, where the feedback is computed as the xor of bits 1 and 15. The second LFSR has a length of 25 bits initialized as $1||K_2||K_3||K_4$, where the feedback is calculated as the xor of bits 11, 19, 20, and 23. Both LFSRs are clocked eight times, each of them producing eight bits of output. This output is treated as bytes, added modulo 256 observing the carry bit from the previous addition. The resulting output is used to encrypt the actual data.

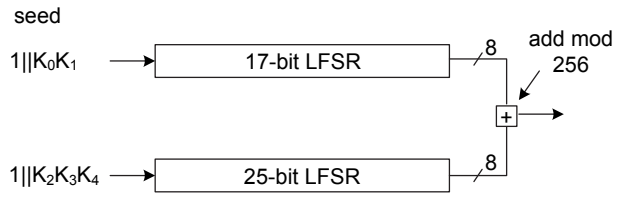

Figure 2.2: Content Scrambling System

The most apparent weakness of CSS is its short key size of 40 bits. This is a consequence of US export regulations that when when CSS was standardized. Brute-force attacks are reported to take 17 hours on a Celeron 366, i.e., on a very old hardware, and thus already much faster on more recent hardware. More sophisticated attacks exist that reduce the complexity of attacks to even only 2^{16}, thus taking only several seconds until CSS is successfully attacked.