Cryptography

Week #5 Stream Ciphers

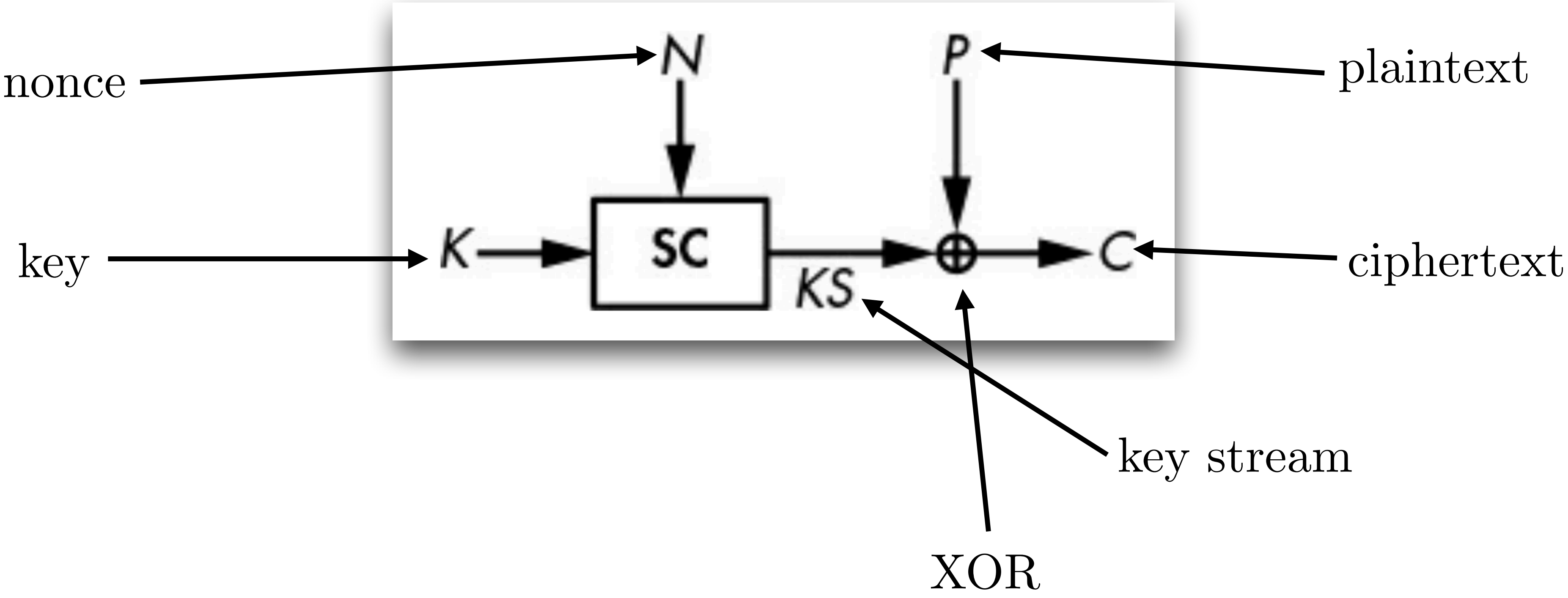
Rogério Reis, rogerio.reis@fc.up.pt
MSI/MCC/MIERSI - 2023/2024
DCC FCUP

13/10/2023

How do Stream Ciphers work?

They rely in the use of Deterministic Random Bit Generators (DRBG) instead of Pseudo Random Bit Generators (PRBG) because is the use of the deterministic property of the RBG that allow the correct decryption of the original data.

With a PRNG, you could encrypt but never decrypt—which is **very** secure, but **very** useless too.



nonce

plaintext

key

ciphertext

key stream

XOR

K - the key, that should remain secret. Usually with 128 or 256 bits

N - a **nonce**, that does not need to be kept in secret, but it should be unique for each key (an each “message”) and is usually between 64 and 128 bits.

$$KS = SC(K, N)$$

$$C = KS \oplus P$$

$$P = KS \oplus C$$

Stream ciphers allow you to encrypt a message with key K1 and nonce N1 and then encrypt another message with key K1 and nonce N2 that is different from N1, or with key K2, which is different from K1 and nonce N1.

However, you should never again encrypt with K1 and N1, because you would then use twice the same keystream KS.

Then...

