Cryptology Part IV: Bitstream Ciphers

Klaus Pommerening
Fachbereich Physik, Mathematik, Informatik
der Johannes-Gutenberg-Universität
Saarstraße 21
D-55099 Mainz

November 27, 2000—English version: October 14, 2015
last change: September 22, 2021

A bitstream cipher sequentially encrypts each single bit of a bitstring by an individual rule. The two possibilities are: leave the bit unchanged, or negate it. Leaving the bit unchanged is equivalent with (binary) adding 0, negating the bit is equivalent with adding 1. Thus every bitstream cipher may be interpreted as an XOR encryption, the key being the "difference" between ciphertext and plaintext. We distinguish between

synchronous bitstream ciphers where the key stream is generated independently of the plaintext,

asynchronous bitstream ciphers where the key stream depends on the plaintext or other context parameters.

In this part of the lecture notes we treat synchronous bitstream ciphers in a systematic way. We disregard asynchronous bitstream ciphers, and also stream ciphers over other alphabets than $\mathbb{F}_2 = \{0, 1\}$.

Key streams are usually provided by random generators, more exactly in most cases by pseudorandom generators—the main basic techniques being:

- feedback shift registers and combinations of them (as application of Boolean algebra),
- number theoretic pseudorandom generators (as application of hard number theoretic problems).

Chapter 1

Classic Pseudorandom Generators: Congruential Generators and Feedback Shift Registers

“Classic” (pseudo-) random generators are algorithms that generate “pseudorandom” numbers or bits for use in statistical applications or simulations instead of “true” random numbers or bits. For this kind of applications their statistical properties are excellent. The standard references are [2] and [3].

However the requirements of cryptology are much stronger. The methods of cryptanalysis mercilessly reveal the weaknesses of classic pseudorandom generators, and make them useless for naive direct cryptographic application.

This chapter introduces the best known classic pseudorandom generators and derives their most important properties.

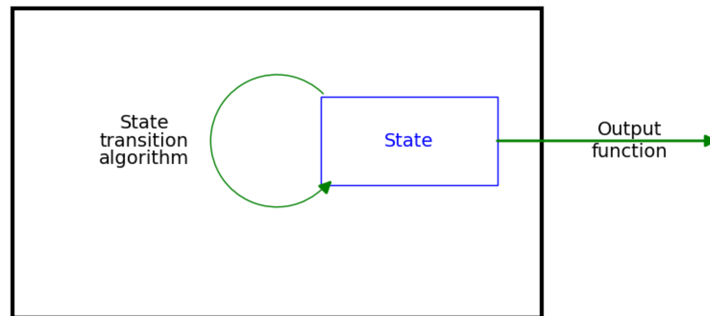


Figure 1.1: A simple model of pseudorandom generation. The state is an element of a set $\mathcal{M}$, changing after each step according to the state transition algorithm $\mathcal{M} \longrightarrow \mathcal{M}$. The output (of each step) is an element of an output alphabet Σ.

1.1 General Discussion of Bitstream Ciphers

As a first example of a bitstream cipher we encountered XOR in Part I of these lectures. SageMath code is in Appendix E.1 of Part II.

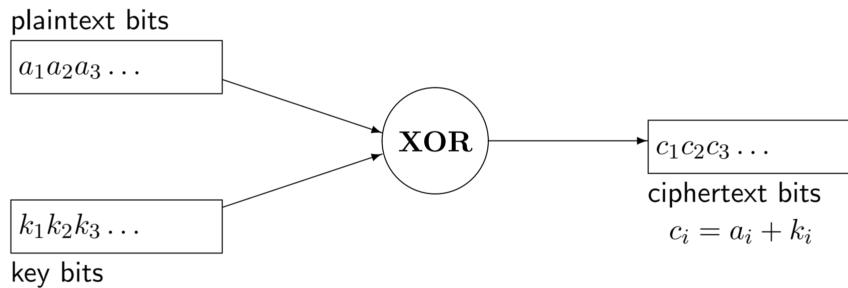


Figure 1.2: The principle of XOR encryption

In the twenties of the 20th century XOR ciphers were invented to encrypt teleprinter messages. These messages were written on five-hole punched tapes as in Figure 1.3, each column representing a five-bit-block. Another punched tape provided the key stream. VERNAM filed this procedure as a U. S. patent in 1918. He used a key tape whose ends were glued together, resulting in a periodic key stream. MAUBORGNE immediately recognized that a nonperiodic key is obligatory for security.

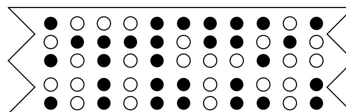


Figure 1.3: Punched tape—each column represents a five-bit character

In its strongest form, the one-time pad, XOR encryption is an example of perfect security in the sense of SHANNON, see Part I, Section 10. As algorithm A5 or E_0 XOR helps to secure mobile phones or the Bluetooth protocol for wireless data transmission. As RC4 it is part of the SSL protocol that (often) encrypts client-server communication in the World Wide Web, and of the PKZIP compression software. There are many other current applications, not all of them satisfying the expected security requirements.

> *The scope of XOR encryption ranges from simple ciphers that are trivially broken to unbreakable ciphers.*

Advantages of XOR ciphers

- Encryption and decryption are done by the same algorithm: Since $c_i = a_i + k_i$ also $a_i = c_i + k_i$. Thus decryption consists of adding key stream and ciphertext (elementwise binary).
- The method is extremely simple to understand and to implement
- ... and very fast—provided that the key stream is available. For high transfer rates one may precompute the key stream at both ends of the line.
- If the key stream is properly chosen the security is high.

Pitfalls

- XOR ciphers are vulnerable under known plaintext attacks: each correctly guessed plaintext bit reveals a key bit.
- If the attacker knows a piece of plaintext she also knows the corresponding piece of the key stream, and then is able to exchange this plaintext at will. For example she might replace "I love you" by "I hate you", or replace an amout of 1000$ by 9999$. In other words the integrity of the message is poorly protected. (To protect message integrity the sender has to implement an extended procedure.)
- XOR ciphers provide no diffusion in the sense of SHANNON's criteria since each plaintext bit affects the one corresponding plaintext bit only.
- Each reuse of a part of the key sequence (also in form of a periodic repetition) opens the door for an attack. The historical successes in breaking stream ciphers almost always used this effect, for example the attacks on encrypted teleprinters in World War II, or the project VENONA during the Cold War.

A remark on the first item, the vulnerability for attacks with known plaintext: The common ISO character set for texts has a systematic weakness. The 8-bit codes of the lower-case letters `a..z` all start with `011`, of the upper-case letters `A..Z`, with `010`. A supposed sequence of six lower-case letters (no matter which) reveals $6 \cdot 3 = 18$ key bits.

> By the way the appearance of many zeroes in the leading bits of the bytes is an important identifying feature of texts in many European languages.

In other words: We cannot prevent the attacker from getting or guessing a good portion of the plaintext. Thus the security against an attack with

known plaintext is a fundamental requirement for an XOR cipher, even more than for any other cryptographic procedure.

This being said the crucial question for a pseudorandom sequence, or for the pseudorandom generator producing it, is:

> *Is it possible to determine some more bits from a (maybe fragmented) chunk of the sequence?*

The answer for the "classic" pseudorandom generators will be YES. But we'll also learn about pseudorandom generators that—supposedly—are cryptographically secure in this sense.

1.2 Methods for Generating a Key Stream

The main naive methods for generating the key stream are:

- periodic bit sequence,
- running-text,
- "true" random sequence.

A better method uses a

- pseudorandom sequence

and leads to really useful procedures. The essential criterion is the quality of the random generator.

Note We sometimes use the term "random generator" for an algorithm that produces pseudorandom sequences (of bits or numbers). The more correct denomination is "pseudorandom generator".

Periodic Bit Sequence

A periodically repeated (longer or shorter) bit sequence serves as key. Technically this is a BELLASO cipher over the alphabet $\mathbb{F}_2$. The classical cryptanalytic methods for periodic polyalphabetic ciphers apply, such as period analysis or probable word.

For an example see XOR in Part I.

Known or probable plaintext easily breaks periodic XOR encryption.

MS Word and Periodic XOR

The following table (generated ad hoc by simple character counts) shows the frequencies of the most frequent bytes in MS Word files.

byte (hexadecimal)	**bits**	**frequency**
`00`	00000000	7–70%
`01`	00000001	0.8–17%
`20` (space)	00100000	0.8–12%
`65` (e)	01100101	1–10%
`FF`	11111111	1–10%

Note that these frequencies relate to the binary files, heavily depend on the type of the document, and may change with every software version. The variation is large, we often find unexpected peaks, and all bytes `00`–`FF` occur. But all this doesn't matter here since we observe *long chains of* `00` *bytes.*

For an MS Word file that is XOR encrypted with a periodically repeated key the ubiquity of zero bits suggests an efficient attack: Split the stream of ciphertext bits into blocks corresponding to the length of the period and add the blocks pairwise. If one of the plaintext blocks essentially consists of 0's, then the sum is readable plaintext. Why? Consider the situation

	...	block 1			...	block 2			...
plaintext:	...	a_1	...	a_s	...	0	...	0	...
key:	...	k_1	...	k_s	...	k_1	...	k_s	...
ciphertext:	...	c_1	...	c_s	...	c'_1	...	c'_s	...

where $c_i = a_i + k_i$ and $c'_i = 0 + k_i = k_i$ for $i = 1, \ldots, s$. Thus the key reveals itself in block 2, however the attacker doesn't recognize this yet. But tentatively paarwise adding all blocks she gets (amongst other things)

$$c_i + c'_i = a_i + k_i + k_i = a_i \quad \text{for } i = 1, \ldots, s,$$

that is, a plaintext block. If she realizes this (for example recognizing typical structures), then she recognizes the key $k_1, \ldots, k_s$.

Should it happen that the sum of two ciphertext blocks is zero then the ciphertext blocks are equal, and so are the corresponding plaintext blocks. The probability is high that both of them are zero. Thus the key could immediately show through. To summarize:

> *XOR encryption with a periodic key stream is quite easily broken for messages with a known structure.*

This is true also for a large period, say 512 bytes = 4096 bits, in spite of the hyperastronomically huge key space of 2^{4096} different possible keys.

Running-Text Encryption

A classical approach to generating an aperiodic key is taking an existing data stream, or file, or text, that has at least the length of the plaintext. In classical cryptography this method was called running-text encryption. We won't repeat the cryptanalytic techniques but summarize:

> *XOR encryption with running-text keys is fairly easily broken.*

True Random Sequence

The extreme choice for a key is a true random sequence of bits as key stream. Then the cipher is called **(binary) one-time pad (OTP)**. In particular no part of the key stream must be repeated at any time. The notation "pad" comes from the idea of a tear-off calendar—each sheet is destroyed after use. This cipher is unbreakable, or "perfectly secure". Shannon gave a formal proof of this, see Part I, Section 10.

Without mathematical formalism the argument is as follows: The ciphertext divulges no information about the plaintext (except the length). It could result from *any* plaintext of the same length: simply take the (binary) difference of ciphertext and alleged plaintext as key. Consider the ciphertext $c = a + k$ with plaintext a and key k, all represented by bitstreams and added bit by bit as in Figure 1.2. For an arbitrary different plaintext b the formula $c = b + k'$ likewise shows a valid encryption using $k' = b + c$ as key.

This property of the OTP could be used in a scenario of forced decryption (also known as "rubber hose cryptanalysis") to produce an innocuous plaintext, as exemplified in Figure 1.4.

