Chapter 2
Stream Ciphers

If we look at the types of cryptographic algorithms that exist in a little bit more detail, we see that the symmetric ciphers can be divided into stream ciphers and block ciphers, as shown in Fig. 2.1.

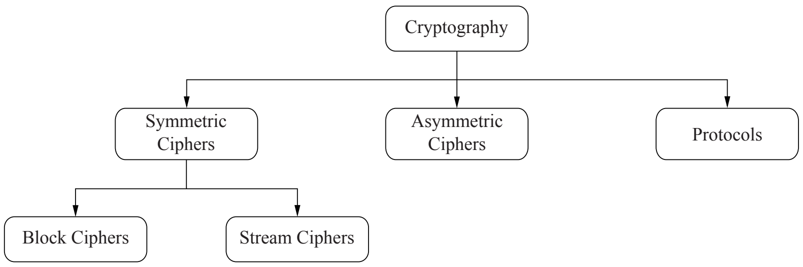

Fig. 2.1 Main areas within cryptography

This chapter gives an introduction to stream ciphers:

- The pros and cons of stream ciphers
- Random and pseudorandom number generators
- A truly unbreakable cipher: the One-Time Pad (OTP)
- Linear feedback shift registers and Trivium, a modern stream cipher

DOI 10.1007/978-3-642-04101-3_2, © Springer-Verlag Berlin Heidelberg 2010

2.1 Introduction

2.1.1 Stream Ciphers vs. Block Ciphers

Symmetric cryptography is split into block ciphers and stream ciphers, which are easy to distinguish. Figure 2.2 depicts the operational differences between stream (Fig. 2.2a) and block (Fig. 2.2b) ciphers when we want to encrypt b bits at a time, where b is the width of the block cipher.

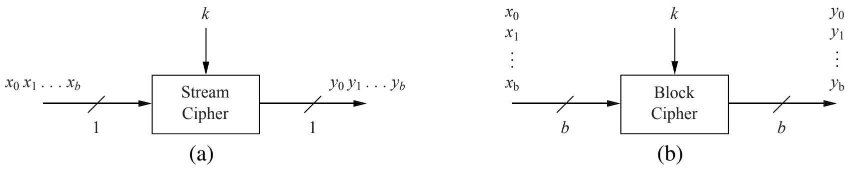

Fig. 2.2 Principles of encrypting b bits with a stream (a) and a block (b) cipher

A description of the principles of the two types of symmetric ciphers follows.

Stream ciphers encrypt bits individually. This is achieved by adding a bit from a *key stream* to a plaintext bit. There are synchronous stream ciphers where the key stream depends only on the key, and asynchronous ones where the key stream also depends on the ciphertext. If the dotted line in Fig. 2.3 is present, the stream cipher is an asynchronous one. Most practical stream ciphers are synchronous ones and Sect. 2.3 of this chapter will deal with them. An example of an asynchronous stream cipher is the cipher feedback (CFB) mode introduced in Sect. 5.1.4.

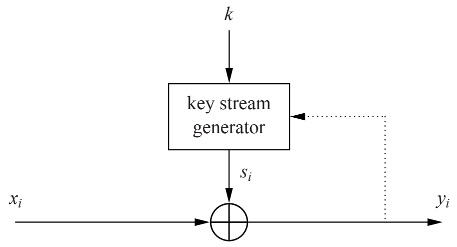

Fig. 2.3 Synchronous and asynchronous stream ciphers

Block ciphers encrypt an entire block of plaintext bits at a time with the same key. This means that the encryption of any plaintext bit in a given block depends on every other plaintext bit in the same block. In practice, the vast majority of block ciphers either have a block length of 128 bits (16 bytes) such as the advanced encryption standard (AES), or a block length of 64 bits (8 bytes) such as the data encryption standard (DES) or triple DES (3DES) algorithm. All of these ciphers are introduced in later chapters.

This chapter gives an introduction to stream ciphers. Before we go into more detail, it will be helpful to learn some useful facts about stream ciphers vs. block ciphers:

1. In practice, in particular for encrypting computer communication on the Internet, block ciphers are used more often than stream ciphers.
2. Because stream ciphers tend to be small and fast, they are particularly relevant for applications with little computational resources, e.g., for cell phones or other small embedded devices. A prominent example for a stream cipher is the A5/1 cipher, which is part of the GSM mobile phone standard and is used for voice encryption. However, stream ciphers are sometimes also used for encrypting Internet traffic, especially the stream cipher RC4.
3. Traditionally, it was assumed that stream ciphers tended to encrypt more efficiently than block ciphers. *Efficient* for software-optimized stream ciphers means that they need fewer processor instructions (or processor cycles) to encrypt one bit of plaintext. For hardware-optimized stream ciphers, *efficient* means they need fewer gates (or smaller chip area) than a block cipher for encrypting at the same data rate. However, modern block ciphers such as AES are also very efficient in software. Moreover, for hardware, there are also highly efficient block ciphers, such as PRESENT, which are as efficient as very compact stream ciphers.

2.1.2 Encryption and Decryption with Stream Ciphers

As mentioned above, stream ciphers encrypt plaintext bits individually. The question now is: How does encryption of an individual bit work? The answer is surprisingly simple: Each bit x_i is encrypted by adding a secret key stream bit s_i modulo 2.

Definition 2.1.1 Stream Cipher Encryption and Decryption
The plaintext, the ciphertext and the key stream consist of individual bits,
i.e., $x_i, y_i, s_i \in \{0,1\}$.
Encryption: $y_i = e_{s_i}(x_i) \equiv x_i + s_i \bmod 2$.
Decryption: $x_i = d_{s_i}(y_i) \equiv y_i + s_i \bmod 2$.

Since encryption and decryption functions are both simple additions modulo 2, we can depict the basic operation of a stream cipher as shown in Fig. 2.4. Note that we use a circle with an addition sign as the symbol for modulo 2 addition.

Just looking at the formulae, there are three points about the stream cipher encryption and decryption function which we should clarify:

1. Encryption and decryption are the same functions!

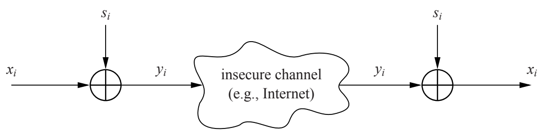

Fig. 2.4 Encryption and decryption with stream ciphers

2. Why can we use a simple modulo 2 addition as encryption?
3. What is the nature of the key stream bits s_i?

The following discussion of these three items will give us already an understanding of some important stream cipher properties.

Why Are Encryption and Decryption the Same Function?

The reason for the similarity of the encryption and decryption function can easily be shown. We must prove that the decryption function actually produces the plaintext bit x_i again. We know that ciphertext bit y_i was computed using the encryption function $y_i \equiv x_i + s_i \bmod 2$. We insert this encryption expression in the decryption function:

$$\begin{aligned} d_{s_i}(y_i) &\equiv y_i + s_i \bmod 2 \\ &\equiv (x_i + s_i) + s_i \bmod 2 \\ &\equiv x_i + s_i + s_i \bmod 2 \\ &\equiv x_i + 2s_i \bmod 2 \\ &\equiv x_i + 0 \bmod 2 \\ &\equiv x_i \bmod 2 \quad Q.E.D. \end{aligned}$$

The trick here is that the expression $(2s_i \bmod 2)$ has always the value zero since $2 \equiv 0 \bmod 2$. Another way of understanding this is as follows: If s_i has either the value 0, in which case $2s_i = 2 \cdot 0 \equiv 0 \bmod 2$. If $s_i = 1$, we have $2s_i = 2 \cdot 1 = 2 \equiv 0 \bmod 2$.