$$C_1 = KS \oplus P_1$$

$$C_2 = KS \oplus P_2$$

If on can get the plaintext P_1, then

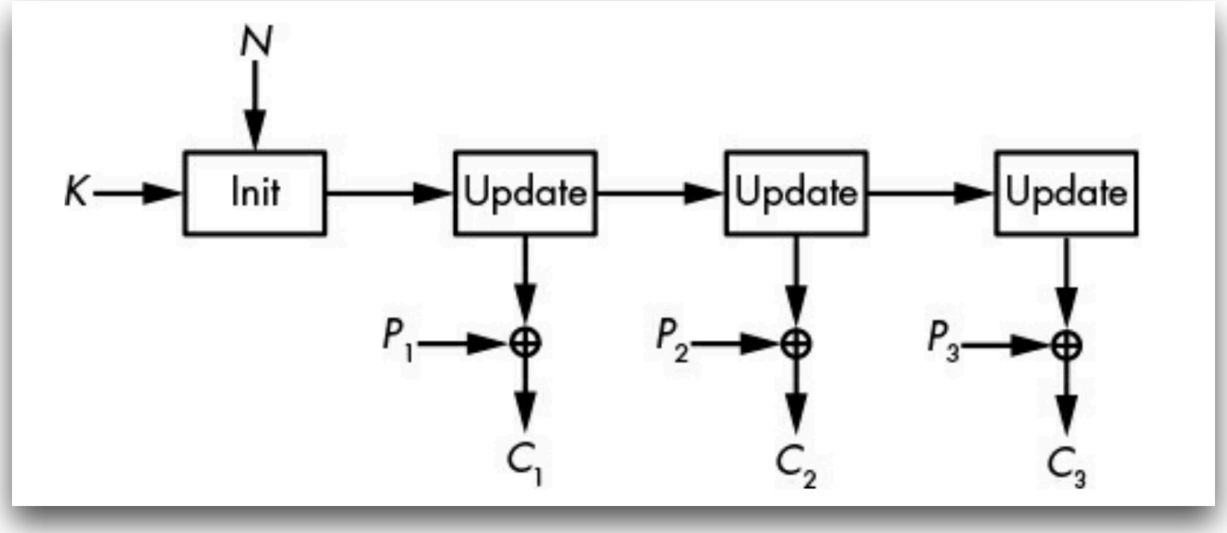
$$KS = C_1 \oplus P1$$

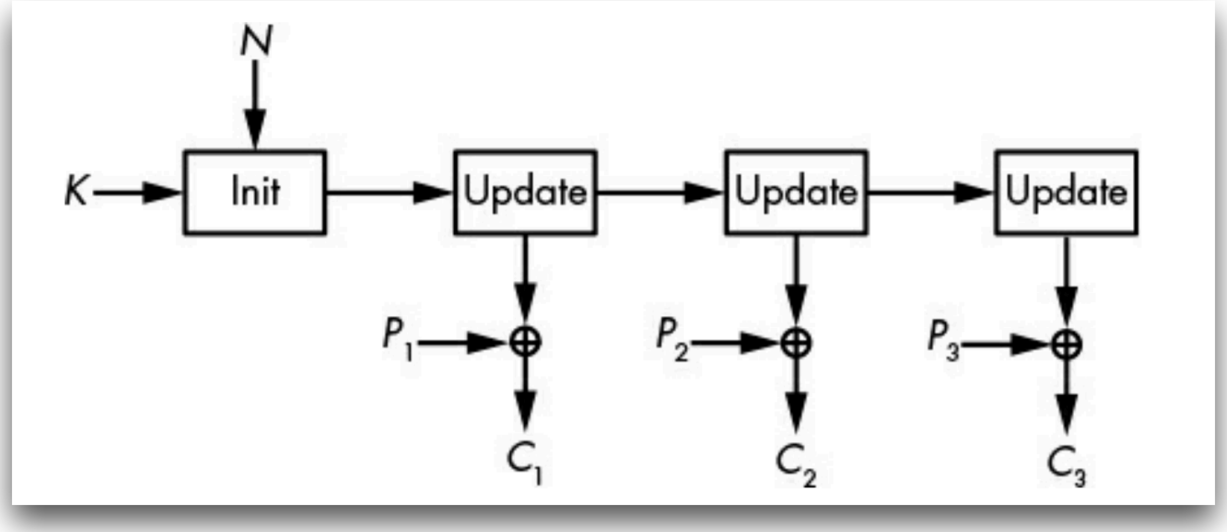
$$\begin{aligned} C_1 \oplus C_2 \oplus P_1 &= KS \oplus P1 \oplus KS \oplus C_2 \oplus P_1 \\ &= (KS \oplus KS) \oplus (P_1 \oplus P_1) \oplus P_2 \\ &= P_2 \end{aligned}$$

Stateful and Counter-Based Stream Ciphers

From a high-level perspective, there are two types of stream ciphers: **stateful** and **counter based**.

Stateful stream ciphers have a secret internal state that evolves throughout keystream generation.

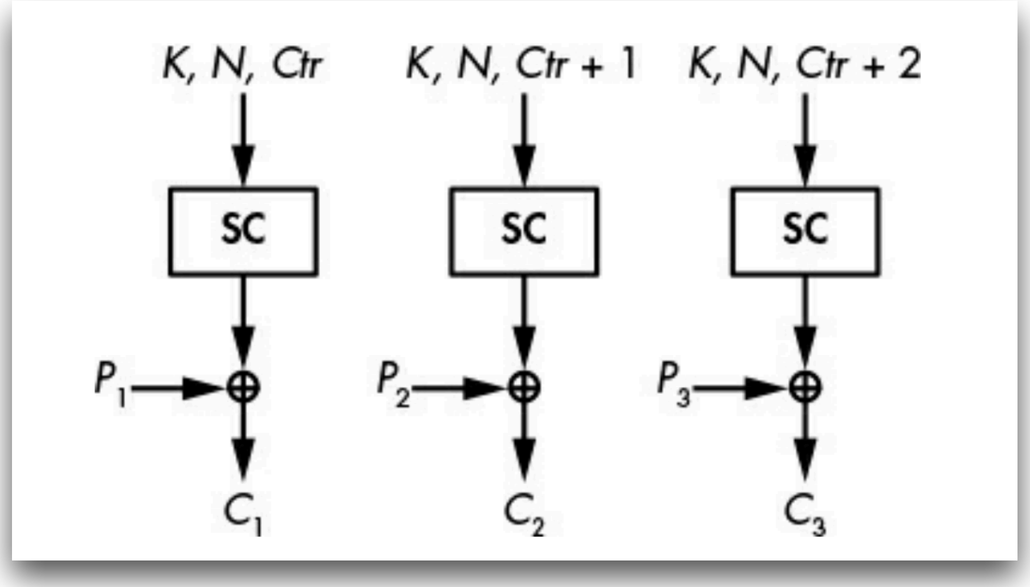




The cipher initialises the state from the key and the nonce and then calls an update function to update the state value and produce one or more keystream bits from the state.

RC4 is a famous example of a stateful cipher.

Counter-based stream ciphers produce chunks of keystream from a key, a nonce, and a counter value. No secret state is memorised during keystream generation.



The internals of the stream cipher also fall into two categories, depending on the target platform of the cipher: hardware oriented and software oriented.