If the one-time pad is perfect—why don't we use it in any case and forget of all other ciphers?

- The key management is unwieldy: Key agreement becomes a severe problem since the key is as long as the plaintext and awkwardly to memorize. Thus the communication partners have to agree on the key stream prior to transmitting the message, and store it. Agreeing on a key only just in time needs a secure communication channel—but if there was one why not use it to transmit the plaintext in clear?

- The key management is inappropriate for mass application or multi-party communication because of its complexity that grows with each additional participant.

- The problem of message integrity requires an extended solution for OTP like for any XOR cipher.

There is another, practical, problem when encrypting on a computer: How to get random sequences? "True random" bits arise from physical events like radioactive decay, or thermal noise on an optical sensor. The apparently deterministic machine "computer" can also generate true random bits, for instance by special chips that produce usable noise. Moreover many events are unpredictable, such as the exact mouse movements of the user, or arriving network packets that, although not completey random, contain random ingredients that may be extracted. On Unix systems these random bits are provided by `/dev/random`.

However these random bits, no matter how "true", are not that useful for encryption by OTP. The problem is on the side of the receiver who cannot reproduce the key. Thus the key stream must be transmitted independently.

There are other, useful, cryptographic applications of "true" random bits: Generating keys for arbitrary encryption algorithms that are unpredictable for the attacker. Many cryptographic protocols rely on "nonces" that have no meaning except for being random, for example the initialization vectors of the block cipher modes of operation, or the "challenge" for strong authentication ("challenge-response protocol").

```
Plain bits and text:
01010100 01101000 01101001 01110011 00100000 01101101   This m
01100101 01110011 01110011 01100001 01100111 01100101   essage
00100000 01101001 01110011 00100000 01101000 01100001    is ha
01111010 01100001 01110010 01100100 01101111 01110101   zardou
01110011 00101110                                       s.

Key bits:
11001000 11010110 00110011 11000000 00111011 10001110
00001000 11101111 01001001 11100101 10111100 10111001
00010010 11000110 01110011 11010111 11000100 01100000
11100110 00010111 01101010 10111011 00010101 11011000
11110000 01000010

Cipher bits:
10011100 10111110 01011010 10110011 00011011 11100011
01101101 10011100 00111010 10000100 11011011 11011100
00110010 10101111 00000000 11110111 10101100 00000001
10011100 01110110 00011000 11011111 01111010 10101101
10000011 01101100

Pseudokey bits:
11001000 11010110 00110011 11000000 00111011 10001110
00001000 11101111 01001001 11100101 10111100 10111001
00010010 11000110 01110011 11010111 11000101 01101111
11110010 00011001 01111011 10101010 00010101 11011000
11110000 01000010

Pseudodecrypted bits and text:
01010100 01101000 01101001 01110011 00100000 01101101   This m
01100101 01110011 01110011 01100001 01100111 01100101   essage
00100000 01101001 01110011 00100000 01101001 01101110    is in
01101110 01101111 01100011 01110101 01101111 01110101   nocuou
01110011 00101110                                       s.
```

Figure 1.4: XOR encryption of a hazardous message, and an alleged alternative plaintext

Pseudorandom Sequence

For XOR encryption—as approximation to the OTP—algorithmically generated bit sequences are much more practicable. But the attacker should have no means to distinguish them from true random sequences. This is the essence of the concept of pseudorandomness, and generating pseudorandom sequences is of fundamental cryptologic relevance.

> *XOR encryption with a pseudorandom key stream spoils the perfect security of the one-time pad. But if the pseudorandom sequence is cryptographically strong (Chapter 4) the attacker has no chance to exploit this fact.*

To be useful for cryptographic purposes the pseudorandom key stream must depend on parameters the attacker has no access to and that represent (parts of) the cryptographic key. Such parameters might be, see Figure 1.5 that extends the basic model of a pseudorandom generator:

- the initial value of the state,
- parameters the transition algorithm depends on.

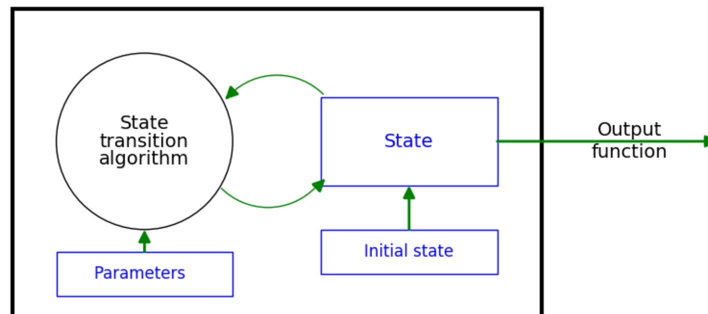


Figure 1.5: Secret parameters for a pseudorandom generator

1.3 Linear Congruential Generators

As a first important class of elementary—“classical”—pseudorandom number generators we consider one-step recursive formulas that use linear congruences. They are very fast, have long periods, and their quality is easily analyzed due to their plain structure.

This simple formula generates a sequence of pseudorandom numbers:

$$x_n = ax_{n-1} + b \bmod m. \tag{1}$$

The recursive sequence $(x_n)_{n\in\mathbb{N}}$ depends on four integer parameters:

- the **module** m where $m \geq 2$,
- the **multiplier** $a \in [0 \ldots m-1]$,
- the **increment** $b \in [0 \ldots m-1]$,
- the **initial value** $x_0 \in [0 \ldots m-1]$.

We call this recursive formula a **linear congruential generator**, in the case $b = 0$ also a **multiplicative generator**, in the case $b \neq 0$, a **mixed congruential generator**. Furthermore we call

$$s : \mathbb{Z}/m\mathbb{Z} \longrightarrow \mathbb{Z}/m\mathbb{Z}, \quad s(x) = ax + b \bmod m.$$

the **generating function** of the generator. Formula (1) then becomes

$$x_n = s(x_{n-1}).$$

Programming a linear congruential pseudorandom generator is extremely easy, even in assembler languages; for Sage see Sage sample 1.1. The algorithm works very fast. Moreover the pseudorandom numbers are statistically good *if the parameters m, a, b are suitably chosen.* In contrast the choice of the initial value is unrestricted. This freedom allows a reasonable variation of the generated pseudorandom numbers.

Use of the pseudorandom sequence as a bitstream for XOR encryption requires at least that we consider the initial value x_0, or the complete parameter set (m, a, b, x_0), as effective key, and keep it secret, cf. Figure 1.5.

Remarks and Examples

1. Since x_n may assume only m different values the sequence is periodic with a period length $\leq m$; including a possible preperiod.
2. Choosing $a = 0$ obviously doesn't make sense. Also for $a = 1$ we get a useless sequence, namely $x_0, x_0 + b, x_0 + 2b, x_0 + 3b, \ldots$, that also $\bmod\, m$ contains several regular subsequences.

Sage Example 1.1 Generating pseudorandom numbers by a linear congruential random generator

```
def lcg(m,a,b,s,n):
  x = s
  outlist = []
  for i in range (0,n):
    y = (a*x + b) % m
    outlist.append(y)
    x = y
  return outlist
```

3. For $m = 13$, $a = 6$, $b = 0$, $x_0 = 1$ we get the sequence

$$6, 10, 8, 9, 2, 12, 7, 3, 5, 4, 11, 1$$

of period length 12 that looks like a fairly random permutation of the integers 1 to 12, despite the small module.

4. Choosing the multiplier $a = 7$ instead of 6 we get a much less sympathic sequence:

$$7, 10, 5, 9, 11, 12, 6, 3, 8, 4, 2, 1.$$

5. If a and m are coprime, then the sequence is purely periodic (no preperiod). For $a \bmod m$ is invertible, hence $ac \equiv 1 \pmod{m}$ for some c. Thus always $x_{n-1} = cx_n - cb \bmod m$. If $x_{\mu+\lambda} = x_\mu$ with $\mu \geq 1$, then also $x_{\mu+\lambda-1} = x_{\mu-1}$ etc., finally $x_\lambda = x_0$.

6. By induction we immediately get

$$x_k = a^k x_0 + (1 + a + \cdots + a^{k-1}) \cdot b \bmod m \tag{2}$$

for all k—a definite warning about the poor randomness of the sequence: Formula (2) allows direct access to any element of the sequence. Note that the coefficient of b is $(a^k - 1)/(a - 1)$ where the division is $\bmod m$.

7. Let $m = 2^e$ and a be even. Then

$$x_k = (1 + a + \cdots + a^{e-1}) \cdot b \bmod m$$

for all $k \geq e$, hence, after a certain preperiod, the period has length 1. More generally common divisors of a and m reduce the period. We want to avoid this effect.

8. Let d be a divisor of m. Then the sequence $y_n = x_n \bmod d$ is the analogous congruential sequence for the module d, generated by the formula $y_n = ay_{n-1} + b \bmod d$. Hence the sequence (x_n), if considered mod d, has a period $\leq d$ that might be very short.

9. This effect is especially inconvenient in the case of a power $m = 2^e$: Then the least significant bit of x_n has a period of length at most 2, hence alternates between 0 and 1, or is constant. And the k least significant bits together have a period of at most 2^k.

10. The innocuously looking example $m = 2^{32}$, $a = 4095 = 2^{12} - 1$, $b = 12794$ exhibits an extremely bad choice of parameters: From $x_0 = 253$ we get $x_1 = 1048829$ and $x_2 = 253 = x_0$.

Preferred modules are

- $m = 2^{32}$ that exhausts the 32 bit range and moreover is computationally efficient,
- $m = 2^{31} - 1$ that is the maximum 32 bit integer, and computationally almost as efficient as a power of 2. Another advantage: This number is prime (claimed by MERSENNE in 1644, proved by EULER in 1772), and this enhances the quality of the pseudorandom sequence. More generally these arguments apply to FERMAT primes $2^k + 1$ and MERSENNE primes $2^k - 1$. The next prime of this kind is $2^{61} - 1$.

Table 1.1 shows the first 100 members of a sequence that is generated with the module $m = 2^{31} - 1 = 2147483647$, the multiplier $a = 397204094$, the increment $b = 0$, and the initial value $x_0 = 58854338$, Sage code sample 1.2. Figure 1.6 gives a visual impression of this information. We see that the sequence doesn't follow any obvious rules. However it is clear that such a visual impression is not a sufficient criterion for the quality of a pseudorandom sequence.

Sage Example 1.2 Using a linear congruential random generator

```
sage: mm = 2**31 - 1
sage: aa = 397204094
sage: bb = 0
sage: seed = 58854338
sage: seq = lcg(mm,aa,bb,seed,100); seq
```

Table 1.1: 100 members of a linear congruential sequence

1292048469	319941267	173739233	1992841820
345565651	2011011872	31344917	592918912
1827933824	1691830787	857231706	1416540893
1184833417	145217588	589958351	1776690121
1330128247	558009026	1479515830	1197548384
1627901332	929586843	19840670	1268974074
1682548197	760357405	666131673	1642023821
787305132	1314353697	167412640	1377012759
963849348	971229179	247170576	1250747100
703109068	1791051358	1978610456	1746992541
177131972	1844679385	1328403386	1811091691
1586500120	1175539757	74957396	753264023
468643347	821920620	1269873360	963348259
1698955999	139484430	30476960	1327705603
1266305157	1337811914	1808105128	640050202
37935526	1185470453	2111728842	380228478
808553600	934194915	824017077	881361640
1492263703	414709486	298916786	1883338449
771128019	558671080	1935988732	798347213
120356246	1378842534	37149011	272238278
1190345324	1006355270	1161592162	1079789655
220609946	1918105148	791775291	979447727
1160648370	779600833	1170336930	1271974642
375813045	1089009771	280197098	1144249742
1236647368	1729816359	650188387	1714906064

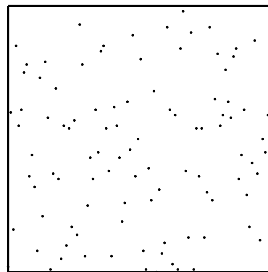


Figure 1.6: A linear congruential sequence. Horizontal axis: counter from 0 to 100, vertical axis: size of the integer from 0 to $2^{31} - 1$.