Why Is Modulo 2 Addition a Good Encryption Function?

A mathematical explanation for this is given in the context of the One-Time Pad in Sect. 2.2.2. However, it is worth having a closer look at addition modulo 2. If we do arithmetic modulo 2, the only possible values are 0 and 1 (because if you divide by 2, the only possible remainders are 0 and 1). Thus, we can treat arithmetic modulo 2 as Boolean functions such as AND gates, OR gates, NAND gates, etc. Let's look at the truth table for modulo 2 addition:

This should look familiar to most readers: It is the truth table of the *exclusive-OR*, also called *XOR*, gate. This is in important fact: **Modulo 2 addition is equivalent to**

x_i	s_i	$y_i \equiv x_i + s_i \bmod 2$
0	0	0
0	1	1
1	0	1
1	1	0

the XOR operation. The XOR operation plays a major role in modern cryptography and will be used many times in the remainder of this book.

The question now is, why is the XOR operation so useful, as opposed to, for instance, the AND operation? Let's assume we want to encrypt the plaintext bit $x_i = 0$. If we look at the truth table we find that we are on either the 1st or 2nd line of the truth table:

Table 2.1 Truth table of the XOR operation

x_i	s_i	y_i
0	**0**	**0**
0	**1**	**1**
1	0	1
1	1	0

Depending on the key bit, the ciphertext y_i is either a zero ($s_i = 0$) or one ($s_i = 1$). If the key bit s_i behaves perfectly randomly, i.e., it is unpredictable and has exactly a 50% chance to have the value 0 or 1, then both possible ciphertexts also occur with a 50% likelihood. Likewise, if we encrypt the plaintext bit $x_i = 1$, we are on line 3 or 4 of the truth table. Again, depending on the value of the key stream bit s_i, there is a 50% chance that the ciphertext is either a 1 or a 0.

We just observed that the XOR function is perfectly balanced, i.e., by observing an output value, there is exactly a 50% chance for any value of the input bits. This distinguishes the XOR gate from other Boolean functions such as the OR, AND or NAND gate. Moreover, AND and NAND gates are not invertible. Let's look at a very simple example for stream cipher encryption.

Example 2.1. Alice wants to encrypt the letter `A`, where the letter is given in ASCII code. The ASCII value for `A` is $65_{10} = 1000001_2$. Let's furthermore assume that the first key stream bits are $(s_0, \ldots, s_6) = 0101100$.

Alice	**Oscar**	**Bob**
$x_0,\ldots,x_6$ = 1000001 = `A`		
$\oplus$		
$s_0,\ldots,s_6$ = 0101100		
$y_0,\ldots,y_6$ = 1101101 = `m`		
	$\xrightarrow{\texttt{m}=1101101}$	
		$y_0,\ldots,y_6$ = 1101101
		$\oplus$
		$s_0,\ldots,s_6$ = 0101100
		$x_0,\ldots,x_6$ = 1000001 = `A`

Note that the encryption by Alice turns the uppercase `A` into the lower case letter `m`. Oscar, the attacker who eavesdrops on the channel, only sees the ciphertext letter `m`. Decryption by Bob *with the same key stream* reproduces the plaintext `A` again.
$\diamond$

So far, stream ciphers look unbelievably easy: One simply takes the plaintext, performs an XOR operation with the key and obtains the ciphertext. On the receiving side, Bob does the same. The "only" thing left to discuss is the last question from above.

What Exactly Is the Nature of the Key Stream?

It turns out that the generation of the values s_i, which are called the *key stream*, is the central issue for the security of stream ciphers. In fact, the security of a stream cipher *completely depends on the key stream*. The key stream bits s_i are *not* the key bits themselves. So, how do we get the key stream? Generating the key stream is pretty much what stream ciphers are about. This is a major topic and is discussed later in this chapter. However, we can already guess that a central requirement for the key stream bits should be that they appear like a random sequence to an attacker. Otherwise, an attacker Oscar could guess the bits and do the decryption by himself. Hence, we first need to learn more about random numbers.

Historical Remark Stream ciphers were invented in 1917 by Gilbert Vernam, even though they were not called stream ciphers back at that time. He built an electromechanical machine which automatically encrypted teletypewriter communication. The plaintext was fed into the machine as one paper tape, and the key stream as a second tape. This was the first time that encryption and transmission was automated in one machine. Vernam studied electrical engineering at Worcester Polytechnic Institute (WPI) in Massachusetts where, by coincidence, one of the authors of this book was a professor in the 1990s. Stream ciphers are sometimes referred to as Vernam ciphers. Occasionally, one-time pads are also called Vernam ciphers. For further reading on Vernam's machine, the book by Kahn [97] is recommended.

2.2 Random Numbers and an Unbreakable Stream Cipher

2.2.1 Random Number Generators

As we saw in the previous section, the actual encryption and decryption of stream ciphers is extremely simple. The security of stream ciphers hinges entirely on a "suitable" key stream $s_0, s_1, s_2, \ldots$. Since randomness plays a major role, we will first learn about the three types of random number generators (RNG) that are important for us.

True Random Number Generators (TRNG)

True random number generators (TRNGs) are characterized by the fact that their output cannot be reproduced. For instance, if we flip a coin 100 times and record the resulting sequence of 100 bits, it will be virtually impossible for anyone on Earth to generate the same 100 bit sequence. The chance of success is $1/2^{100}$, which is an extremely small probability. TRNGs are based on physical processes. Examples include coin flipping, rolling of dice, semiconductor noise, clock jitter in digital circuits and radioactive decay. In cryptography, TRNGs are often needed for generating session keys, which are then distributed between Alice and Bob, and for other purposes.

(General) Pseudorandom Number Generators (PRNG)

Pseudorandom number generators (PRNGs) generate sequences which are *computed* from an initial seed value. Often they are computed recursively in the following way:

$$\begin{aligned} s_0 &= \text{seed} \\ s_{i+1} &= f(s_i), \quad i = 0, 1, \ldots \end{aligned}$$

A generalization of this are generators of the form $s_{i+1} = f(s_i, s_{i-1}, \ldots, s_{i-t})$, where t is a fixed integer. A popular example is the *linear congruential generator*:

$$\begin{aligned} s_0 &= \text{seed} \\ s_{i+1} &\equiv a\,s_i + b \bmod m, \quad i = 0, 1, \ldots \end{aligned}$$

where a, b, m are integer constants. Note that PRNGs are not random in a true sense because they can be computed and are thus completely deterministic. A widely used example is the *rand()* function used in ANSI C. It has the parameters:

$$\begin{aligned} s_0 &= 12345 \\ s_{i+1} &\equiv 1103515245\,s_i + 12345 \bmod 2^{31}, \quad i = 0, 1, \ldots \end{aligned}$$

A common requirement of PRNGs is that they possess good statistical properties, meaning their output approximates a sequence of true random numbers. There are many mathematical tests, e.g., the chi-square test, which can verify the statistical behavior of PRNG sequences. Note that there are many, many applications for pseudorandom numbers outside cryptography. For instance, many types of simulations or testing, e.g., of software or of VLSI chips, need random data as input. That is the reason why a PRNG is included in the ANSI C specification.