Hardware-Oriented Stream Ciphers

A cipher's hardware implementation is an electronic circuit that implements the cryptographic algorithm at the bit level and that can't be used for anything else; in other words, the circuit is dedicated hardware.

Software implementations of cryptographic algorithms simply tell a microprocessor what instructions to execute in order to run the algorithm. These instructions operate on bytes or words and then call pieces of electronic circuit that implement general-purpose operations such as addition and multiplication. Software deals with bytes or words of 32 or 64 bits, whereas hardware deals with bits.

Feedback Shift Registers

An FSR is simply an array of bits equipped with an update feedback function, which we will denote as f. The FSR's state is stored in the array, or register, and each update of the FSR uses the feedback function to change the state's value and to produce one output bit.

In practice, an FSR works like this: if R0 is the initial value of the FSR, the next state, R1, is defined as R0 left-shifted by 1 bit, where the bit leaving the register is returned as output, and where the empty position is filled with f(R0).

The same rule is repeated to compute the subsequent state values R2, R3, and so on. That is, given R_t, the FSR's state at time t, the next state, R_{t+1}, is the following:

$$R_{t+1} = (R_t << 1) \mid f(R_t)$$

In this equation, | is the logical OR operator and << is the shift operator.

For example, a 4-bit FSR whose feedback function f XORs all 4 bits together. Initialize the state to the following:

$$1100$$

Now shift the bits to the left, where 1 is output and the rightmost bit is set to the following:

$$f(1100) = 1 \oplus 1 \oplus 0 \oplus 0 = 0$$

Now the state becomes this:

$$1000$$

The next update outputs 1, left-shifts the state, and sets the rightmost bit to the following:

$$f(1000) = 1 \oplus 0 \oplus 0 \oplus 0 = 1$$

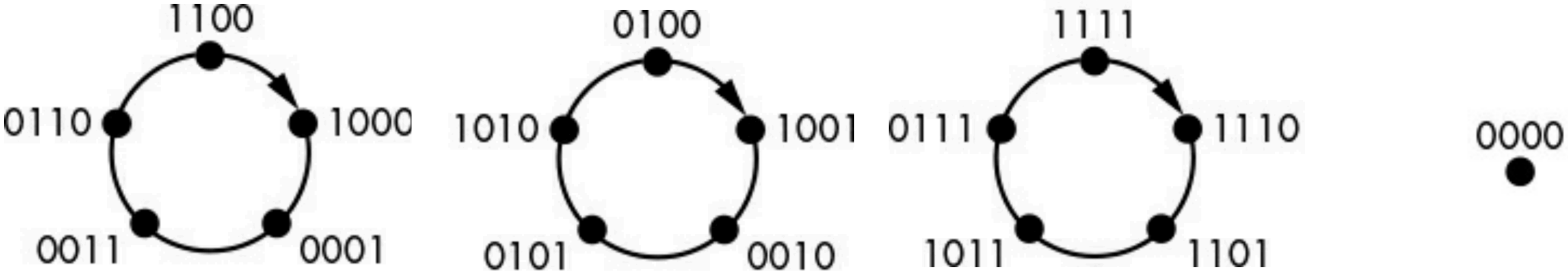
And the new state is:

$$0001$$

The next three updates return three 0 bits and give the following state values:

$$\begin{array}{c} 0011 \\ 0110 \\ 1100 \end{array}$$

Thus returning to the initial state of 1100 after five iterations, and we can see that updating the state five times from any of the values observed throughout this cycle will return us to this initial value. We say that 5 is the period of the FSR given any one of the values 1100, 1000, 0001, 0011, or 0110.



The period of an FSR, from some initial state, is the number of updates needed until the FSR enters the same state again.

Linear Feedback Shift Registers

Linear feedback shift registers (LFSRs) are FSRs with a linear feedback function.

The choice of which bits are XORed together is crucial for the period of the LFSR and thus for its cryptographic value.

How to choose the bits to be XORed so that the period is maximal? Let

$$b_n b_{n-1} \cdots b_3 b_2 b_1$$

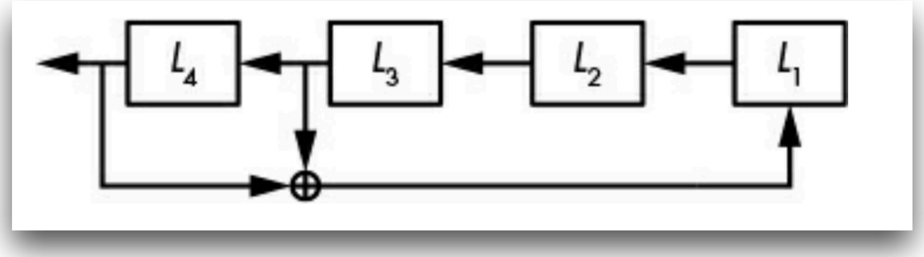
be the bits of the register, and consider the polynomial

$$1 + x + x^2 + x^3 + \cdots + x^{n-1} + x^n$$

corresponding to the positions of the selected bits.

The period is maximal iff the corresponding polynomial is **primitive**.

Considering the following LFSR



The corresponding polynomial is a primitive one:

$$1 + x^3 + x^4$$

and the respective period is maximal (15).