1.4 The Maximum Period Length

Under what conditions does the period of a linear congruential generator with module m attain the theoretic maximum length m? A multiplicative generator will never attain this period since the output 0 reproduces itself forever. Thus for this question we consider mixed generators with nonzero increment. As the trivial generator with generating function $s(x) = x + 1 \bmod m$ shows the period length m really occurs; on the other hand this example also shows that a period of maximum length is insufficient as a proof of quality for a random generator. Nevertheless maximum period is an important criterion, and the general result is easily stated:

Proposition 1 (Hull/Dobell **1962,** Knuth) *The linear congruential generator with generating function $s(x) = ax + b \bmod m$ has period m if and only if the following three conditions hold:*

(i) *b and m are coprime.*

(ii) *Each prime divisor p of m divides $a - 1$.*

(iii) *If 4 divides m, then 4 divides $a - 1$.*

From the first condition we conclude $b \neq 0$, hence the generator is mixed. Before giving the proof of the proposition we state and prove a lemma. (We'll use two more lemmas from Part III, Appendix A.1, that we state here without proofs.)

Lemma 1 *Let $m = m_1 m_2$ with coprime natural numbers m_1 and m_2. Let λ, λ_1, and λ_2 be the periods of the congruential generators $x_n = s(x_{n-1}) \bmod m$, $\bmod m_1$, $\bmod m_2$ with initial value x_0 in each case. Then λ is the least common multiple of λ_1 and λ_2.*

Proof. Let $x_n^{(1)}$ and $x_n^{(2)}$ be the corresponding outputs for m_1 and m_2. Then $x_n^{(i)} = x_n \bmod m_i$. Since $x_{n+\lambda} = x_n$ for all sufficiently large n we immediately see that λ is a multiple of λ_1 and λ_2. On the other hand from $m \mid t \iff m_1, m_2 \mid t$ we get

$$x_n = x_k \iff x_n^{(i)} = x_k^{(i)} \quad \text{for } i = 1 \text{ and } 2.$$

Hence λ is not larger than the least common multiple of λ_1 and λ_2. $\diamond$

The two lemmas without proofs:

Lemma 2 *Let $n = 2^e$ with $e \geq 2$.*

(i) *If a is odd, then*

$$a^{2^s} \equiv 1 \pmod{2^{s+2}} \quad \textit{for all } s \geq 1.$$

(ii) *If* $a \equiv 3 \pmod 4$, *then* $n \mid 1 + a + \cdots + a^{n/2-1}$.

Lemma 3 *Let p be prime, and e, a natural number with $p^e \geq 3$. Assume p^e is the largest power of p that divides $x - 1$. Then p^{e+1} is the largest power of p that divides $x^p - 1$.*

Proof of the proposition For both directions we may assume $m = p^e$ where p is prime by Lemma 1.

"$\Longrightarrow$": Each residue class in $[0 \ldots m-1]$ occurs exactly once during a full period. Hence we may assume $x_0 = 0$. Then

$$x_n = (1 + a + \cdots + a^{n-1}) \cdot b \bmod m \quad \text{for all } n.$$

Since x_n assumes the value 1 for some n we conclude that b is invertible $\bmod m$, or that b and m are coprime.

Let $p \mid m$. From $x_m = 0$ we now get $m \mid 1 + a + \cdots + a^{m-1}$, hence

$$p \mid m \mid a^m - 1 = (a-1)(1 + a + \cdots + a^{m-1}).$$

Fermat's little theorem gives $a^p \equiv a \pmod p$, hence

$$a^m = a^{p^e} \equiv a^{p^{e-1}} \equiv \ldots \equiv a \pmod p,$$

hence $p \mid a - 1$. This proves (ii).

Statement (iii) corresponds to the case $p = 2$ with $e \geq 2$. From (ii) we get that a is even. The assumption $a \equiv 3 \pmod 4$ would result in the contradiction $x_{m/2} = 0$ by Lemma 2. Hence $a \equiv 1 \pmod 4$.

"$\Longleftarrow$": Again we may assume $x_0 = 0$. Then

$$x_n = 0 \Longleftrightarrow m \mid 1 + a + \cdots + a^{n-1}.$$

In particular the case $a = 1$ is trivial. Hence assume $a \geq 2$. Then

$$x_n = 0 \Longleftrightarrow m \mid \frac{a^n - 1}{a - 1}.$$

We have to show:

- $m \mid \frac{a^m-1}{a-1}$—then $\lambda | m$;
- m doesn't divide $\frac{a^{m/p}-1}{a-1}$—then $\lambda \geq m$ since m is a power of p.

Let p^h be the maximum power that divides $a - 1$. By Lemma 3 we conclude

$$a^p \equiv 1 \pmod{p^{h+1}}, \quad a^p \not\equiv 1 \pmod{p^{h+2}}$$

and successively

$$a^{p^k} \equiv 1 \pmod{p^{h+k}}, \quad a^{p^k} \not\equiv 1 \pmod{p^{h+k+1}}$$

for all k. In particular $p^{h+e} \mid a^m - 1$. Since no larger power than p^h divides $a - 1$ we conclude that $m = p^e \mid \frac{a^m-1}{a-1}$. The assumption $p^e \mid \frac{a^{m/p}-1}{a-1}$ leads to the contradiction $p^{e+h} \mid a^{p^{e-1}} - 1$. $\diamondsuit$

The main application of Proposition 1 is for modules that are powers of 2:

Corollary 1 (Greenberger **1961)** *For the module $m = 2^e$ with $e \geq 2$ the period m is attained if and only if:*

(i) *b is odd.*

(ii) *$a \equiv 1 \pmod 4$.*

For prime modules Proposition 1 is useless, as the following corollary shows.

Corollary 2 *For a prime module m the period m is attained if and only if b is coprime with m and $a = 1$.*

This (lousy) result admits an immediate generalization to squarefree modules m:

Corollary 3 *For a squarefree module m the period m is attained if and only if b is coprime with m and $a = 1$.*

In summary Proposition 1 shows how to get the maximum possible period, and Corollary 1 provides a class of half-decent useful examples.

1.5 The Maximum Period of a Multiplicative Generator

A multiplicative generator $x_n = ax_{n-1} \bmod m$ never has period m since the output 0 reproduces itself. So what is the largest possible period? In the following proposition λ is the CARMICHAEL function, and this is exactly the context where it occurred for the first time.

Proposition 2 (CARMICHAEL 1910) *The maximum period of a multiplicative generator with generating function $s(x) = ax \bmod m$ is $\lambda(m)$. A sufficient condition for the period $\lambda(m)$ is:*

(i) *a is primitive* $\bmod m$.

(ii) *x_0 is relatively prime to m.*

Proof. We have $x_n = a^n x_0 \bmod m$. If $k = \operatorname{ord}_m a$ is the order of a in the multiplicative group of $\mathbb{Z}/m\mathbb{Z}$, then $x_k = x_0$. Thus the period is $\leq k \leq \lambda(m)$. Now assume a is primitive $\bmod m$, hence $1, a, \ldots, a^{\lambda(m)-1} \bmod m$ are distinct, and let x_0 be relatively prime to m. Then the x_n are distinct for $n = 0, \ldots, \lambda(m) - 1$, and the period is $\lambda(m)$. $\diamond$

Corollary 1 *Let $m = p$ prime. Then the generator has the maximum period $\lambda(p) = p - 1$ if and only if:*
(i) *a is primitive* $\bmod p$.
(ii) *$x_0 \neq 0$.*

Thus for prime modules we are in a comfortable situation: The period misses the maximum value for one-step recursive generators only by 1, and any initial value is good except 0.

Section 1.9 will broadly generalize this result.

How to find a primitive element is comprehensively discussed in Appendix A of Part III.

1.6 Feedback Shift Registers

Feedback shift registers (FSR) are a classical and popular method of generating pseudorandom sequences. The method goes back to GOLOMB in 1955 [2], but is often named after TAUSWORTHE who picked up the idea in a 1965 paper. FSRs are especially convenient for hardware implementation.

An FSR of length l is specified by a Boolean function $f\colon \mathbb{F}_2^l \longrightarrow \mathbb{F}_2$, the "feedback function". Figure 1.7 shows the mode of operation—representing f by a Boolean circuit yields an explicit construction plan. The output consists of the rightmost bit u_0, all the other bits are shifted to the right by one position, and the leftmost cell is filled by the bit $u_l = f(u_{l-1}, \ldots, u_0)$. Thus the recursive formula

$$u_n = f(u_{n-1}, \ldots, u_{n-l}) \quad \text{for } n \geq l \tag{3}$$

represents the complete sequence.

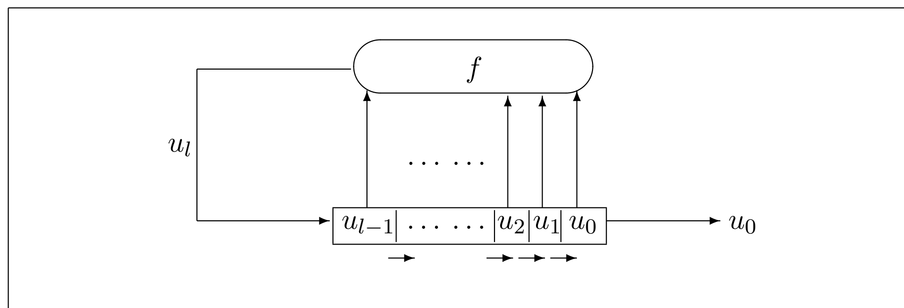


Figure 1.7: A feedback shift register (FSR)

The bits $u_0, \ldots, u_{l-1}$ form the start value. The "key expansion" transforms the short sequence $u = (u_0, \ldots, u_{l-1})$ (the effective key) of length l into a key stream $u_0, u_1, \ldots$ of arbitrary length. In a cryptographic context the bits $u_0, u_1, \ldots$ form the key stream. In other contexts it might be unnecessary to conceal the output bits, but even then hiding the initial state might make sense, starting the output sequence at u_l. Additionally in a cryptographic context treating the internal parameters, that is the feedback function f or some of its coefficients, as components of the key makes sense. Then the effective key length is larger than l.

In this respect the realization in hardware differs from a software implementation: Hardware allows using an adjustable feedback function only by complex additional circuits. Thus in the hardware case we usually assume an unchangeable feedback function, and (at least in the long run) we cannot prevent the attacker from figuring it out. In contrast a software implementation allows a comfortable change of the feedback function at any time such that it may serve as part of the key.

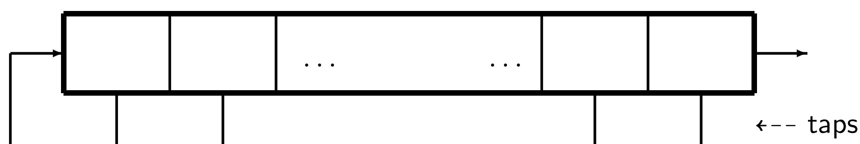


Figure 1.8: Simple graphical representation of an LFSR

For a Sage example we use the procedure `fsr` from Appendix C.1 and the construction of a Boolean function from Appendix E.3 of Part II. (Attach the modules `Bitblock.sage`, `boolF.sage`, `FSR.sage`.)