Cryptographically Secure Pseudorandom Number Generators (CSPRNG)

Cryptographically secure pseudorandom number generators (CSPRNGs) are a special type of PRNG which possess the following additional property: A CSPRNG is PRNG which is *unpredictable*. Informally, this means that given n output bits of the key stream $s_i, s_{i+1}, \ldots, s_{i+n-1}$, where n is some integer, it is computationally infeasible to compute the subsequent bits $s_{i+n}, s_{i+n+1}, \ldots$. A more exact definition is that given n consecutive bits of the key stream, there is no polynomial time algorithm that can predict the next bit s_{n+1} with better than 50% chance of success. Another property of CSPRNG is that given the above sequence, it should be computationally infeasible to compute any preceding bits $s_{i-1}, s_{i-2}, \ldots$.

Note that the need for unpredictability of CSPRNGs is unique to cryptography. In virtually all other situations where pseudorandom numbers are needed in computer science or engineering, unpredictability is not needed. As a consequence, the distinction between PRNG and CSPRN and their relevance for stream ciphers is often not clear to non-cryptographers. Almost all PRNG that were designed without the clear purpose of being stream ciphers are not CSPRNGs.

2.2.2 The One-Time Pad

In the following we discuss what happens if we use the three types of random numbers as generators for the key stream sequence $s_0, s_1, s_2, \ldots$ of a stream cipher. Let's first define what a perfect cipher should be:

Definition 2.2.1 Unconditional Security
A cryptosystem is unconditionally or information-theoretically secure if it cannot be broken even with infinite computational resources.

Unconditional security is based on information theory and assumes no limit on the attacker's computational power. This looks like a pretty straightforward definition. It is in fact straightforward, but the requirements for a cipher to be unconditionally secure are tremendous. Let's look at it using a gedankenexperiment: Assume we have a symmetric encryption algorithm (it doesn't matter whether it's a block cipher or stream cipher) with a key length of 10,000 bits, and the only attack that works is an exhaustive key search, i.e, a brute-force attack. From the discussion in Sect. 1.3.2, we recall that 128 bits are more than enough for long-term security. So, is a cipher with 10,000 bits unconditionally secure? The answer is simple: No! Since an attacker can have *infinite* computational resources, we can simply assume that the attacker has 2^{10000} computers available and every computer checks exactly one key. This will give us a correct key in one time step. Of course, there is no way that 2^{10000} computer can ever be built, the number is too large. (It is estimated that there are “only” about 2^{266} atoms in the known universe.) The cipher would merely be *computationally secure* but not unconditionally.

All this said, we now show a way to build an unconditionally secure cipher that is quite simple. This cipher is called the One-Time Pad.

Definition 2.2.2 One-Time Pad (OTP)
A stream cipher for which

1. *the key stream $s_0, s_1, s_2, \ldots$ is generated by a true random number generator, and*
2. *the key stream is only known to the legitimate communicating parties, and*
3. *every key stream bit s_i is only used once*

is called a one-time pad. The one-time pad is unconditionally secure.

It is easy to show why the OTP is unconditionally secure. Here is a sketch of a proof. For every ciphertext bit we get an equation of this form:

$$\begin{aligned} y_0 &\equiv x_0 + s_0 \bmod 2 \\ y_1 &\equiv x_1 + s_1 \bmod 2 \\ &\vdots \end{aligned}$$

Each individual relation is a linear equation modulo 2 with two unknowns. They are impossible to solve. If the attacker knows the value for y_0 (0 or 1), he cannot determine the value of x_0. In fact, the solutions $x_0 = 0$ and $x_0 = 1$ are exactly equally likely if s_0 stems from a truly random source and there is 50% chance that it has the value 0 and 1. The situation is identical for the second equation and all subsequent ones. Note that the situation is different if the values s_i are not truly random. In this case, there is some functional relationship between them, and the equations shown above are not independent. Even though it might still be hard to solve the system of equations, it is not provably secure!

So, now we have a simple cipher which is perfectly secure. There are rumors that the red telephone between the White House and the Kremlin was encrypted using an OTP during the Cold War. Obviously there must be a catch since OTPs are not used for Web browsers, e-mail encryption, smart cards, mobile phones, or other important applications. Let’s look at the implications of the three requirements in Defintion 2.2.2. The first requirement means that we need a TRNG. That means we need a device, e.g., based on white noise of a semiconductor, that generates truly random bits. Since standard PCs do not have TRNG, this requirement might not be that convenient but can certainly be met. The second requirement means that Alice has to get the random bits securely to Bob. In practice that could mean that Alice burns the true random bits on a CD ROM and sends them securely, e.g., with a trusted courier, to Bob. Still doable, but not great. The third requirement is probably the most impractical one: Key stream bits cannot be re-used. *This implies that we need one key bit for every bit of plaintext.* Hence, our key is as long as the plaintext! This is probably the major drawback of the OTP. Even if Alice and Bob share a CD with 1 MByte of true random numbers, we run quickly into limits. If they send a single email with an attachment of 1 MByte, they could encrypt and decrypt it, but after that they would need to exchange a true random key stream again.

For these reasons OTPs are rarely used in practice. However, they give us a great design idea for secure ciphers: If we XOR truly random bits and plaintext, we get ciphertext that can certainly not be broken by an attacker. We will see in the next section how we can use this fact to build practical stream ciphers.

2.2.3 Towards Practical Stream Ciphers

In the previous section we saw that OTPs are unconditionally secure, but that they have drawbacks which make them impractical. What we try to do with practical stream ciphers is to replace the truly random key stream bits by a pseudorandom number generator where the key k serves as a seed. The principle of practical stream ciphers is shown in Fig. 2.5.

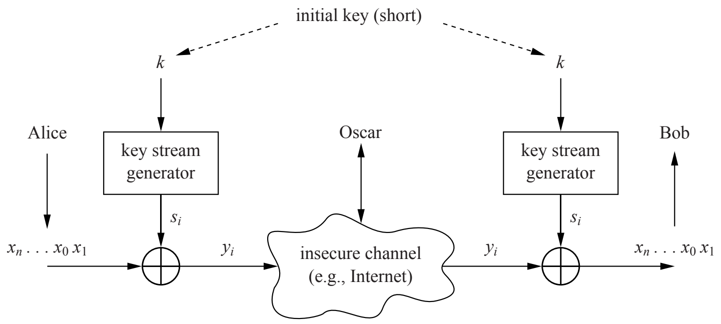

Fig. 2.5 Practical stream ciphers

Before we turn to stream ciphers used in the real world, it should be stressed that practical stream ciphers are not unconditionally secure. In fact, *all* known practical crypto algorithms (stream ciphers, block ciphers, public-key algorithms) are not unconditionally secure. The best we can hope for is *computational security*, which we define as follows:

Definition 2.2.3 Computational Security
A cryptosystem is computationally secure *if the best known algorithm for breaking it requires at least* t *operations.*

This seems like a reasonable definition, but there are still several problems with it. First, often we do not know what the best algorithm for a given attack is. A prime example is the RSA public-key scheme, which can be broken by factoring large integers. Even though many factoring algorithms are known, we do not know whether there exist any better ones. Second, even if a lower bound on the complexity of one attack is known, we do not know whether any other, more powerful attacks are possible. We saw this in Sect. 1.2.2 during the discussion about the substitution cipher: Even though we know the exact computational complexity for an exhaustive key search, there exist other more powerful attacks. The best we can do in practice is to design crypto schemes for which it is *assumed* that they are computationally secure. For symmetric ciphers this usually means one hopes that there is no attack method with a complexity better than an exhaustive key search.