This is the sequence of configurations (orbit)

0001	0010	0100	1001
0011	0110	1101	1010
0101	1011	0111	1111
1110	1100	1000	0001

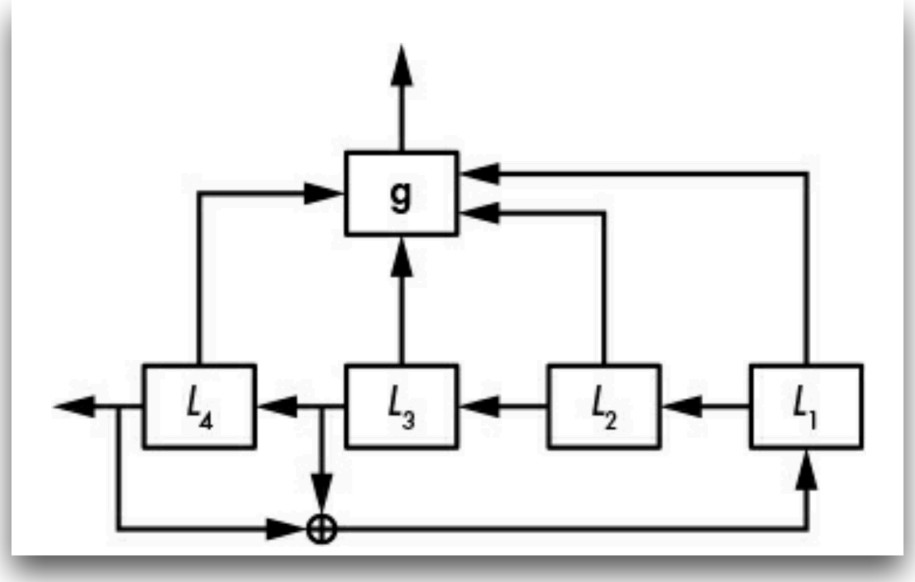
Using an LFSR as a stream cipher is insecure. If n is the LFSR's bit length, an attacker needs only n output bits to recover the LFSR's initial state, allowing them to determine all previous bits and predict all future bits.

The upshot is that LFSRs are cryptographically weak because they're linear.

To strengthen LFSRs, let's thus add a pinch of nonlinearity.

Filtered LFSRs

To mitigate the insecurity of LFSRs, you can hide their linearity by passing their output bits through a nonlinear function before returning them to produce what is called a filtered LFSR



The g function must be a nonlinear function—one that both XORs bits together and combines them with logical AND or OR operations.

Filtered LFSRs are stronger than plain LFSRs because their nonlinear function thwarts straightforward attacks. Still, more complex attacks will break the system.

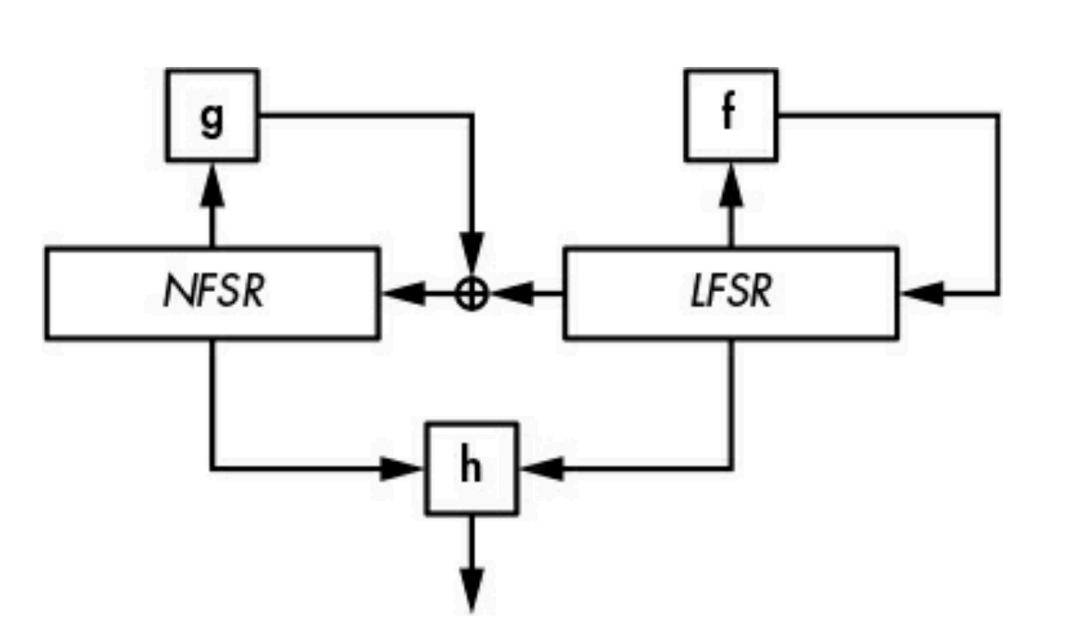
Nonlinear FSRs

Nonlinear FSRs (NFSRs) are like LFSRs but with a nonlinear feedback function instead of a linear one.

One benefit of the addition of nonlinear feedback functions is that they make NFSRs cryptographically stronger than LFSRs because the output bits depend on the initial secret state in a complex fashion, according to equations of exponential size.

One downside to NFSRs is that there's no efficient way to determine an NFSR's period, or simply to know whether its period is maximal.

Grain-128a



Grain-128a combines a 128-bit LFSR, a 128-bit NFSR, and a filter function, h. The LFSR has a maximal period of $2^{128}-1$, which ensures that the period of the whole system is at least $2^{128}-1$ to protect against potential short cycles in the NFSR. At the same time, the NFSR and the nonlinear filter function h add cryptographic strength.

Grain-128a takes a 128-bit key and a 96-bit nonce. It copies the 128 key bits into the NFSR's 128 bits and copies the 96 nonce bits into the first 96 LFSR bits, filling the 32 bits left with ones and a single zero bit at the end. The initialisation phase updates the whole system 256 times before returning the first keystream bit. During initialisation, the bit returned by the h function is thus not output as a keystream, but instead goes into the LFSR to ensure that its subsequent state depends on both the key and the nonce.