Sage Example 1.3 Generating a bitsequence by a nonlinear FSR

```
sage: f2 = BoolF([1,1,1,0,1,1,0,0,0,1,0,0,0,1,1,0])
sage: start = [0,1,1,1]
sage: seq = fsr(f2,start,20); seq
[1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
```

The simplest and best understood instances of FSRs are the linear feedback shift registers (LFSR). Their feedback functions f are linear. From Part II we know that a linear function

$$f\colon \mathbb{F}_2^l \longrightarrow \mathbb{F}_2$$

is simply a partial sum from an l-bit block:

$$f(u_{n-1},\ldots,u_{n-l}) = \sum_{j=1}^{l} a_j u_{n-j}, \qquad (4)$$

where the coefficients a_j are 0 or 1. If I is the subset of indices j with $a_j = 1$, then the iteration (3) takes the form

$$u_n = \sum_{j\in I} u_{n-j}. \qquad (5)$$

A simple graphical representation of an LFSR is shown in Figure 1.8. Here the subset I defines the contacts ("taps") that feed the respective cell contents into the feedback sum.

In SageMath we implement a special class `LFSR`, see Appendix C.1 whose use is demonstrated in the code sample 1.4.

For a good choice of the parameters, see Section 1.9, the sequence has a period of about 2^l, the number of possible different states of the register, and statistical tests are hardly able to distinguish it from a uniformly distributed true random sequence, see Section 1.10. It is remarkable that such a simple

Sage Example 1.4 Generating a bitsequence by a linear FSR

```
sage: coeff = [0,1,1,0,1,0,0,0,0,0,0,0,0,0,0,1]
sage: reg = LFSR(coeff)
sage: start = [0,1,1,0,1,0,1,1,0,0,0,1,0,0,1,1]
sage: reg.setState(start)
sage: bitlist = reg.nextBits(20); bitlist
[1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1]
```

approach generates pseudorandom sequences of fairly high quality! Of course the initial state $u = (0, \ldots, 0)$ is inappropriate. For an initial state $\neq 0$ the maximum possible period is $2^l - 1$, see Section 1.9.

For using an LFSR for bitstream encryption the secret inner parameters—the coefficients $a_1, \ldots, a_l$—as well as the initial state $u_0, \ldots, u_{l-1}$ together constitute the key. In contrast the length l of the register is assumed as known to the attacker. Beware of Section 2.3!

1.7 Multistep generators

Multistep (linear recursive) generators are a common generalization of linear congruential generators and LFSRs. A convenient framework for their treatment is a finite ring R (commutative with 1); this comprises not only the residue class rings $\mathbb{Z}/m\mathbb{Z}$ but also the finite fields including the prime fields $\mathbb{F}_p$.

An r-step linear recursive generator outputs a sequence (x_n) in R by the rule

$$x_n = a_1 x_{n-1} + \cdots + a_r x_{n-r} + b.$$

The parameters of this procedure are

- the **recursion depth** r (assume $a_r \neq 0$),
- the **coefficient tuple** $a = (a_1, \ldots, a_r) \in R^r$,
- the **increment** $b \in R$,
- a **start vector** $(x_0, \ldots, x_{r-1}) \in R^r$.

The linear recursive generator is called **homogeneous** if the increment $b = 0$, **inhomogeneous** otherwise.

Figure 1.9 visualizes the operation of a linear recursive generator in analogy with an LFSR.

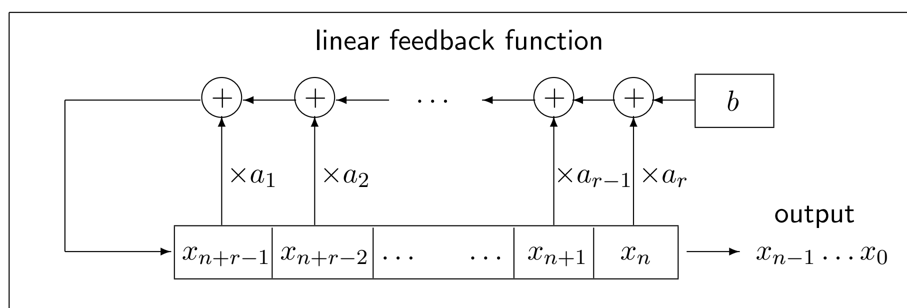


Figure 1.9: A linear recursive generator

Inhomogeneous linear recursive generators easily reduce to homogeneous ones, but only with an additional recursion step: Subtracting the two equations

$$\begin{aligned} x_{n+1} &= a_1 x_n + \cdots + a_r x_{n-r+1} + b, \\ x_n &= a_1 x_{n-1} + \cdots + a_r x_{n-r} + b, \end{aligned}$$

we get

$$x_{n+1} = (a_1 + 1)x_n + (a_2 - a_1)x_{n-1} \cdots + (-a_r)x_{n-r}.$$

Example In the case $r = 1$, $x_n = ax_{n-1} + b$, this formula becomes

$$x_n = (a+1)x_{n-1} - ax_{n-2}.$$

In the following we often neglect the inhomogeneous case.

In the homogeneous case we introduce the **state vectors** $x_{(n)} = (x_n, \ldots, x_{n+r-1})^t$ and write

$$x_{(n)} = Ax_{(n-1)} \quad \text{for } n \geq 1,$$

using the **companion matrix**

$$A = \begin{pmatrix} 0 & 1 & \ldots & 0 \\ & \ddots & \ddots & \\ & & & 1 \\ a_r & a_{r-1} & \ldots & a_1 \end{pmatrix}.$$

This suggests the next step of generalization: the **matrix generator** with parameters:

- an $r \times r$-matrix $A \in M_r(R)$,
- a start vector $x_0 \in R^r$.

The output sequence is generated by the formula

$$x_n = Ax_{n-1} \in R^r.$$

1.8 General linear generators

Even more general (and conceptually simpler) is the abstract algebraic version, the **general linear generator**. This is the setting:

- a ring R (commutative with 1),
- an R-module M,
- an R-linear map $A : M \longrightarrow M$,
- a start value $x_0 \in M$.

From this we generate a sequence $(x_n)_{n\in\mathbb{N}}$ by the formula

$$x_n = Ax_{n-1} \quad \text{for } n \geq 1. \tag{6}$$

Examples

1. For a homogeneous linear congruential generator we have

$$R = \mathbb{Z}/m\mathbb{Z}, \quad M = R \quad (r = 1), \quad A = (a).$$

2. For an inhomogeneous linear congruential generator we have

$$R = \mathbb{Z}/m\mathbb{Z}, \quad M = R^2 \quad (r = 2), \quad A = \begin{pmatrix} 0 & 1 \\ -a & a+1 \end{pmatrix}.$$

3. For an LFSR we have

$$R = \mathbb{F}_2, \quad M = \mathbb{F}_2^l \quad (r = l), \quad A = \text{the companion matrix},$$

that contains only 0's and 1's.

In the case of a finite M the recursion (6) can assume only finitely many different values, therefore (after a potential preperiod) must become periodic.

Proposition 3 *Let M be a finite R-module and $A : M \longrightarrow M$ be linear. Then the following statements are equivalent:*

(i) *All sequences generated by the corresponding general linear generator (6) are purely periodic.*

(ii) *A is bijective.*

Proof. "(i) $\Longrightarrow$ (ii)": Assume that A is not bijective. Since M is finite A is not surjective. Hence there is an $x_0 \in M - A(M)$. Then $x_0 = Ax_t$ can never occur, hence the sequence is not purely periodic.

"(ii) $\Longrightarrow$ (i)": Let A be bijective and x_0, an arbitrary start vector. Let t be the first index such that x_t assumes a value that occured before, and let s be the smallest index with $x_t = x_s$. Since $x_s = Ax_{s-1}$ and $x_t = Ax_{t-1}$ the assumption $s \geq 1$ leads to

$$x_{t-1} = A^{-1}x_t = A^{-1}x_s = x_{s-1},$$

contradicting the minimality of t. $\diamond$

Looking at the companion matrix we immediately apply this result to homogeneous multistep congruential generators, and in particular to LFSRs:

Corollary 1 *A homogeneous linear congruential generator of recursion depth r always generates purely periodic sequences if the coefficient a_r is invertible in $\mathbb{Z}/m\mathbb{Z}$.*

This is true also in the inhomogeneous case since the formula

$$x_{n-r} = a_r^{-1}(x_n - a_1x_{n-1} - \cdots - a_{r-1}x_{n-r+1} - b)$$

reproduces the sequence in the reverse direction.

Corollary 2 *An LFSR of length l generates only purely periodic sequences if the rightmost tap is set (that is, $a_l \neq 0$).*

1.9 Matrix generators over finite fields

A matrix generator over a field K is completely specified by an $r \times r$ matrix

$$A \in M_r(K)$$

(except for the choice of the start vector $x_0 \in K^r$). The objective of the present section is the characterization of the sequences with maximum period length.

In the polynomial ring $K[T]$ in one indeterminate T the set

$$\{\rho \in K[T] \mid \rho(A) = 0\}$$

is an ideal. Since $K[T]$ is a principal ring (even Euclidean) this ideal is generated by a unique monic polynomial μ. This polynomial is called the **minimal polynomial** of A. Since the matrix A is a zero of its own characteristic polynomial χ we have $\mu|\chi$. If A is invertible, then the absolute term of μ is $\neq 0$; otherwise μ would have the root 0, and A, the eigenvalue 0.

Lemma 4 *Let K be a field, $A \in GL_r(K)$, a matrix of finite order t, μ, the minimal polynomial of A, $s = \deg \mu$, $R := K[T]/\mu K[T]$, and $a \in R$, the residue class of T. Then:*

$$a^k = 1 \iff \mu | T^k - 1 \iff A^k = \mathbf{1}.$$

In particular $a \in R^\times$, t is also the order of a, and $\mu | T^t - 1$.

Proof. R is a K-algebra of dimension s. If $\mu = b_s T^s + \cdots + b_0$ (where $b_s = 1$), then

$$\mu - b_0 = T \cdot (b_s T^{s-1} + \cdots + b_1).$$

Since $b_0 \neq 0$, the residue class $T \bmod \mu$ is invertible, hence $a \in R^\times$. Since a^k is the residue class of T^k all the equivalences follow. $\diamond$

Corollary 1 *If K is a finite field with q elements, then*

$$t \leq \#R^\times \leq q^s - 1 \leq q^r - 1.$$

From now on let K be a finite field with q elements. Then also the group $GL_r(K)$ of invertible $r \times r$-matrices is finite. The vector space K^r consists of q^r vectors. We know already that every sequence from the matrix generator corresponding to $A \in GL_r(K)$ is purely periodic. One full cycle consists of the null vector $0 \in K^r$ alone. The remaining vectors in general distribute over several cycles. If s is the length of such a cycle, and x_0, the corresponding start vector, then $x_0 = x_s = A^s x_0$. Hence A^s has the eigenvalue 1, and consequently, A has as eigenvalue an s-th root of unity.