Let's go back to Fig. 2.5. This design emulates ("behaves like") a one-time pad. It has the major advantage over the OTP that Alice and Bob only need to exchange a secret key that is at most a few 100 bits long, and that does not have to be as long as the message we want to encrypt. We now have to think carefully about the properties of the key stream $s_0, s_1, s_2, \ldots$ that is generated by Alice and Bob. Obviously, we need some type of random number generator to derive the key stream. First, we note that we cannot use a TRNG since, by definition, Alice and Bob will not be able to generate the same key stream. Instead we need deterministic, i.e., pseudorandom, number generators. We now look at the other two generators that were introduced in the previous section.

Building Key Streams from PRNGs

Here is an idea that seems promising (but in fact is pretty bad): Many PRNGs possess good statistical properties, which are necessary for a strong stream cipher. If we apply statistical tests to the key stream sequence, the output should pretty much behave like the bit sequence generated by tossing a coin. So it is tempting to assume that a PRNG can be used to generate the key stream. But all of this is not sufficient for a stream cipher since our opponent, Oscar, is smart. Consider the following attack:

Example 2.2. Let's assume a PRNG based on the linear congruential generator:

$$\begin{aligned} S_0 &= \text{seed} \\ S_{i+1} &\equiv A\,S_i + B \bmod m, \quad i = 0, 1, \ldots \end{aligned}$$

where we choose m to be 100 bits long and $S_i, A, B \in \{0, 1, \ldots, m-1\}$. Note that this PRNG can have excellent statistical properties if we choose the parameters carefully. The modulus m is part of the encryption scheme and is publicly known. The secret key comprises the values (A, B) and possibly the seed S_0, each with a length of 100. That gives us a key length of 200 bit, which is more than sufficient to protect against a brute-force attack. Since this is a stream cipher, Alice can encrypt:

$$y_i \equiv x_i \, + \, s_i \bmod 2$$

where s_i are the bits of the binary representation of the PRNG output symbols S_j.

But Oscar can easily launch an attack. Assume he knows the first 300 bits of plaintext (this is only 300/8=37.5 byte), e.g., file header information, or he guesses part of the plaintext. Since he certainly knows the ciphertext, he can now compute the first 300 bits of key stream as:

$$s_i \equiv y_i \, + \, x_i \bmod m \ , \ i = 1,2,\ldots,300$$

These 300 bits immediately give the first three output symbols of the PRNG: $S_1 = (s_1,\ldots,s_{100})$, $S_2 = (s_{101},\ldots,s_{200})$ and $S_3 = (s_{201},\ldots,s_{300})$. Oscar can now generate two equations:

$$\begin{aligned} S_2 &\equiv A\,S_1 + B \bmod m \\ S_3 &\equiv A\,S_2 + B \bmod m \end{aligned}$$

This is a system of linear equations over $\mathbb{Z}_m$ with two unknowns A and B. But those two values are the key, and we can immediately solve the system, yielding:

$$\begin{aligned} A &\equiv (S_2 - S_3)/(S_1 - S_2) \bmod m \\ B &\equiv S_2 - S_1(S_2 - S_3)/(S_1 - S_2) \bmod m \end{aligned}$$

In case $\gcd((S_1 - S_2), m)) \neq 1$ we get multiple solutions since this is an equation system over $\mathbb{Z}_m$. However, with a fourth piece of known plaintext the key can uniquely be detected in almost all cases. Alternatively, Oscar simply tries to encrypt the message with each of the multiple solutions found. Hence, in summary: if we know a few pieces of plaintext, we can compute the key and decrypt the entire ciphertext!
$\diamond$

This type of attack is why the notation of CSPRNG was invented.

Building Key Streams from CSPRNGs

What we need to do to prevent the attack above is to use a CSPRNG, which assures that the key stream is unpredictable. We recall that this means that given the first n output bits of the key stream $s_1, s_2, \ldots, s_n$, it is computationally infeasible to compute the bits $s_{n+1}, s_{n+2}, \ldots$. Unfortunately, pretty much all pseudorandom number generators that are used for applications outside cryptography are *not* cryptographically secure. Hence, in practice, we need to use specially designed pseudorandom number generators for stream ciphers.

The question now is how practical stream ciphers actually look. There are many proposals for stream ciphers out in the literature. They can roughly be classified as ciphers either optimized for software implementation or optimized for hardware implementation. In the former case, the ciphers typically require few CPU instructions to compute one key stream bit. In the latter case, they tend to be based on operations which can easily be realized in hardware. A popular example is shift registers with feedback, which are discussed in the next section. A third class of stream ciphers is realized by using block ciphers as building blocks. The cipher feedback mode, output feedback mode and counter mode to be introduced in Chap. 5 are examples of stream ciphers derived from block ciphers.

It could be argued that the state-of-the-art in block cipher design is more advanced than stream ciphers. Currently it seems to be easier for scientists to design "secure" block ciphers than stream ciphers. Subsequent chapters deal in great detail with the two most popular and standardized block ciphers, DES and AES.

2.3 Shift Register-Based Stream Ciphers

As we have learned so far, practical stream ciphers use a stream of key bits $s_1, s_2, \ldots$ that are generated by the key stream generator, which should have certain properties. An elegant way of realizing long pseudorandom sequences is to use linear feedback shift registers (LFSRs). LFSRs are easily implemented in hardware and many, but certainly not all, stream ciphers make use of LFSRs. A prominent example is the A5/1 cipher, which is standardized for voice encryption in GSM. As we will see, even though a plain LFSR produces a sequence with good statistical properties, it is cryptographically weak. However, combinations of LFSRs, such as A5/1 or the cipher Trivium, can make secure stream ciphers. It should be stressed that there are many ways for constructing stream ciphers. This section only introduces one of several popular approaches.

2.3.1 Linear Feedback Shift Registers (LFSR)

An LFSR consists of clocked storage elements (*flip-flops*) and a *feedback path*. The number of storage elements gives us the *degree* of the LFSR. In other words, an LFSR with m flip-flops is said to be of degree m. The feedback network computes the input for the last flip-flop as XOR-sum of certain flip-flops in the shift register.