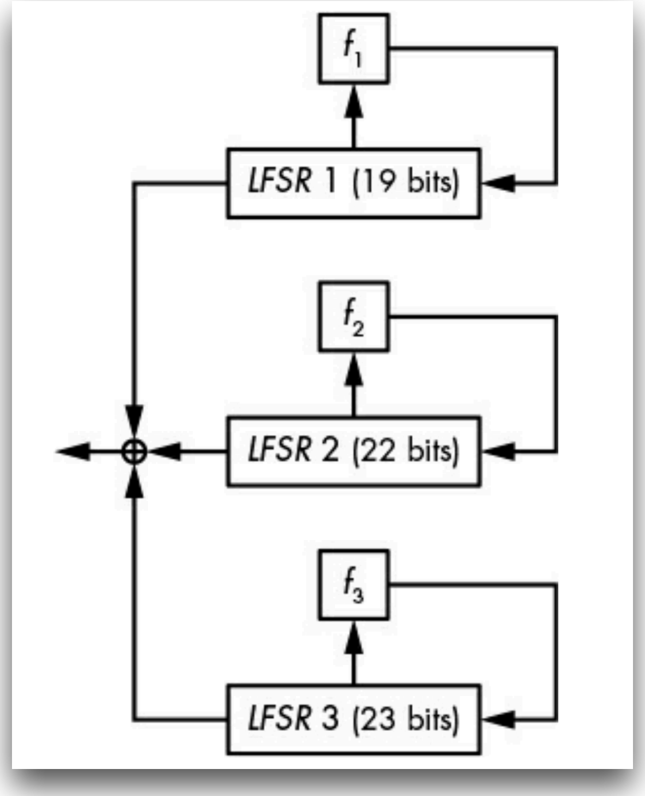
$$f(L) = L_{32} + L_{47} + L_{58} + L_{90} + L_{121} + L_{128}$$

The feedback polynomial of Grain-128a's NFSR has degree 4 and can't be approximated by a linear function because it is highly nonlinear.

The filter function h is a nonlinear function that takes 9 bits from the NFSR and 7 bits from the LFSR and combines them in a way that ensures good cryptographic properties.

A5/1

A5/1 is a stream cipher that was used to encrypt voice communications in the 2G mobile standard. Attacks appeared in the early 2000s, and A5/1 was eventually broken in a way that allows actual decryption of encrypted communications.



How could this be seen as secure with only LFSRs and no NFSR? The trick lies in A5/1's update mechanism. Instead of updating all three LFSRs at each clock cycle, the designers of A5/1 added a clocking rule that does the following:

1. Checks the value of the ninth bit of LFSR 1, the 11th bit of LFSR 2, and the 11th bit of LFSR 3, called the clocking bits. Of those three bits, either all have the same value (1 or 0) or exactly two have the same value.

2. Clocks the registers whose clocking bits are equal to the majority value, 0 or 1. Either two or three LFSRs are clocked at each update.

2G communications use A5/1 with a key of **64 bits** and a 22-bit nonce, which is changed for every new data frame.

Two kinds of attacks

Subtle Attacks: Exploit the internal linearity of A5/1 and its simple irregular clocking system

In a subtle attack called a guess-and-determine attack, an attacker guesses certain secret values of the state in order to determine others. In cryptanalysis, “guessing” means brute-forcing: for each possible value of LFSRs 1 and 2, and all possible values of LFSR 3’s clocking bit during the first 11 clocks, the attack reconstructs LFSR 3’s bits by solving equations that depend on the bits guessed. When the guess is correct, the attacker gets the right value for LFSR 3.

```
For all $2^{19}$ values of LFSR 1's initial state
 For all $2^{22}$ values of LFSR 2's initial state
  For all 2 11 values of LFSR 3's clocking bit during the first 11
clocks
   Reconstruct LFSR 3's initial state
   Test whether guess is correct;
     if yes, return; else continue
```

This attack makes at most $2^{19} \times 2^{22} \times 2^{11} = 2^{52}$ operations in the worst case, when the algorithm only succeeds at the very last test. That's 2^{12} (or about 4000) times faster than in the bruteforce search, assuming that the last two operations in the above pseudocode require about as much computation as testing a 64-bit key in a brute-force search.

The guess-and-determine attack on A5/1 can decrypt encrypted mobile communications, but it takes a couple of hours to recover the key when run on a cluster of dedicated hardware devices. In other words, it's nowhere near real-time decryption.

Brutal Attacks: The time-memory trade-off (TMTO) attack is the brutal attack on A5/1. This attack doesn't care about A5/1's internals; it cares only that its state is 64 bits long. The TMTO attack sees A5/1 as a black box that takes in a 64-bit value (the state) and spits out a 64-bit value (the first 64 keystream bits).

The idea behind the attack is to reduce the cost of a brute-force search in exchange for using lots of memory. The simplest type of TMTO is the codebook attack. In a codebook attack, you precompute a table of 2^{64} elements containing a combination of key and value pairs (key:value), and store the output value for each of the 2^{64} possible keys.

TMTO attacks reduce the memory required by a codebook attack at the price of increased computation during the online phase of the attack; the smaller the table, the more computations required to crack a key. Regardless, it will still cost about 2^{64} operations to prepare the table, but that needs to be done only once.

In 2010, researchers took about two months to generate two terabytes' worth of tables, using graphics processing units (GPUs) and running 10^5 instances of A5/1 in parallel. With the help of such large tables, calls encrypted with A5/1 could be decrypted almost in real time. Telecommunication operators have implemented workarounds to mitigate the attack, but a real solution came with the later 3G and 4G mobile telephony standards, which ditched A5/1 altogether.

Software-Oriented Stream Ciphers

Today, there is considerable interest in software stream ciphers for a few reasons. First, because many devices embed powerful CPUs and hardware has become cheaper. For example, the two stream ciphers in the mobile communications standard 4G work with 32-bit words and not bits, unlike the older A5/1.