Maybe all vectors $\neq 0$ are in a single cycle of the maximum possible period length $q^r - 1$. In this case $A^s x = x$ for all vectors $x \in K^r$ if $s = q^r - 1$, but not for a smaller exponent > 0. Hence $t = q^r - 1$ is the order of A. This shows:

Corollary 2 *Let K be finite with q elements. Then:*

(i) *If the matrix generator for A and a start vector $\neq 0$ outputs a sequence of period s, then A has as eigenvalue an s-th root of unity.*

(ii) *If there is an output sequence of period length $q^r - 1$, then $t = q^r - 1$ is the order of A.*

Lemma 5 *Let K be a finite field with q elements, and $\varphi \in K[T]$ be an irreducible polynomial of degree d. Then $\varphi | T^{q^d - 1} - 1$.*

Proof. The residue class ring $R = k[T]/\varphi K[T]$ is an extension field of degree $d = \dim_K R$, hence has $h := q^d$ elements, and R contains at least one zero a of φ, namely the residue class of T. Since each $x \in R^\times$ satisfies the equation $x^{h-1} = 1$ we conclude that a is also a zero of $T^{h-1} - 1$. Hence $\mathrm{ggT}(\varphi, T^{h-1} - 1)$ is not a constant. Since φ is irreducible $\varphi | T^{h-1} - 1$. $\diamond$

Definition Let K be a finite field with q elements. A polynomial $\varphi \in K[T]$ of degree d is called **primitive** if φ is irreducible and is not a divisor of $T^k - 1$ for $1 \leq k < q^d - 1$.

Theorem 1 *Let K be a finite field with q elements and $A \in GL_r(K)$. Then the following statements are equivalent:*

(i) *The matrix generator for A generates a sequence of period $q^r - 1$.*

(ii) *The order of A is $q^r - 1$.*

(iii) *The characteristic polynomial χ of A is primitive.*

Proof. “(i) $\Longrightarrow$ (ii)”: See Corollary 2 (ii).

“(ii) $\Longrightarrow$ (iii)”: In Corollary 1 we now have $t = q^r - 1$. Hence $\#R^\times = q^s - 1$, hence R is a field, and thus μ is irreducible. Moreover $s = r$, hence $\mu = \chi$, and μ is not a divisor of $T^k - 1$ for $1 \leq k < q^r - 1$ by Lemma 4. Therefore μ is primitive.

“(iii) $\Longrightarrow$ (i)”: Since χ is irreducible, $\chi = \mu$. The residue class a of T is a zero of μ and has multiplicative order $q^r - 1$ by the definition of “primitive”. Since taking the q-th power is an automorphism of the field R that fixes K elementwise all the r powers a^{q^k} for $0 \leq k < r$ are zeroes of μ, and they are all different. Therefore they must represent all the zeroes, and they all have

multiplicative order $q^r - 1$. Hence A has no eigenvalue of lower order. By Corollary 2 (i) there is no shorter period. $\diamond$

For an LFSR take A as the companion matrix as in Section 1.7. Hence the characteric polynomial is $T^l - a_1T^{l-1} - \cdots - a_l$.

Corollary 1 *An LFSR of length l generates a sequence of the maximum possible period length $2^l - 1$ if and only if its characteristic polynomial is primitive, and the start vector is $\neq 0$.*

This result reduces the construction of LFSRs that generate maximum period sequences to the construction of primitive polynomials over the field $\mathbb{F}_2$.

The special case of dimension $r = 1$ describes a multiplicative generator $x_n = ax_{n-1}$ over the finite field K with q elements. The corresponding 1×1 matrix $A = (a)$ is the multiplication by a. Thus a is the only eigenvalue, and $\chi = T - a \in K[T]$ is the characteristic polynomial. It is linear, hence irreducible. Since

$$\chi | T^k - 1 \iff a \text{ is a zero of } T^k - 1 \iff a^k = 1,$$

χ is primitive if and only if a is a generating element of the multiplicative group $K^\times$, hence a primitive element. This proves the following slight generalization of the corollary of Proposition 2:

Corollary 2 *The multiplicative generator over K with multiplier a generates a sequence of period $q - 1$ if and only if a is primitive and the start value is $x_0 \neq 0$.*

1.10 Statistical properties of LFSRs

The study of the statistical properties of LFSR sequences of maximum period $2^l - 1$, where l is the length of the LFSR, goes back to Golomb [2].

Here are some results:

1. Each full period contains exactly 2^{l-1} ones and $2^{l-1} - 1$ zeroes.

   Proof Each of the 2^l state vectors $\in \mathbb{F}_2^l$ (except 0) occurs exactly once, corresponding to the integers in the interval $[1 \dots 2^l - 1]$. Of these integers 2^{l-1} are odd, the remaining ones are even, and their parities yield the exact output sequence of the LFSR.

2. A **run** in a sequence is a constant subsequence of maximum length.

   Example: $\dots 0111110 \dots$ is a run of ones of length 5.

   Noting that the pieces of length l of the LFSR sequence are exactly the different state vectors $\neq 0$ we immediately see that a full period contains:

   - no run of length $> l$,
   - exactly one run of 1's and no run of 0's of length l—otherwise the zero state vector would occur, or the all-1 state would occur more often than once,
   - exactly one run of 1's and exactly one run of 0's of length $l - 1$,
   - more generally exactly 2^{k-1} runs of 1's or 0's each of length $l - k$ for $1 \leq k \leq l - 1$,
   - in particular exactly 2^{l-1} runs of length 1, exactly half of them consisting of 0's or 1's.

3. For a periodic sequence $x = (x_n)_{n\in\mathbb{N}}$ in $\mathbb{F}_2$ of period s the **autocorrelation** w. r. t. the shift t is defined as

$$\begin{aligned} \kappa_x(t) &= \frac{1}{s} \cdot [\#\{n \mid x_{n+t} = x_n\} - \#\{n \mid x_{n+t} \neq x_n\}] \\ &= \frac{1}{s} \cdot \sum_{n=0}^{s-1} (-1)^{x_{n+t}+x_n} \end{aligned}$$

   (as in Part II for Boolean functions). Consider a sequence x generated by an LFSR of length l,

$$x_n = a_1 x_{n-1} + \cdots + a_l x_{n-l} \quad \text{for } n \geq l,$$

   and the sequence $y_n = x_{n+t} - x_n$ of its differences. This sequence is obviously generated by the same LFSR. If the start values $y_0, \dots, y_{l-1}$ are all 0, then the y sequence is constant $= 0$, the t-th state

vector $x_{(t)} = x_{(0)}$, hence t is a multiple of the period, and $\kappa_x(t) = 1$. Otherwise—and if x has the maximum possible period $s = 2^l - 1$—a full period of y consists of exactly 2^{l-1} ones and $2^{l-1} - 1$ zeroes by Remark 1. Thus

$$\kappa_x(t) = \begin{cases} 1, & \text{if } s|t, \\ -\frac{1}{s}, & \text{else.} \end{cases}$$

Hence the auto-correlation is uniformly small, except for shifts by a multiple of the period.

GOLOMB called these statements the three randomness postulates. They tell us that the sequence is very uniformly distributed. Therefore electrical engineers are fond of LFSR sequences of maximum period, and call them PN sequences (= pseudo-noise sequences).

Executing the Sage code sample 1.4 with the parameter 1024 instead of 20 yields the output (without parentheses and commas):

```
11001000110101100011001111000000 00111011100011100000100011101111
01001001111001011011110010111001 00010010110001100111001111010111
11000100011000001110011000010111 01101010101110110001010111011000
11110000010000100010111100011110 10100111000001111000100001011000
01010101000101111110110011011101 11001001110111110001011000100010
11100100101111110011011001010011 00001100100001100110100011100100
11101000100101110110011011001010 11011100100110111001011100000011
00100010111101111000110000010001 01110100001110011111101000100101
00111010001111000100000000110110 10000101110101110001100000010001
11011011011110111001000110101001 10001111110110101010011111100001
11101110111101011001010110001010 00000100001001100110001110100110
00010100101110100000010101100100 10010110101011111110111111011101
11001010010100010010110111111110 10100101001111110110100100010001
10111100011001111001011111010110 01110111010100100010100101101111
01100111011000000111011111010000 11011101111111110000010001000100
10010111111110101011101110111111 01110010110000010001111001100111
```

The visualization in Figure 1.10 shows that the output looks quite random, at least at first sight.

By the way the LFSR of this example generates a sequence of maximum period $2^{16} - 1 = 65535$ since its characteristic polynomial

$$T^{16} + T^{14} + T^{13} + T^{11} + 1 \in \mathbb{F}_2$$

is primitive.

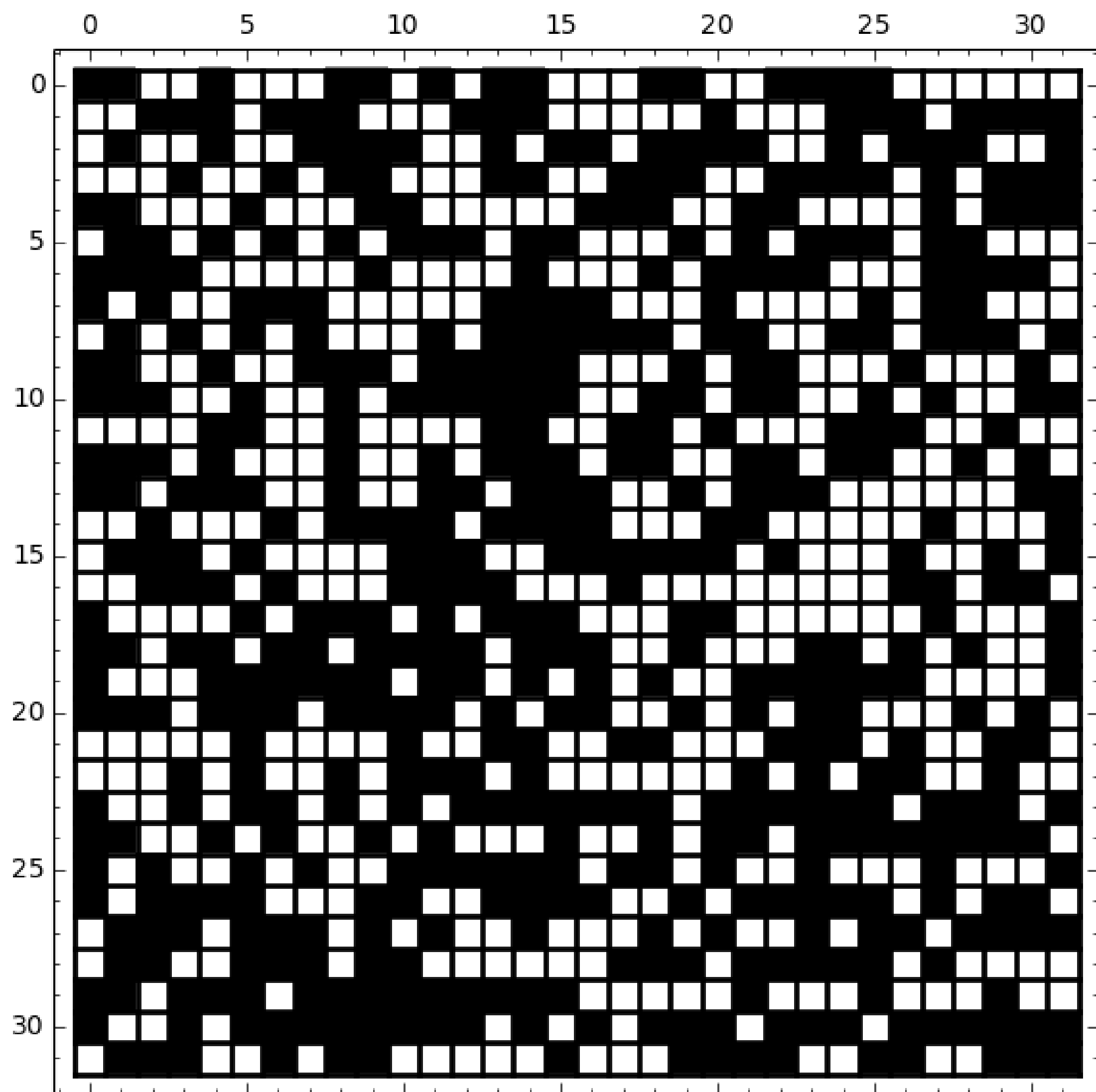


Figure 1.10: An LFSR sequence

Chapter 2

Cryptanalysis of Pseudorandom Generators

We slightly enlarge the black box model of a pseudorandom generator, cf. Figure 1.5, to distinguish between secret and public parameters:

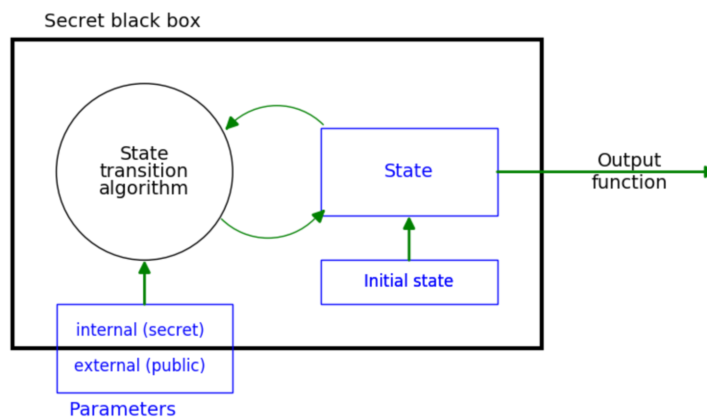


Figure 2.1: Pseudorandom generator (one element per state to be used as pseudorandom sequence

The black box hides an inner state that changes with each step by a given algorithm. This algorithm is controlled by parameters some of which are "public", but some of which are secret and serve as components of the key. The initial state (= start value) is a true random value and likewise secret. With each step the pseudorandom generator outputs a value, depending on its current inner state, until an exterior intervention stops it.

Cryptanalysis of pseudorandom generators assumes a known-plaintext attack. Thus the attacker is supposed to observe (or correctly guess) some elements of the output sequence. Her potential targets are the following

data:

- the secret internal parameters,
- the initial state,
- further elements of the output (“prediction problem”).

2.1 The General Linear Generator

Remember that a general linear generator is characterized by

- a ring R and an R-module M as external parameters,
- a linear map $A\colon M \longrightarrow M$ as internal parameter,
- a sequence of vectors $x_n \in M$ as states and output elements,
- a vector $x_0 \in M$ as initial state,
- a recursive formula $x_n = Ax_{n-1}$ for $n \geq 1$ as state transition.

Remark (the trivial case): If A is known, then from each member x_r of the output sequence we may predict all of the following members $(x_n)_{n>r}$. Therefore this case lacks cryptological relevance. Note that calculating the sequence backwards, that is x_n for $0 \leq n < r$, is uniquely possible only if A is injective. But this effect doesn't rescue the cryptologic value of the generator. For simplicity in the following we usually treat forwards prediction only, assuming that an initial chunk $x_0, \ldots, x_{k-1}$ of the output sequence is known. However we should bear in mind that also backwards "prediction" might be an issue.

Assumption for the following considerations: R and M are known, A is unknown, and an initial segment $x_0, \ldots, x_{k-1}$ is given. To avoid trivialities we assume $x_0 \neq 0$. The *prediction problem* for this scenario is: Can the attacker determine $x_k, x_{k+1}, \ldots$?

Yes she can, provided she somehow finds a linear combination

$$x_k = c_1 x_{k-1} + \cdots + c_k x_0$$

with known coefficients $c_1, \ldots, c_k$. For then

$$\begin{aligned} x_{k+1} &= Ax_k = c_1 A x_{k-1} + \cdots + c_k A x_0 \\ &= c_1 x_k + \cdots + c_k x_1 \\ &\vdots \\ x_n &= c_1 x_{n-1} + \cdots + c_k x_{n-k} \quad \text{for all } n \geq k, \end{aligned}$$

and by this formula she gets the complete remaining sequence—without determining A (!). But how to find such a linear combination?

A simple example is periodicity: $x_n = x_{n-k}$ for all $n \geq k$. Linear algebra provides a more general solution. In the present abstract framework it is natural to assume M as Noetherian (usually the "proper" generalization of a finite-dimensional vector space). Then the ascending chain of submodules

$$Rx_0 \subseteq Rx_0 + Rx_1 \subseteq \ldots \subseteq M$$

is stationary: there is an r with $x_r \in Rx_0 + \cdots + Rx_{r-1}$. And this yields the linear relation we need; of course it is useful only when we succeed with explicitly determining the involved coefficients. Note that a finite module M—that we usually consider for random generation—is trivially Noetherian.

By this consideration we have shown:

Proposition 4 (Noetherian principle for linear generators) *Let R be a ring, M, an R-module, $A\colon M \longrightarrow M$ linear, and $(x_n)_{n\in\mathbb{N}}$ a sequence in M with $x_n = Ax_{n-1}$ for $n \geq 1$. Then for $r \geq 1$ the following statements are equivalent:*

(i) $x_r \in Rx_0 + \cdots + Rx_{r-1}$.

(ii) *There exist $c_1, \ldots, c_k \in R$ such that $x_n = c_1 x_{n-1} + \cdots + c_r x_{n-k}$ for all $r \geq k$.*

If M is Noetherian, then an r with (i) *and* (ii) *exists.*

But how to explicitly determine the index k and the coefficients $c_1, \ldots, c_k$? Of course this can work only for rings R and modules M that admit explicit arithmetic operations.

In the following our main examples are: $R = K$ a finite field, or $R = \mathbb{Z}/m\mathbb{Z}$ a residue class ring of integers. In both cases we have a-priori knowledge on the number of true increments in the chain of submodules; that is, an explicit bound for r. If for example R is a field, then the number of proper steps is bounded by the vector space dimension $\dim M$. In the general case we have:

Proposition 5 (Krawczyk) *Let M be an R-module, and $0 \subset M_1 \subset \ldots \subset M_l \subseteq M$ be a properly increasing chain of submodules. Then $2^l \leq \#M$.*

This result is useful only for a finite module M. However this is the case we are mainly interested in when treating congruential generators. Then we may express it also as $l \leq \log_2(\#M)$. This is not too bad compared with the case field/vector space, both finite: $l \leq \mathrm{Dim}(M) \leq \log_2(\#M)/\log_2(\#R)$.

Proof. Let $b_i \in M_i - M_{i-1}$ for $i = 1, \ldots, l$ (where $M_0 = 0$). Then the subset

$$U = \{c_1 b_1 + \cdots + c_l b_l \mid \text{all } c_i = 0 \text{ or } 1\} \subseteq M$$

consists of 2^l distinct elements. For if two of these expressions would represent the same element, their difference would have the form

$$e_1 b_1 + \cdots + e_t b_t = 0 \quad \text{with } e_i \in \{0, \pm 1\},\ e_t \neq 0,$$

for some t with $1 \leq t \leq l$. From $e_t = \pm 1 \in R^\times$ we would derive the contradiction $b_t = -e_t^{-1}(e_1 b_1 + \cdots + e_{t-1} b_{t-1}) \in M_{t-1}$. Hence $\#M \geq \#U = 2^l$. $\diamond$

2.2 Linear Generators over Fields

In this section we consider the special case where $R = K$ is a field and M a finite dimensional vector space over K (hence a Noetherian K-module).

Then we have to find the minimal k with

$$\dim(Kx_0 + \cdots + Kx_k) = \dim(Kx_0 + \cdots + Kx_{k-1})$$

and then to find the linear combination

$$x_k = c_1 x_{k-1} + \cdots + c_k x_0.$$

This is a standard exercise in linear algebra.

For a concrete calculation we chose a fixed basis $(e_1, \ldots, e_m)$ of M. Let

$$x_n = \sum_{i=1}^{m} x_{in} e_i$$

denote the corresponding basis representation. Since $\operatorname{rank}(x_0, \ldots, x_{k-1}) = k$, there is a set $I = \{i_1, \ldots, i_k\} \subseteq \{1, \ldots, m\}$ of indices with $\#I = k$ such that the matrix

$$X = (x_{ij})_{i \in I, 0 \le j < k} = \begin{pmatrix} x_{i_1 0} & \ldots & x_{i_1 k-1} \\ \vdots & & \vdots \\ x_{i_k 0} & \ldots & x_{i_k k-1} \end{pmatrix}$$

is invertible. The coefficients c_j in the relation

$$x_k = \sum_{j=0}^{k-1} c_j x_j,$$

are not yet known, we get them by substituting

$$\sum_{i=1}^{m} x_{ik} e_i = \sum_{j=0}^{k-1} \sum_{i=1}^{m} c_j x_{ij} e_i,$$

hence by the uniqueness of basis coefficients

$$x_{ik} = \sum_{j=0}^{k-1} x_{ij} c_j \quad \text{for all } i \in I,$$

or, in matrix notation,

$$\bar{x} = (x_{ik})_{i \in I} = X \cdot c.$$

The solution for the coefficients c_j is

$$c = X^{-1} \cdot \bar{x}.$$

This proves the first two statements of the following proposition that extends Proposition 4:

Proposition 6 *Under the assumptions of Proposition 4 let $R = K$ be a field and M be finite dimensional of dimension m. Then:*

(i) *The minimal k that fulfils the statements on r in Proposition 4 is the smallest index with $\dim(Kx_0 + \cdots + Kx_k) = k$, and $k \leq m$.*

(ii) *The coefficients $c_1, \ldots, c_k$ are determined by a system of linear equations with an invertible square coefficient matrix whose entries consist of basis coefficients of $x_0, \ldots, x_{k-1}$.*

(iii) *If $k = m$, then A is uniquely determined by the basis coefficients of $x_0, \ldots, x_k$.*

Proof. (iii) Let

$$X_1 = (x_m, \ldots, x_1),\ X_0 = (x_{m-1}, \ldots, x_0) \in M_m(K).$$

Then $X_1 = AX_0$ in matrix representation for the basis $(e_1, \ldots, e_m)$ of M. Since $\operatorname{rank} X_0 = m$ the matrix X_0 is invertible, and

$$A = X_1 X_0^{-1},$$

as claimed. $\diamond$

If A is invertible, then we can determine the sequence (x_n) also in backwards direction as soon as we have a subsequence $x_t, \ldots, x_{t+m}$ of length $m + 1$ with $\operatorname{rank}(x_t, \ldots, x_{t+m-1}) = m$ at our disposition.

Example

For the special case of an r-step homogeneous linear congruential generator $x_n = a_1 x_{n-1} + \cdots + a_r x_{n-r}$ over $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$ where p is prime we use the companion matrix

$$A = \begin{pmatrix} 0 & 1 & & \\ & \ddots & \ddots & \\ & & 0 & 1 \\ a_r & \ldots & a_2 & a_1 \end{pmatrix}, \quad \operatorname{Det} A = (-1)^r a_r.$$

In this case A is invertible if and only if $a_r \neq 0$, a condition we may assume without loss of generality—otherwise the recursion depth would be $< r$.