Example 2.3. **Simple LFSR** We consider an LFSR of degree $m = 3$ with flip-flops FF_2, FF_1, FF_0, and a feedback path as shown in Fig. 2.6. The internal state bits are denoted by s_i and are shifted by one to the right with each clock tick. The rightmost state bit is also the current output bit. The leftmost state bit is computed in the feedback path, which is the XOR sum of some of the flip-flop values in the previous clock period. Since the XOR is a linear operation, such circuits are called linear feedback shift registers. If we assume an initial state of $(s_2 = 1, s_1 = 0, s_0 = 0)$, Table 2.2 gives the complete sequence of states of the LFSR. Note that the rightmost column is the output of the LFSR. One can see from this example that the LFSR

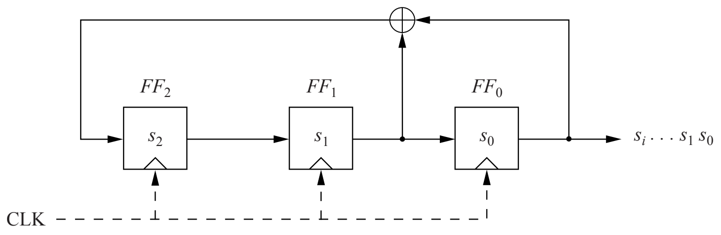

Fig. 2.6 Linear feedback shift register of degree 3 with initial values s_2, s_1, s_0

Table 2.2 Sequence of states of the LFSR

clk	FF_2	FF_1	$FF_0 = s_i$
0	1	0	0
1	0	1	0
2	1	0	1
3	1	1	0
4	1	1	1
5	0	1	1
6	0	0	1
7	1	0	0
8	0	1	0

starts to repeat after clock cycle 6. This means the LFSR output has period of length 7 and has the form:

$$0010111\ 0010111\ 0010111\ \dots$$

There is a simple formula which determines the functioning of this LFSR. Let's look at how the output bits s_i are computed, assuming the initial state bits s_0, s_1, s_2:

$$\begin{aligned} s_3 &\equiv s_1 + s_0 \bmod 2 \\ s_4 &\equiv s_2 + s_1 \bmod 2 \\ s_5 &\equiv s_3 + s_2 \bmod 2 \\ &\vdots \end{aligned}$$

In general, the output bit is computed as:

$$s_{i+3} \equiv s_{i+1} + s_i \mod 2$$

where $i = 0, 1, 2, \dots$

$\diamond$

This was, of course, a simple example. However, we could already observe many important properties. We will now look at general LFSRs.

A Mathematical Description of LFSRs

The general form of an LFSR of degree m is shown in Fig. 2.7. It shows m flip-flops and m possible feedback locations, all combined by the XOR operation. Whether a feedback path is active or not, is defined by the *feedback coefficient* $p_0, p_1, \ldots, p_{m-1}$:

- If $p_i = 1$ (closed switch), the feedback is active.
- If $p_i = 0$ (open switch), the corresponding flip-flop output is not used for the feedback.

With this notation, we obtain an elegant mathematical description for the feedback path. If we *multiply* the output of flip-flop i by its coefficient p_i, the result is either the output value if $p_i = 1$, which corresponds to a closed switch, or the value zero if $p_i = 0$, which corresponds to an open switch. The values of the feedback coefficients are crucial for the output sequence produced by the LFSR.

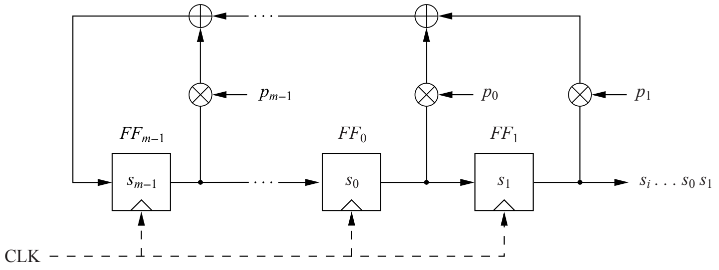

Fig. 2.7 General LFSR with feedback coefficients p_i and initial values $s_{m-1}, \ldots, s_0$

Let's assume the LFSR is initially loaded with the values $s_0, \ldots, s_{m-1}$. The next output bit of the LFSR s_m, which is also the input to the leftmost flip-flop, can be computed by the XOR-sum of the products of flip-flop outputs and corresponding feedback coefficient:

$$s_m \equiv s_{m-1}p_{m-1} + \cdots + s_1 p_1 + s_0 p_0 \bmod 2$$

The next LFSR output can be computed as:

$$s_{m+1} \equiv s_m p_{m-1} + \cdots + s_2 p_1 + s_1 p_0 \bmod 2$$

In general, the output sequence can be described as:

$$s_{i+m} \equiv \sum_{j=0}^{m-1} p_j \cdot s_{i+j} \bmod 2; \quad s_i, p_j \in \{0,1\};\ i = 0,1,2,\ldots \tag{2.1}$$

Clearly, the output values are given through a combination of some previous output values. LFSRs are sometimes referred to as *linear recurrences*.

Due to the finite number of recurring states, the output sequence of an LFSR repeats periodically. This was also illustrated in Example 2.3. Moreover, an LFSR can produce output sequences of different lengths, depending on the feedback coefficients. The following theorem gives us the *maximum length* of an LFSR as function of its degree.

> **Theorem 2.3.1** *The maximum sequence length generated by an LFSR of degree m is* $2^m - 1$.

It is easy to show that this theorem holds. The *state* of an LFSR is uniquely determined by the m internal register bits. Given a certain state, the LFSR deterministically assumes its next state. Because of this, as soon as an LFSR assumes a previous state, it starts to repeat. Since an m-bit state vector can only assume $2^m - 1$ nonzero states, the maximum sequence length before repetition is $2^m - 1$. Note that the all-zero state must be excluded. If an LFSR assumes this state, it will get "stuck" in it, i.e., it will never be able to leave it again. Note that only certain configurations $(p_0, \ldots, p_{m-1})$ yield maximum length LFSRs. We give a small example for this below.

Example 2.4. LFSR with maximum-length output sequence
Given an LFSR of degree $m = 4$ and the feedback path $(p_3 = 0, p_2 = 0, p_1 = 1, p_0 = 1)$, the output sequence of the LFSR has a period of $2^m - 1 = 15$, i.e., it is a maximum-length LFSR.
◇

Example 2.5. LFSR with non-maximum output sequence
Given an LFSR of degree $m = 4$ and $(p_3 = 1, p_2 = 1, p_1 = 1, p_0 = 1)$, then the output sequence has period of 5; therefore, it is not a maximum-length LFSR. ◇

The mathematical background of the properties of LFSR sequences is beyond the scope of this book. However, we conclude this introduction to LFSRs with some additional facts. LFSRs are often specified by polynomials using the following notation: An LFSR with a feedback coefficient vector $(p_{m-1}, \ldots, p_1, p_0)$ is represented by the polynomial

$$P(x) = x^m + p_{m-1}x^{m-1} + \ldots + p_1 x + p_0$$

For instance, the LFSR from the example above with coefficients $(p_3 = 0, p_2 = 0, p_1 = 1, p_0 = 1)$ can alternatively be specified by the polynomial $x^4 + x + 1$. This seemingly odd notation as a polynomial has several advantages. For instance, maximum-length LFSRs have what is called *primitive polynomials*. Primitive polynomials are a special type of irreducible polynomial. Irreducible polynomials are roughly comparable with prime numbers, i.e., their only factors are 1 and the polynomial itself. Primitive polynomials can relatively easily be computed. Hence, maximum-length LFSRs can easily be found. Table 2.3 shows one primitive polynomial for every value of m in the range from $m = 2, 3, \ldots, 128$. As an example, the notation $(0,2,5)$ refers to the polynomial $1+x^2+x^5$. Note that there are many primitive polynomials for every given degree m. For instance, there exist 69,273,666 different primitive polynomials of degree $m = 31$.