Second, stream ciphers have gained popularity in software at the expense of block ciphers, notably after successful attacks to the later in CBC mode. In addition, stream ciphers are easier to specify and to implement than block ciphers: instead of mixing message and key bits together, stream ciphers just ingest key bits as a secret. In fact, one of the most popular stream ciphers is actually a block cipher in disguise: AES in counter mode (CTR).

RC4

Designed in 1987 by Ron Rivest of RSA Security RC4 has long been the most widely used stream cipher. RC4 is used in the first Wi-Fi encryption standard Wireless Equivalent Privacy (WEP) and in the Transport Layer Security (TLS) protocol used to establish HTTPS connections.

How RC4 Works

It simply swaps bytes. RC4's internal state is an array, S, of 256 bytes, first set to $S[0] = 0$, $S[1] = 1$, $S[2] = 2$, . . . , $S[255] = 255$, and then initialized from an n-byte K using its key scheduling algorithm (KSA).

```
j=0
S = range(256)
for i in range(256):
   j = (j + S[i] + K[i % n]) % 256
   S[i], S[j] = S[j], S[i]
```

Once this algorithm completes, array S still contains all the byte values from 0 to 255, but now in a random-looking order.

Given the initial state S, RC4 generates a keystream, KS, of the same length as the plaintext, P, in order to compute a ciphertext: $C = P \oplus KS$.

```
i=0
j=0
for b in range(m):
    i = (i + 1) % 256
    j = (j + S[i]) % 256
    S[i], S[j] = S[j], S[i]
    KS[b] = S[(S[i] + S[j]) % 256]
```

Each iteration of the for loop modifies up to 2 bytes of RC4's internal state S: S[i] and S[j] whose values are swapped. If j equals i, then S[i] isn't modified.

This looks too simple to be secure, yet it took 20 years for cryptanalysts to find exploitable flaws. Before the flaws were revealed, we only knew RC4's weaknesses in specific implementations, as in the first Wi-Fi encryption standard, WEP.

RC4 in WEP

WEP, the first generation Wi-Fi security protocol, is now completely broken due to weaknesses in the protocol's design and in RC4.

In its WEP implementation, RC4 encrypts payload data of 802.11 frames, the datagrams (or packets) that transport data over the wireless network. All payloads delivered in the same session use the same secret key of 40 or 104 bits but have what is a supposedly unique 3-byte nonce encoded in the frame header (the part of the frame that encodes metadata and comes before the actual payload).

The problem is that RC4 doesn't support a nonce, at least not in its official specification, and a stream cipher can't be used without a nonce.

The WEP designers addressed this limitation with a workaround: they included a 24-bit nonce in the wireless frame's header and prepended it to the WEP key to be used as RC4's secret key. That is, if the nonce is the bytes N[0], N[1], N[2] and the WEP key is K[0], K[1], K[2], K[3], K[4], the actual RC4 key is N[0], N[1], N[2], K[0], K[1], K[2], K[3], K[4]. The net effect is to have 40-bit secret keys yield 64-bit effective keys, and 104bit keys yield 128-bit effective keys.

The result? The advertised 128-bit WEP protocol actually offers only 104-bit security, at best.

The nonces are too small at only 24 bits . This means that if a nonce is chosen randomly for each new message, you'll have to wait about $2^{24/2} = 2^{12}$ packets, or a few megabytes' worth of traffic, until you can find two packets encrypted with the same nonce, and thus the same keystream. Even if the nonce is a counter running from 0 to $2^{24} - 1$, it will take a few gigabytes' worth of data until a rollover, when the repeated nonce can allow the attacker to decrypt packets.

Combining the nonce and key in this fashion helps recover the key. WEP's three non-secret nonce bytes let an attacker determine the value of S after three iterations of the key scheduling algorithm. Because of this, cryptanalysts found that the first keystream byte strongly depends on the first secret key byte—the fourth byte ingested by the KSA—and that this bias can be exploited to recover the secret key.

Following the appearance of the first attacks on WEP in 2001, researchers found faster attacks that required fewer ciphertexts. Today, you can even find tools such as `aircrack-ng` that implement the entire attack, from network sniffing to cryptanalysis.

WEP's insecurity is due to both weaknesses in RC4, which takes a single one-use key instead of a key and nonce (as in any decent stream cipher), and weaknesses in the WEP design itself.

RC4 in TLS

TLS is the single most important security protocol used on the internet. It is best known for underlying HTTPS connections, but it's also used to protect some virtual private network (VPN) connections, as well as email servers, mobile applications, and many others. And sadly, TLS has long supported RC4.

Unlike WEP, the TLS implementation doesn't make the same blatant mistake of tweaking the RC4 specs in order to use a public nonce. Instead, TLS just feeds RC4 a unique 128-bit session key, which means it's a bit less broken than WEP.

The weakness in TLS is due only to RC4 and its inexcusable flaws: statistical biases, or non-randomness, which we know is a total deal breaker for a stream cipher. For example, the second keystream byte produced by RC4 is zero, with a probability of 1/128, whereas it should be 1/256 ideally. Crazier still is the fact that most experts continued to trust RC4 as late as 2013, even though its statistical biases have been known since 2001.

All of the first 256 bytes were biased as well. In 2011, it was found that the probability that one of those bytes comes to zero equals $1/256 + c/256^2$, for some constant, c, taking values between 0.24 and 1.34. It's not just for the byte zero but for other byte values as well. The amazing thing about RC4 is that it fails where even many non cryptographic PRNGs succeed—namely, at producing uniformly distributed pseudorandom bytes.