For predicting the sequence we need at most $r + 1$ state vectors, or $2r$ elements of the sequence:

Corollary 1 *An r-step homogeneous linear congruential generator with known prime module is predictable given the $2r$ elements $x_0, \ldots, x_{2r-1}$ of the output sequence.*

Corollary 2 *An LFSR of length l is predictable from the first $2l$ output bits.*

Corollary 3 *A homogeneous linear congruential generator with known prime module is predictable from x_0, x_1, an inhomogeneous one, from x_0, x_1, x_2, x_3.*

In the Section 2.4 we'll see that even x_0, x_1, x_2 suffice.

These results knock off LFSRs as sources of key bits for cryptological applications. Keeping the length secret is useless since the attacker may easily determine it by trial and error, putting up with a slight complication of the attack.

For linear congruential generators we might hope that keeping the module m secret (and maybe not choosing a prime) might erect a serious obstacle. However we'll also put this hope at rest in Section 2.5.

2.3 Cracking an LFSR Stream XOR Encryption

Let us break down the abstract setting of Section 2.2 to an explicit procedure for cracking an XOR cipher that uses an LFSR sequence as keystream. (This section is inspired by the abstract scenario of 2.1 or 2.2 however doesn't depend on this but follows a direct approach.)

Consider a key bitstream $u_0, u_1, \ldots$ generated by an LFSR by the formula $u_n = s_1 u_{n-1} + \cdots + s_l u_{n-l}$. Assume a plaintext a is XOR encrypted using this key stream, resulting in the ciphertext c, where $c_i = a_i + u_i$ for $i = 0, 1, \ldots$ What are the prospects of an attacker who knows a chunk of the plaintext?

Well, assume she knows the first $l+1$ bits $a_0, \ldots, a_l$ of the plaintext. She immediately derives the corresponding bits $u_0, \ldots, u_l$ of the key stream, in particular the initial state of the LFSR. For the yet unknown coefficients s_i she knows a linear relation:

$$s_1 u_{l-1} + \cdots + s_l u_0 = u_l.$$

Each additional known plaintext bit yields one more relation, and having l relations, from $2l$ bits of known plaintext, the easy linear algebra over the field $\mathbb{F}_2$ finds a unique solution (in non-degenerate cases).

So assume we know the first $2l$ bits $u_0, \ldots, u_{2l-1}$ from an LFSR of length l. The state vector

$$u_{(i)} = (u_i, \ldots, u_{i+l-1}) \quad \text{for } i = 0, 1, \ldots$$

is the register content for step i (in reversed order compared with Figure 1.7). Thus the analysis focusses on the states, not directly on the output. The recursion in matrix form (for $n \geq l$) is

$$\begin{pmatrix} u_{n-l+1} \\ \vdots \\ u_{n-1} \\ u_n \end{pmatrix} = \begin{pmatrix} 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \\ s_l & s_{l-1} & \ldots & s_1 \end{pmatrix} \begin{pmatrix} u_{n-l} \\ \vdots \\ u_{n-2} \\ u_{n-1} \end{pmatrix}$$

or more parsimoniously (the indices being substituted by $m = n - l + 1$)

$$u_{(m)} = S \cdot u_{(m-1)} \quad \text{for } m \geq 1$$

where S is the companion matrix. As a further step we collect l consecutive state vectors $u_{(i)}, \ldots, u_{(i+l-1)}$ in a state matrix

$$U_{(i)} = \begin{pmatrix} u_i & u_{i+1} & \ldots & u_{i+l-1} \\ u_{i+1} & u_{i+2} & \ldots & u_{i+l} \\ \vdots & \vdots & \ddots & \vdots \\ u_{i+l-1} & u_{i+l} & \ldots & u_{2l-2} \end{pmatrix}$$

and set $U = U_{(0)}$, $V = U_{(1)}$. This yields the formula

$$V = S \cdot U$$

that expresses the unknown coefficients $s_1, \ldots, s_l$ by the known plaintext bits $u_0, \ldots, u_{2l-1}$. Most notably it allows us to write down the solution immediately—provided that the matrix U is invertible:

$$S = V \cdot U^{-1}.$$

The matrix S explicitly displays the coefficients $s_1, \ldots, s_l$. We'll discuss the invertibility later on.

Example

Assume we are given a ciphertext:

```
10011100 10100100 01010110 10100110 01011101 10101110
01100101 10000000 00111011 10000010 11011001 11010111
00110010 11111110 01010011 10000010 10101100 00010010
11000110 01010101 00001011 11010011 01111011 10110000
10011111 00100100 00001111 01010011 11111101
```

We suspect that the cipher is XOR with a key stream from an LFSR of length $l = 16$. The context suggest that the text is in German and begins with the word "Treffpunkt" (meeting point). To solve the cryptogram we need 32 bits of plaintext, that is the first four letters only, presupposed that the theory applies. This gives 32 bits of the key stream:

```
01010100 01110010 01100101 01100110 = T r e f
10011100 10100100 01010110 10100110   cipher bits
-------- -------- -------- --------
11001000 11010110 00110011 11000000   key bits
```

Sage sample 2.1 determines the coefficient matrix. Its last row tells us that all $s_i = 0$ except $s_{16} = s_5 = s_3 = s_2 = 1$.

Now we know the LFSR and the initial state, and can reconstruct the complete key stream—yes, it is the same as in Section 1.10—and write down the plaintext (that by the way begins a bit differently from our guess).

We have shown that the cofficients are uniquely determined assuming the state matrix $U = U_{(0)}$ is invertible. As a consequence in this case the LFSR is completely known, and all output bits are predictable. We have yet to discuss the case where the matrix U is singular.

If one of the first l state vectors (= rows of the matrix U) is zero, then all following state vectors are zero too, and prediction is trivial.

Thus we may assume that none of these vectors are zero, but that they are linearly dependent (reinventing the Noetherian principle for this special

Sage Example 2.1 Determining a coefficient matrix

```
sage: l = 16
sage: kbits =
      [1,1,0,0,1,0,0,0,1,1,0,1,0,1,1,0,0,0,1,1,0,0,1,1,1,1,0,0,0,0,0,0]
sage: ulist = []
sage: for i in range(0,l):
        state = kbits[i:(l+i)]
        ulist.append(state)
sage: U = matrix(GF(2),ulist)
sage: det(U)
1
sage: W = U.inverse()
sage: vlist = []
sage: for i in range(1,l+1):
        state = kbits[i:(l+i)]
        vlist.append(state)
sage: V = matrix(GF(2),vlist)
sage: S = V*W
sage: S
[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
[1 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0]
```

scenario). Then there is a smallest index $k \geq 1$ such that $u_{(k)}$ is contained in the subspace spanned by $u_{(0)}, \ldots, u_{(k-1)}$, and we find coefficients $t_1, \ldots, t_k \in \mathbb{F}_2$ such that

$$u_{(k)} = t_1 u_{(k-1)} + \cdots + t_k u_{(0)}.$$

Then also $u_{(k+1)} = S \cdot u_{(k)} = t_1 S \cdot u_{(k-1)} + \cdots + t_k S \cdot u_{(0)} = t_1 u_{(k)} + \cdots + t_k u_{(1)}$,

and by induction we get

$$u_{(n)} = t_1 u_{(n-1)} + \cdots + t_k u_{(n-k)} \quad \text{for all } n \geq k.$$

This formula predicts all the following bits.

Discussion

- For a singular state matrix this consideration yields a shorter LFSR (of length $k < l$) that generates exactly the same sequence. Then our method doesn't determine the coefficients of the original register but nevertheless correctly predicts the sequence.

- If the bits the attacker knows aren't just the first ones but $2l$ contiguous ones at a later position, then the theorem yields only the prediction of the following bits. In the main case of an invertible state matrix U the LFSR is completely known and may be run backwards to get the previous bits. For a singular state matrix we achieve the same effect using the shorter LFSR constructed above.

- The situation where $2l$ bits of the key stream are known but at non-contiguous positions is slightly more involved. We get linear relations that contain additional (unknown) intermediate bits. If m is the number of these then we get $l + m$ linear equations for $l + m$ unknown bits.

- What if the length l of the LFSR is unknown? Exhaustively trying all values $l = 1, 2, 3, \ldots$ is nasty but feasible. A better approach is provided by the Berlekamp-Massey algorithm, see Section 3.3 that is efficient also without knowledge of l. The ciphertext of the present section will be attacked again in Section 3.4.

2.4 Linear Congruential Generators with Known Module

This section uses elementary methods only and is independent of the general theory from the preceding sections of Chapter 2.

Assume the parameters a and b of the linear congruential generator $x_n = ax_{n-1} + b \bmod m$ are unknown, whereas the module m is known.

We'll show that for predicting the complete output sequence we only need 3 successive elements x_0, x_1, x_2 of the sequence, even for a composite module m. Starting with the relation

$$x_2 - x_1 \equiv a(x_1 - x_0) \pmod m$$

we immediately get (assuming for the moment that $x_1 - x_0$ and m are coprime)

$$a = \frac{x_2 - x_1}{x_1 - x_0} \bmod m,$$

where the division is $\bmod m$ (using the extended Euclidean algorithm). The increment b is given by

$$b = x_1 - ax_0 \bmod m.$$

So we found the defining formula and may predict the complete sequence.

A typical tool for this simple case was the **sequence of differences**

$$y_i = x_i - x_{i-1} \quad \text{for } i \geq 1.$$

It follows the rule

$$y_{i+1} \equiv ay_i \pmod m.$$

Note that the y_i may be negative lying between the bounds $-m < y_i < m$. Since m is known we might replace them by $y_i \bmod m$, but this was irrelevant in the example, and for an unknown m—to be considered later on—it is not an option.

Lemma 6 (on the sequence of differences) *Assume the sequence (x_i) is generated by the linear congruential generator with module m, multiplier a, and increment b. Let (y_i) be the sequence of differences, $c = \gcd(m, a)$, and $d = \gcd(m, y_1)$. Then:*

- (i) *The following statements are equivalent:*
    - (a) *The sequence (x_i) is constant.*
    - (b) $y_1 = 0$.
    - (c) $y_i = 0$ *for all* i.
- (ii) $\gcd(m, y_i) \mid \gcd(m, y_{i+1})$ *for all* i.
- (iii) $d \mid y_i$ *for all* i.

(iv) *If* $\gcd(y_1, \ldots, y_t) = 1$ *for some* $t \geq 1$*, then* $d = 1$.
(v) $c|y_i$ *for all* $i \geq 2$.
(vi) *If* $\gcd(y_2, \ldots, y_t) = 1$ *for some* $t \geq 2$*, then* $c = 1$.
(vii) $m|y_i y_{i+2} - y_{i+1}^2$ *for all* i.
(viii) *If* $\tilde{a}, \tilde{m}$ *are integers,* $\tilde{m} \geq 1$*, with* $y_i \equiv \tilde{a} y_{i-1} \pmod{\tilde{m}}$ *for* $i = 2, \ldots, r$*, then* $x_i = \tilde{a} x_{i-1} + \tilde{b} \bmod \tilde{m}$ *for all* $i = 1, \ldots, r$ *with* $\tilde{b} = x_1 - \tilde{a} x_0 \bmod \tilde{m}$.

Proof. (i) Note that $y_i = 0$ implies that all following elements are 0.

(ii) If e divides y_i and m, then it also divides $y_{i+1} = a y_i + k_i m$.

(iii) is a special case of (ii).

(iv) follows from $d|\gcd(y_1, \ldots, y_t)$, and this, from (iii).

(v) Let $m = c\tilde{m}$ and $a = c\tilde{a}$. Then $y_{i+1} = c\tilde{a} y_i + k_i c\tilde{m}$, hence $c|y_{i+1}$ for $i \geq 1$.