Table 2.3 Primitive polynomials for maximum-length LFSRs

(0,1,2)	(0,1,3,4,24)	(0,1,46)	(0,1,5,7,68)	(0,2,3,5,90)	(0,3,4,5,112)
(0,1,3)	(0,3,25)	(0,5,47)	(0,2,5,6,69)	(0,1,5,8,91)	(0,2,3,5,113)
(0,1,4)	(0,1,3,4,26)	(0,2,3,5,48)	(0,1,3,5,70)	(0,2,5,6,92)	(0,2,3,5,114)
(0,2,5)	(0,1,2,5,27)	(0,4,5,6,49)	(0,1,3,5,71)	(0,2,93)	(0,5,7,8,115)
(0,1,6)	(0,1,28)	(0,2,3,4,50)	(0,3,9,10,72)	(0,1,5,6,94)	(0,1,2,4,116)
(0,1,7)	(0,2,29)	(0,1,3,6,51)	(0,2,3,4,73)	(0,11,95)	(0,1,2,5,117)
(0,1,3,4,8)	(0,1,30)	(0,3,52)	(0,1,2,6,74)	(0,6,9,10,96)	(0,2,5,6,118)
(0,1,9)	(0,3,31)	(0,1,2,6,53)	(0,1,3,6,75)	(0,6,97)	(0,8,119)
(0,3,10)	(0,2,3,7,32)	(0,3,6,8,54)	(0,2,4,5,76)	(0,3,4,7,98)	(0,1,3,4,120)
(0,2,11)	(0,1,3,6,33)	(0,1,2,6,55)	(0,2,5,6,77)	(0,1,3,6,99)	(0,1,5,8,121)
(0,3,12)	(0,1,3,4,34)	(0,2,4,7,56)	(0,1,2,7,78)	(0,2,5,6,100)	(0,1,2,6,122)
(0,1,3,4,13)	(0,2,35)	(0,4,57)	(0,2,3,4,79)	(0,1,6,7,101)	(0,2,123)
(0,5,14)	(0,2,4,5,36)	(0,1,5,6,58)	(0,2,4,9,80)	(0,3,5,6,102)	(0,37,124)
(0,1,15)	(0,1,4,6,37)	(0,2,4,7,59)	(0,4,81)	(0,9,103)	(0,5,6,7,125)
(0,1,3,5,16)	(0,1,5,6,38)	(0,1,60)	(0,4,6,9,82)	(0,1,3,4,104)	(0,2,4,7,126)
(0,3,17)	(0,4,39)	(0,1,2,5,61)	(0,2,4,7,83)	(0,4,105)	(0,1,127)
(0,3,18)	(0,3,4,5,40)	(0,3,5,6,62)	(0,5,84)	(0,1,5,6,106)	(0,1,2,7,128)
(0,1,2,5,19)	(0,3,41)	(0,1,63)	(0,1,2,8,85)	(0,4,7,9,107)	
(0,3,20)	(0,1,2,5,42)	(0,1,3,4,64)	(0,2,5,6,86)	(0,1,4,6,108)	
(0,2,21)	(0,3,4,6,43)	(0,1,3,4,65)	(0,1,5,7,87)	(0,2,4,5,109)	
(0,1,22)	(0,5,44)	(0,3,66)	(0,8,9,11,88)	(0,1,4,6,110)	
(0,5,23)	(0,1,3,4,45)	(0,1,2,5,67)	(0,3,5,6,89)	(0,2,4,7,111)	

2.3.2 Known-Plaintext Attack Against Single LFSRs

As indicated by its name, LFSRs are linear. Linear systems are governed by linear relationships between their inputs and outputs. Since linear dependencies can relatively easily be analyzed, this can be a major advantage, e.g., in communication systems. However, a cryptosystem where the key bits only occur in linear relationships makes a highly insecure cipher. We will now investigate how the linear behavior of a LFSR leads to a powerful attack.

If we use an LFSR as a stream cipher, the secret key k is the feedback coefficient vector $(p_{m-1},\ldots,p_1,p_0)$. An attack is possible if the attacker Oscar knows some plaintext and the corresponding ciphertext. We further assume that Oscar knows the degree m of the LFSR. The attack is so efficient that he can easily try a large number of possible m values, so that this assumption is not a major restriction. Let the known plaintext be given by $x_0,x_1,\ldots,x_{2m-1}$ and the corresponding ciphertext by $y_0,y_1,\ldots,y_{2m-1}$. With these $2m$ pairs of plaintext and ciphertext bits, Oscar reconstructs the first $2m$ key stream bits:

$$s_i \equiv x_i + y_i \bmod 2; \quad i = 0,1,\ldots,2m-1.$$

The goal is now to find the key which is given by the feedback coefficients p_i.

Eq. (2.1) is a description of the relationship of the unknown key bits p_i and the key stream output. We repeat the equation here for convenience:

$$s_{i+m} \equiv \sum_{j=0}^{m-1} p_j \cdot s_{i+j} \bmod 2; \quad s_i, p_j \in \{0,1\}; \quad i = 0,1,2,\ldots$$

Note that we get a different equation for every value of i. Moreover, the equations are linearly independent. With this knowledge, Oscar can generate m equations for the first m values of i:

$$\begin{array}{llll} i=0, & s_m & \equiv p_{m-1}s_{m-1} + \ldots + p_1 s_1 + p_0 s_0 & \bmod 2 \\ i=1, & s_{m+1} & \equiv p_{m-1}s_m + \ldots + p_1 s_2 + p_0 s_1 & \bmod 2 \\ \vdots & \vdots & \vdots \;\vdots & \vdots \\ i=m-1, & s_{2m-1} & \equiv p_{m-1}s_{2m-2} + \ldots + p_1 s_m + p_0 s_{m-1} & \bmod 2 \end{array} \tag{2.2}$$

He has now m linear equations in m unknowns $p_0, p_1, \ldots, p_{m-1}$. This system can easily be solved by Oscar using Gaussian elimination, matrix inversion or any other algorithm for solving systems of linear equations. Even for large values of m, this can be done easily with a standard PC.

This situation has major consequences: **as soon as Oscar knows $2m$ output bits of an LFSR of degree m, the p_i coefficients can exactly be constructed by merely solving a system of linear equations**. Once he has computed these feedback coefficients, he can "build" the LFSR and load it with any m consecutive output bits that he already knows. Oscar can now clock the LFSR and produce the entire output sequence. Because of this powerful attack, LFSRs by themselves are extremely insecure! They are a good example of a PRNG with good statistical properties but with terrible cryptographical ones. Nevertheless, all is not lost. There are many stream ciphers which use *combinations* of several LFSRs to build strong cryptosystems. The cipher Trivium in the next section is an example.

2.3.3 Trivium

Trivium is a relatively new stream cipher which uses an 80-bit key. It is based on a combination of three shift registers. Even though these are feedback shift registers, there are nonlinear components used to derive the output of each register, unlike the LFSRs that we studied in the previous section.

Description of Trivium

As shown in Fig. 2.8, at the heart of Trivium are three shift registers, A, B and C. The lengths of the registers are 93, 84 and 111, respectively. The XOR-sum of all three register outputs forms the key stream s_i. A specific feature of the cipher is that

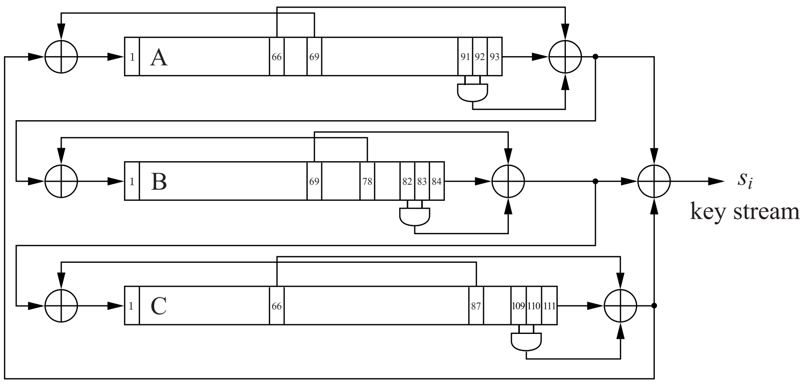

Fig. 2.8 Internal structure of the stream cipher Trivium

the output of each register is connected to the input of another register. Thus, the registers are arranged in circle-like fashion. The cipher can be viewed as consisting of one circular register with a total length of $93+84+111=288$. Each of the three registers has similar structure as described below.