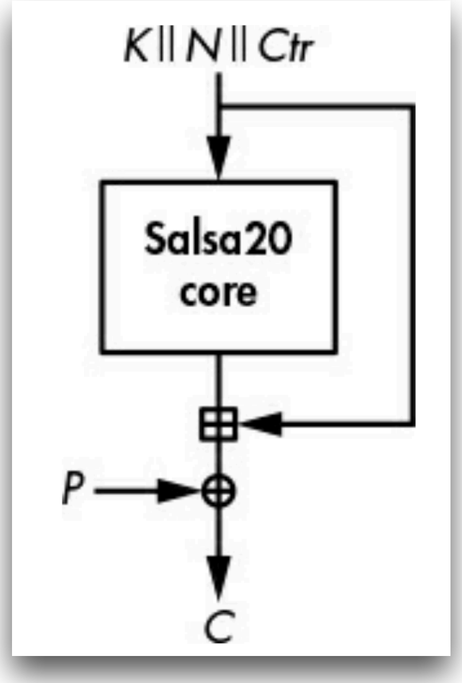
For example, say you want to decrypt the plaintext byte P_1 given many ciphertext bytes obtained by intercepting the different ciphertexts of the same message. The first four ciphertext bytes will therefore look like this:

$$\begin{aligned} C_1^2 &= P_1 \oplus KS_1^2 \\ C_1^3 &= P_1 \oplus KS_1^3 \\ C_1^4 &= P_1 \oplus KS_1^4 \end{aligned}$$

Because of RC4's bias, keystream bytes KS_1^i are more likely to be zero than any other byte value. Therefore, C_1^i bytes are more likely to be equal to P_1 than to any other value. In order to determine P_1 given the C_1^i bytes, you simply count the number of occurrences of each byte value and return the most frequent one as P_1. However, because the statistical bias is very small, you'll need millions of values to get it right with any certainty.

Salsa20

Salsa20 is a simple, software-oriented cipher optimised for modern CPUs that has been implemented in numerous protocols and libraries.



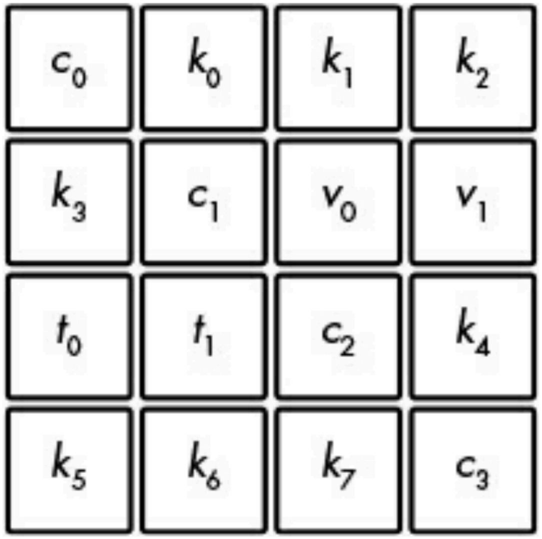
Salsa20 is a counter-based stream cipher—it generates its keystream by repeatedly processing a counter incremented for each block. The Salsa20 core algorithm transforms a 512-bit block using a key (K), a nonce (N), and a counter value (Ctr). Salsa20 then adds the result to the original value of the block to produce a keystream block. (If the algorithm were to return the core's permutation directly as an output, Salsa20 would be totally insecure, because it could be inverted. The final addition of the initial secret state K || N || Ctr makes the transform key-to-keystream-block non-invertible.)

The Quarter-Round Function

Salsa20's core permutation uses a function called quarter-round (QR) to transform four 32-bit words (a, b, c, and d):

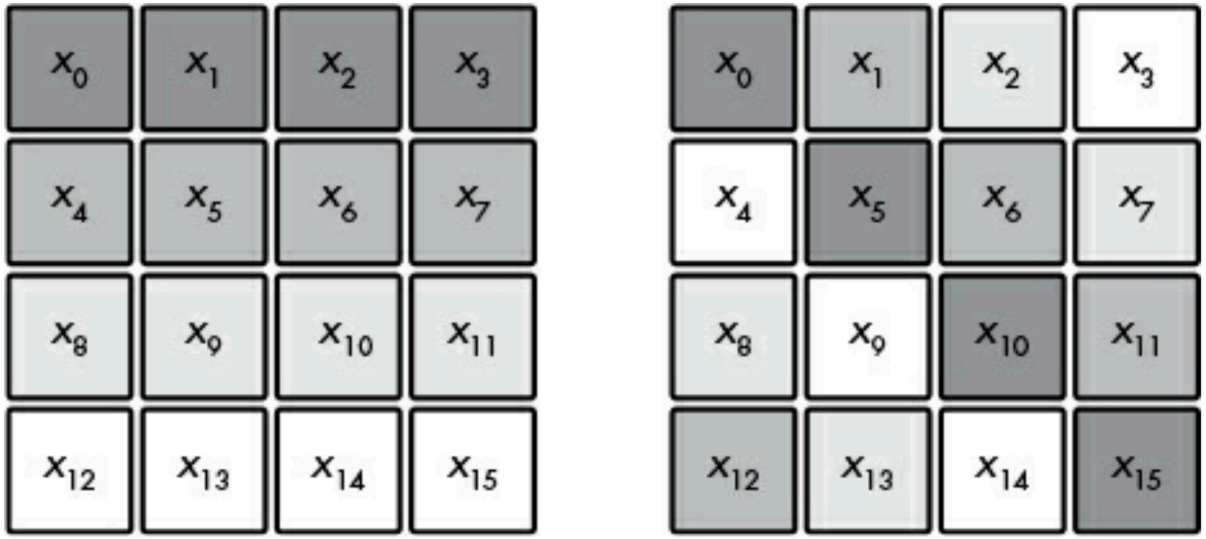
$$\begin{aligned} b &= b \oplus ((a+d) <<< 7) \\ c &= c \oplus ((b+a) <<< 9) \\ d &= d \oplus ((c+b) <<< 13) \\ a &= a \oplus ((d+c) <<< 18) \end{aligned}$$