The input of each register is computed as the XOR-sum of two bits:

- The output bit of another register according to Fig. 2.8. For instance, the output of register A is part of the input of register B.
- One register bit at a specific location is fed back to the input. The positions are given in Table 2.4. For instance, bit 68 of register A is fed back to its input.

The output of each register is computed as the XOR-sum of three bits:

- The rightmost register bit.
- One register bit at a specific location is fed forward to the output. The positions are given in Table 2.4. For instance, bit 66 of register A is fed to its output.
- The output of a logical AND function whose input is two specific register bits. Again, the positions of the AND gate inputs are given in Table 2.4.

Table 2.4 Specification of Trivium

	register length	feedback bit	feedforward bit	AND inputs
A	93	69	66	91, 92
B	84	78	69	82, 83
C	111	87	66	109, 110

Note that the AND operation is equal to multiplication in modulo 2 arithmetic. If we multiply two unknowns, and the register contents are the unknowns that an attacker wants to recover, the resulting equations are no longer linear as they contain products of two unknowns. Thus, the feedforward paths involving the AND operation are crucial for the security of Trivium as they prevent attacks that exploit the linearity of the cipher, as the one applicable to plain LFSRs shown in the previous section.

Encryption with Trivium

Almost all modern stream ciphers have two input parameters: a key k and an initialization vector IV. The former is the regular key that is used in every symmetric crypto system. The IV serves as a randomizer and should take a new value for every encryption session. It is important to note that the IV does not have to be kept secret, it merely must change for every session. Such values are often referred to as *nonces*, which stands for "number used once". Its main purpose is that two key streams produced by the cipher should be different, even though the key has not changed. If this were not the case, the following attack becomes possible. If an attacker has known plaintext from a first encryption, he can compute the corresponding key stream. The second encryption using the same key stream can now immediately be deciphered. Without a changing IV, stream cipher encryption is highly deterministic. Methods for generating IVs are discussed in Sect. 5.1.2. Let's look at the details of running Trivium:

Initialization Initially, an 80-bit IV is loaded into the 80 leftmost locations of register A, and an 80-bit key is loaded in the 80 leftmost locations of register B. All other register bits are set to zero with the exception of the three rightmost bits of register C, i.e., bits c_{109}, c_{110} and c_{111}, which are set to 1.

Warm-up Phase In the first phase, the cipher is clocked $4 \times 288 = 1152$ times. No cipher output is generated.

Encryption Phase The bits produced hereafter, i.e., starting with the output bit of cycle 1153, form the key stream.

The warm-up phase is needed for randomizing the cipher sufficiently. It makes sure that the key stream depends on both the key k and the IV.

An attractive feature of Trivium is its compactness, especially if implemented in hardware. It mainly consists of a 288-bit shift register and a few Boolean operations. It is estimated that a hardware implementation of the cipher occupies and area of between about 3500 and 5500 gate equivalences, depending on the degree of parallelization. (A gate equivalence is the chip area occupied by a 2-input NAND gate.) For instance, an implementation with 4000 gates computes the key stream at a rate of 16 bits/clock cycle. This is considerably smaller than most block ciphers such as AES and is very fast. If we assume that this hardware design is clocked at a moderate 125 MHz, the encryption rate would be $16\text{bit} \times 125\text{MHz} = 2$ Gbit/sec. In software, it is estimated that computing 8 output bits takes 12 cycles on a 1.5 GHz Intel CPU, resulting in a theoretical encryption rate of 1 Gbit/sec.

Even though there are no known attacks at the time of writing, one should keep in mind that Trivium is a relatively new cipher and attacks in the future are certainly a possibility. In the past, many other stream ciphers were found to be not secure. More information on Trivium can be found in [164].

2.4 Discussion and Further Reading

Established Stream Ciphers Even though many stream ciphers have been proposed over the years, there are considerably fewer well-investigated ones. The security of many proposed stream ciphers is unknown, and many stream ciphers have been broken. In the case of software-oriented stream ciphers, arguably the best-investigated ones are RC4 [144] and SEAL [120, Sect. 6.4.1]. Note that there are some known weaknesses in RC4, even though it is still secure in practice if it is used correctly [142]. The SEAL cipher, on the other hand, is patented.

In the case of hardware-oriented ciphers, there is a wealth of LFSR-based algorithms. Many proposed ciphers have been broken; see references [8, 85] for an introduction. Among the best-studied ones are the A5/1 and A5/2 algorithms which are used in GSM mobile networks for voice encryption between cell phones and base stations. A5/1, which is the cipher used in most industrialized nations, had originally been kept secret but was reverse-engineered and published on the Internet in 1998. The cipher is borderline secure today [22], whereas the weaker A5/2 has much more serious flaws [11]. Neither of the two ciphers is recommended based on today's understanding of cryptanalysis. For 3GPP mobile communication, a different cipher A5/3 (also named *KASUMI*) is used, but it is a block cipher.

This somewhat pessimistic outlook on the state-of-the-art in stream ciphers changed with the eSTREAM project, described below.

eSTREAM Project The *eSTREAM project* had the explicit goal to advance the state-of-the-art knowledge about stream cipher design. As part of this objective, new stream ciphers that might become suitable for widespread adoption were investigated. eSTREAM was organized by the European Network of Excellence in Cryptography (ECRYPT). The call for stream ciphers was first issued in November 2004 and ended in 2008. The ciphers were divided into two "profiles", depending on the intended application:

- Profile 1: Stream ciphers for software applications with high throughput requirements.
- Profile 2: Stream ciphers for hardware applications with restricted resources such as limited storage, gate count, or power consumption.

Some cryptographers had emphasized the importance of including an authentication method, and hence two further profiles were also included to deal with ciphers that also provide authentication.

A total of 34 candidates were submitted to eSTREAM. At the end of the project four software-oriented ("Profile 1") ciphers were found to have desirable properties: *HC-128*, *Rabbit*, *Salsa20/12* and *SOSEMANUK*. With respect to hardware-oriented ciphers ("Profile 2"), the following three ciphers were selected: *Grain v1*, *MICKEY* *v2* and *Trivium*. Note that all of these are relatively new ciphers and only time will tell whether they are really cryptographically strong. The algorithm description, source code and the results of the four-year evaluation process are available online [69], and the official book provides more detailed information [146].

It is important to keep in mind that ECRYPT is not a standardization body, so the status of the eSTREAM finalist ciphers cannot be compared to that of AES at the end of its selection process (cf. Sect. 4.1).