The initial state



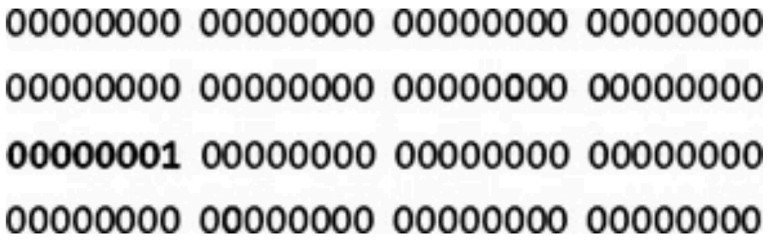
c_0	k_0	k_1	k_2
k_3	c_1	v_0	v_1
t_0	t_1	c_2	k_4
k_5	k_6	k_7	c_3

To transform the initial 512-bit state, Salsa20 first applies the QR transform to all four columns independently (known as the column-round) and then to all four rows independently (the row-round). Together this is called a double-round. Salsa20 repeats **10 double-rounds**, for **20 rounds in total**, thus the 20 in Salsa20.

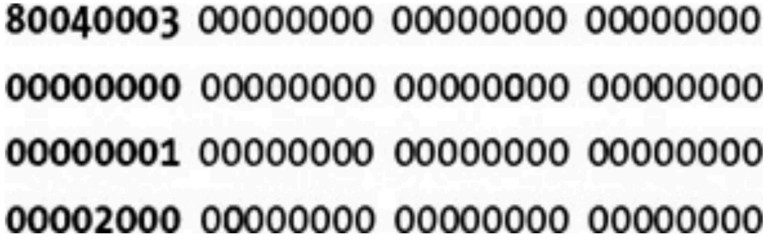


x_0	x_1	x_2	x_3
x_4	x_5	x_6	x_7
x_8	x_9	x_{10}	x_{11}
x_{12}	x_{13}	x_{14}	x_{15}

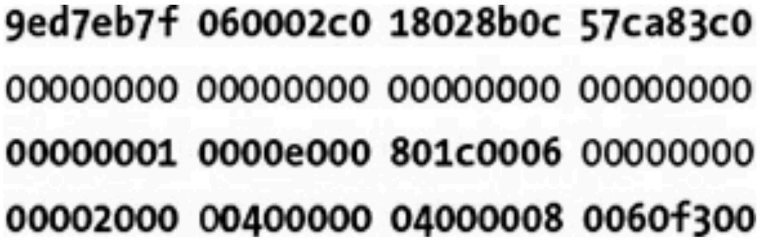
x_0	x_1	x_2	x_3
x_4	x_5	x_6	x_7
x_8	x_9	x_{10}	x_{11}
x_{12}	x_{13}	x_{14}	x_{15}



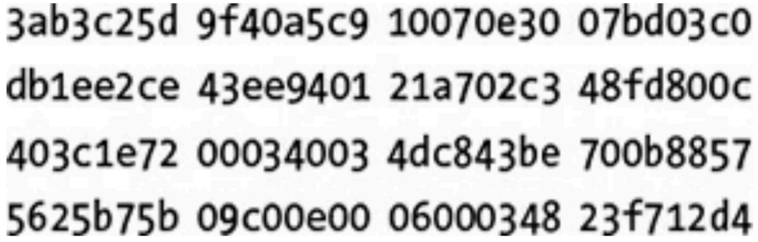
```
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000001 00000000 00000000 00000000
00000000 00000000 00000000 00000000
```



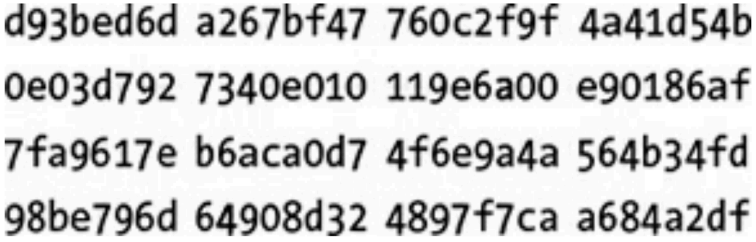
```
80040003 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000001 00000000 00000000 00000000
00002000 00000000 00000000 00000000
```



```
9ed7eb7f 060002c0 18028b0c 57ca83c0
00000000 00000000 00000000 00000000
00000001 0000e000 801c0006 00000000
00002000 00400000 04000008 0060f300
```



```
3ab3c25d 9f40a5c9 10070e30 07bd03c0
db1ee2ce 43ee9401 21a702c3 48fd800c
403c1e72 00034003 4dc843be 700b8857
5625b75b 09c00e00 06000348 23f712d4
```



```
d93bed6d a267bf47 760c2f9f 4a41d54b
0e03d792 7340e010 119e6a00 e90186af
7fa9617e b6aca0d7 4f6e9a4a 564b34fd
98be796d 64908d32 4897f7ca a684a2df
```

So after only four rounds, a single difference propagates to most of the bits in the 512-bit state. In cryptography, this is called full **diffusion**.

Breaking Salsa20 should ideally take 2^{256} operations, thanks to its use of a 256-bit key. If the key can be recovered by performing any fewer than 2^{256} operations, the cipher is in theory broken. That's exactly the case with Salsa20/8. Where, theoretically the key can be recovered at 2^{251} operations.

How Things Can Go Wrong

Alas, many things can go wrong with stream ciphers, from brittle, insecure designs to strong algorithms incorrectly implemented.

Nonce Reuse The most common failure seen with stream ciphers is an amateur mistake: it occurs when a nonce is used more than once with the same key. This produces identical keystreams, allowing you to break the encryption by XORing two ciphertexts together. The keystream then vanishes, and you're left with the XOR of the two plaintexts.

For example, older versions of Microsoft Word and Excel used a unique nonce for each document, but the nonce wasn't changed once the document was modified.