True Random Number Generation We introduced in this chapter different classes of RNGs, and found that cryptographically secure pseudorandom number generators are of central importance for stream ciphers. For other cryptographic applications, true random number generators are important. For instance, true random numbers are needed for the generation of cryptographic keys which are then to be distributed. Many ciphers and modes of operation rely on initial values that are often generated from TRNGs. Also, many protocols require nonces (numbers used only once), which may stem from a TRNG. All TRNGs need to exploit some entropy source, i.e., some process which behaves truly randomly. Many TRNG designs have been proposed over the years. They can coarsely be classified as approaches that use specially designed hardware as an entropy source or as TRNGs that exploit external sources of randomness. Examples of the former are circuits with random behavior, e.g., that are based on semiconductor noise or on several uncorrelated oscillators. Reference [104, Chap. 5] contains a good survey. Examples of the latter ones are computer systems which measure the times between key strokes or the arrival times of packets at a network interface. In all these cases, one has to be extremely careful to make sure that the noise source in fact has enough entropy. There are many examples of TRNG designs which turned out to have poor random behavior and which constitute a serious security weakness, depending on how they are used. There are tools available that test the statistical properties of TRNG output sequences [56, 125]. There are also standards with which TRNGs can be formally evaluated [80].

2.5 Lessons Learned

- Stream ciphers are less popular than block ciphers in most domains such as Internet security. There are exceptions, for instance, the popular stream cipher RC4.
- Stream ciphers sometimes require fewer resources, e.g., code size or chip area, for implementation than block ciphers, and they are attractive for use in constrained environments such as cell phones.
- The requirements for a *cryptographically secure* pseudorandom number generator are far more demanding than the requirements for pseudorandom number generators used in other applications such as testing or simulation.

- The One-Time Pad is a provable secure symmetric cipher. However, it is highly impractical for most applications because the key length has to equal the message length.
- Single LFSRs make poor stream ciphers despite their good statistical properties. However, careful combinations of several LFSR can yield strong ciphers.

Problems

2.1. The stream cipher described in Definition 2.1.1 can easily be generalized to work in alphabets other than the binary one. For manual encryption, an especially useful one is a stream cipher that operates on letters.

1. Develop a scheme which operates with the letters A, B,..., Z, represented by the numbers 0,1,...,25. What does the key (stream) look like? What are the encryption and decryption functions?
2. Decrypt the following cipher text:
   `bsaspp kkuosp`
   which was encrypted using the key:
   `rsidpy dkawoa`
3. How was the young man murdered?

2.2. Assume we store a one-time key on a CD-ROM with a capacity of 1 Gbyte. Discuss the *real-life* implications of a One-Time-Pad (OTP) system. Address issues such as life cycle of the key, storage of the key during the life cycle/after the life cycle, key distribution, generation of the key, etc.

2.3. Assume an OTP-like encryption with a short key of 128 bit. This key is then being used periodically to encrypt large volumes of data. Describe how an attack works that breaks this scheme.

2.4. At first glance it seems as though an exhaustive key search is possible against an OTP system. Given is a short message, let's say 5 ASCII characters represented by 40 bit, which was encrypted using a 40-bit OTP. Explain *exactly* why an exhaustive key search will not succeed even though sufficient computational resources are available. This is a paradox since we know that the OTP is unconditionally secure. That is, explain why a brute-force attack does not work.

Note: You have to resolve the paradox! That means answers such as "The OTP is unconditionally secure and therefore a brute-force attack does not work" are not valid.

2.5. We will now analyze a pseudorandom number sequence generated by a LFSR characterized by $(c_2 = 1, c_1 = 0, c_0 = 1)$.

1. What is the sequence generated from the initialization vector $(s_2 = 1, s_1 = 0, s_0 = 0)$?
2. What is the sequence generated from the initialization vector $(s_2 = 0, s_1 = 1, s_0 = 1)$?
3. How are the two sequences related?

2.6. Assume we have a stream cipher whose period is quite short. We happen to know that the period is 150–200 bit in length. We assume that we do *not* know anything else about the internals of the stream cipher. In particular, we should not assume that it is a simple LFSR. For simplicity, assume that English text in ASCII format is being encrypted.

Describe in detail how such a cipher can be attacked. Specify exactly what Oscar has to know in terms of plaintext/ciphertext, and how he can decrypt all ciphertext.

2.7. Compute the first two output bytes of the LFSR of degree 8 and the feedback polynomial from Table 2.3 where the initialization vector has the value FF in hexadecimal notation.

2.8. In this problem we will study LFSRs in somewhat more detail. LFSRs come in three flavors:

- LFSRs which generate a maximum-length sequence. These LFSRs are based on *primitive polynomials*.
- LFSRs which do not generate a maximum-length sequence but whose sequence length is independent of the initial value of the register. These LFSRs are based on *irreducible polynomials* that are not primitive. Note that all primitive polynomials are also irreducible.
- LFSRs which do not generate a maximum-length sequence and whose sequence length depends on the initial values of the register. These LFSRs are based on *reducible polynomials*.

We will study examples in the following. Determine *all* sequences generated by

1. x^4+x+1
2. x^4+x^2+1
3. $x^4+x^3+x^2+x+1$

Draw the corresponding LFSR for each of the three polynomials. Which of the polynomials is primitive, which is only irreducible, and which one is reducible? Note that the lengths of all sequences generated by each of the LFSRs should add up to 2^m-1.

2.9. Given is a stream cipher which uses a single LFSR as key stream generator. The LFSR has a degree of 256.

1. How many plaintext/ciphertext bit pairs are needed to launch a successful attack?
2. Describe all steps of the attack in detail and develop the formulae that need to be solved.
3. What is the key in this system? Why doesn't it make sense to use the initial contents of the LFSR as the key or as part of the key?

2.10. We conduct a known-plaintext attack on an LFSR-based stream cipher. We know that the plaintext sent was:
`1001 0010 0110 1101 1001 0010 0110`
By tapping the channel we observe the following stream:
`1011 1100 0011 0001 0010 1011 0001`

1. What is the degree m of the key stream generator?
2. What is the initialization vector?
3. Determine the feedback coefficients of the LFSR.

4. Draw a circuit diagram and verify the output sequence of the LFSR.

2.11. We want to perform an attack on another LFSR-based stream cipher. In order to process letters, each of the 26 uppercase letters and the numbers 0, 1, 2, 3, 4, 5 are represented by a 5-bit vector according to the following mapping:

$$\begin{aligned} A &\leftrightarrow 0 = 00000_2 \\ &\vdots \\ Z &\leftrightarrow 25 = 11001_2 \\ 0 &\leftrightarrow 26 = 11010_2 \\ &\vdots \\ 5 &\leftrightarrow 31 = 11111_2 \end{aligned}$$

We happen to know the following facts about the system:

- The degree of the LFSR is $m = 6$.
- Every message starts with the header `WPI`.

We observe now on the channel the following message (the fourth letter is a zero):
`j5a0edj2b`

1. What is the initialization vector?
2. What are the feedback coefficients of the LFSR?
3. Write a program in your favorite programming language which generates the whole sequence, and find the whole plaintext.
4. Where does the thing after `WPI` live?
5. What type of attack did we perform?

2.12. Assume the *IV* and the key of Trivium each consist of 80 all-zero bits. Compute the first 70 bits $s_1, \ldots, s_{70}$ during the warm-up phase of Trivium. Note that these are only internal bits which are not used for encryption since the warm-up phase lasts for 1152 clock cycles.

Springer

http://www.springer.com/978-3-642-04100-6

Understanding Cryptography
A Textbook for Students and Practitioners
Paar, C.; Pelzl, J.
2010, XVIII, 372 p., Hardcover
ISBN: 978-3-642-04100-6