(vi) follows from $c|\gcd(y_2, \ldots, y_t)$ and this, from (v).

(vii) $y_i y_{i+2} - y_{i+1}^2 \equiv a^2 y_i - a^2 y_i \pmod{m}$.

(viii) by induction: For $i = 1$ the assertion is the definition of $\tilde{b}$. For $i \geq 2$ we have

$$x_i - \tilde{a} x_{i-1} - \tilde{b} \equiv x_i - \tilde{a} x_{i-1} - x_{i-1} + \tilde{a} x_{i-2} \equiv y_i - \tilde{a} y_{i-1} \equiv 0 \pmod{\tilde{m}},$$

as claimed. $\diamond$

The trivial case of a constant sequence merits no further care. However it shows that in general the parameters of a linear congruential generator are not uniquely determined by the output sequence. For the constant sequence may be generated with an arbitrary module m and an arbitrary multiplier a if only the increment is set to $b = -(a-1)x_0 \bmod m$. Even if m is fixed a is not uniquely determined, not even $a \bmod m$.

Previously we considered the case where y_1 and m are coprime, yielding $a = y_2/y_1 \bmod m$. In the general case it might happen that division $\bmod m$ is not unique. This happens if and only if m and y_1 have a non-trivial common divisor, hence $d = \gcd(m, y_1) > 1$. The **sequence of reduced differences** $\bar{y}_i = y_i/d$ (see (iii) in Lemma 6) then follows the recursive formula

$$\bar{y}_{i+1} \equiv \bar{a} \bar{y}_i \pmod{\bar{m}}$$

with the reduced module $\bar{m} = m/d$ and reduced multiplier $\bar{a} = a \bmod \bar{m}$, from which we get a unique $\bar{a} = \bar{y}_2/\bar{y}_1$. Setting $\tilde{a} = \bar{a} + k\bar{m}$ with an arbitrary integer k and $\tilde{b} = x_1 - \tilde{a} x_0 \bmod m$, from Lemma 6 (viii) we also get $x_i = \tilde{a} x_{i-1} + \tilde{b} \bmod m$ for all $i \geq 1$. This proves:

Proposition 7 *Assume the sequence* (x_i) *is generated by a linear congruential generator with known module* m*, but unknown multiplier* a *and increment* b*. Then the complete output sequence is predictable from its first three*

elements x_0, x_1, x_2. If the sequence (x_i) is not constant, then the multiplier a is uniquely determined up to a multiple of the reduced module $\bar{m}$.

Thus also in this situation we sometimes have to content ourselves with predicting the sequence without revealing the parameters used for its generation. Here is a simple concrete example: For $m = 24$, $a = 2k + 1$ with $k \in [0 \ldots 11]$, $b = 12 - 2k \bmod 24$, and initial value $x_0 = 1$ we always get the sequence $(1, 13, 1, 13, \ldots)$.

2.5 Linear Congruential Generators with Unknown Module

The attack on linear congruential generators surely becomes harder if the module m is kept secret and cannot be guessed in an obvious way. We assume that the attacker has a (short) subsequence $x_0, x_1, \ldots$ of the output sequence at her disposal.

Surprisingly it is easier to attack the multiplier first. The following proposition yields a "surrogate" value a' in a few steps. Note the Noetherian approach via the (implicit) formula $y_{t+1} \in \mathbb{Z}y_1 + \cdots + \mathbb{Z}y_t$, the principal ideal generated by the integer $\gcd(y_1, \ldots, y_t)$.

Proposition 8 (Plumstead-Boyar) *Let (y_i) be the sequence of differences of the linear congruential generator with generating function $s(x) = ax + b \bmod m$, $m \geq 2$, and initial value x_0. Let $y_1 \neq 0$ and t be the smallest index such that $e = \gcd(y_1, \ldots, y_t) \mid y_{t+1}$. Then:*

(i) $t < 1 + \log_2 m$.

(ii) *If $e = c_1 y_1 + \cdots + c_t y_t$ with $c_i \in \mathbb{Z}$ and $a' = (c_1 y_2 + \cdots + c_t y_{t+1})/e$, then $a' \in \mathbb{Z}$ and*

$$y_{i+1} \equiv a' y_i \pmod{m} \quad \textit{for all } i.$$

(iii) *If $b' = x_1 - a' x_0$, then*

$$x_i = a' x_{i-1} + b' \bmod m \quad \textit{for all } i.$$

Proof. (i) If $e_j = \gcd(y_1, \ldots, y_j)$ doesn't divide y_{j+1}, then $e_{j+1} \leq e_j/2$. Since $e_1 = |y_1| < m$ we conclude $e = e_t < m/2^{t-1}$, hence $t - 1 < \log_2 m$.

(ii) We have

$$ae = c_1 a y_1 + \cdots + c_t a y_t \equiv c_1 y_2 + \cdots + c_t y_{t+1} = a' e \pmod{m}.$$

The greatest common divisor d of m and y_1 divides e by Lemma 6, hence also $d = \gcd(m, e)$. We divide the congruence first by d:

$$a \frac{e}{d} \equiv a' \frac{e}{d} \pmod{\bar{m}}$$

with the reduced module $\bar{m} = m/d$. Since e/d and $\bar{m}$ are coprime we may divide by e/d:

$$a \equiv a' \pmod{\bar{m}}, \; a = a' + k\bar{m}.$$

Hence $y_{i+1} \equiv a y_i = a' y_i + k y_i \bar{m} \pmod{m}$. From $d \mid y_i$ follows $y_i \bar{m} \equiv 0$, hence $y_{i+1} \equiv a' y_i \pmod{m}$.

(iii) is an immediate consequence of Lemma 6 (viii). ◇

Examples

1. Let $m = 8397$, $a = 4381$, $b = 7364$ [Reeds 1977]. Generate

$$\begin{array}{lll} x_0 = 2134 & & \\ x_1 = 2160 & y_1 = 26 & e_1 = 26 \\ x_2 = 6905 & y_2 = 4745 & e_2 = 13 \\ x_3 = 3778 & y_3 = -3127 & e_3 = 1 \\ x_4 = 8295 & y_4 = 4517 & \end{array}$$

   We get $c_1 = 87542$, $c_2 = -481$, $c_3 = -1$, and $a' = 416881843$.

2. Let $m = 2^q + 1$, $a = 2^{q-1}$, $b = 2^q$, and $x_0 = 0$. By the corollary of the following Lemma 7 we have $y_i = (-1)^{i-1} \cdot 2^{q-i+1}$ for $i = 1, \ldots, q + 1$, and thus $e_i = 2^{q-i+1}$. Hence $t = q + 1$. Thus the upper bound for t in Proposition 8 is sharp, and indeed we need the $q + 3$ elements x_0 to x_{q+2} of the output sequence to determine the surrogate multiplier a'.

Lemma 7 *Let the sequence (c_i) in $\mathbb{Z}$ be defined by $c_0 = 0$, $c_i = 2^{i-1} - c_{i-1}$ for $i \geq 1$. Then*

(i) $c_i = \frac{1}{3} \cdot [2^i - (-1)^i]$ *for all i,*

(ii) $c_i - 2c_{i-1} = (-1)^{i-1}$ *for all $i \geq 1$.*

Proof. (i) follows by induction, (ii) by a direct calculation. $\diamond$

Corollary 1 *Let (x_i) be the output sequence of the linear congruential generator with module $m = 2^q + 1$, multiplier $a = 2^{q-1}$, increment $b = 2^q$, and initial value $x_0 = 0$. Let (y_i) be the sequence of differences. Then*

(i) $x_i = c_i \cdot 2^{q-i+1}$ *for $i = 0, \ldots, q + 1$,*

(ii) $y_i = (-1)^{i-1} \cdot 2^{q-i+1}$ *for $i = 1, \ldots, q + 1$.*

Proposition 8 provides a surrogate multiplier in an efficient way. Now we need a procedure for determining the module m. We close in on it by "successive correcting". In step j we determine a new surrogate module m_j and a new surrogate multiplier a_j as follows:

- In the first step set $m_1 = \infty$ and $a_1 = a'$. [Calculating mod ∞ simply means calculating with integers, and $\gcd(c, \infty) = c$ for $c \neq 0$, but $= \infty$ for $c = 0$.]
- In step j, $j \geq 2$, let $y'_j := a_{j-1} y_{j-1} \bmod m_{j-1}$. Then set $m_j = \gcd(m_{j-1}, y'_j - y_j)$ and $a_j = a_{j-1} \bmod m_j$.

Thus in iteration step j we use the current surrogate values m_{j-1} and a_{j-1} for m and a and predict a value y'_j for y_j that we compare with the real (known) value y_j. If these two numbers differ, then their difference is a multiple of m. In this case we correct the surrogate values. We always have $m \mid m_j$. The corrected values don't invalidate the former calculations since $y_i \equiv a_j y_{i-1} \pmod{m_j}$ for $i = 2, \ldots, j$, and also $y_i \equiv a_j y_{i-1} \pmod{m}$ for all $i \geq 2$. Also the true sequence (x_i) always fulfils $x_i \equiv a_j x_{i-1} + b_j \pmod{m_j}$ for $i = 1, \ldots, j$ with $b_j = x_1 - a_j x_0$ by Lemma 6 (viii).

In Example 1 above we have

	$m_1 = \infty$	$a_1 = 416881843$
$y'_2 = 10838927918$	$m_2 = 10838923173$	$a_2 = 416881843$
$y'_3 = 5420327549$	$m_3 = 8397$	$a_3 = 4381$

The calculation for m_3 is

$$\gcd(10838923173, 5420330676) = 8397.$$

Since $m_3 \leq 2x_2$ we conclude that necessarily $m = m_3$, $a = a_3$, and $b = x_1 - ax_0 \bmod m = 7364$. Thus we found the true values after two correction steps, and we didn't need any further elements of the output sequence than the five we used for determining a'. Note the large intermediate results that suggest that in general the procedure relies on multi-precision integer arithmetic.

Does the procedure always terminate? At the latest when we reach the period of the sequence, that is after at most m steps, the complete sequence is predictable. However this bound is practically useless. Unfortunately it is tight: For arbitrary m let $a = 1$, $b = 1$, and $x_0 = 0$. Then $x_i = i$ and $y_i = 1$ for $i = 0, \ldots, m-1$. The initial value for the surrogate multiplier is $a' = 1$. The first false prediction is $y'_m = 1$ instead of the correct value $y_m = 1 - m$. The end is reached only after evaluating x_m. Although this worst case is easily recognized and might be treated separately it nevertheless hints at the difficulty of finding good general results. And indeed we don't know of any.

From a slightly different point of view we count the number of necessary correction steps where the surrogate module changes. For if $m_j \neq m_{j-1}$, then $m_j \leq m_{j-1}/2$. Let $m^{(0)} = \infty > m^{(1)} > \ldots$ be the sequence of *distinct* surrogate modules. Then

$$m^{(1)} = m_{j_1} = |y'_{j_1} - y_{j_1}| < a'|y_{j_1 - 1}| + m < m(a' + 1),$$

$$m \leq m^{(j)} < \frac{m(a'+1)}{2^{j-1}},$$

hence always $j < 1 + \log_2(a'+1)$. This gives an upper bound for the number of necessary corrections. Joan Plumstead-Boyar described a variant of the

algorithm that results in a potentially smaller value of a', and eventually in the upper bound $2 + \log_2 m$ for the number of correction steps. However in general the algorithm doesn't involve that many corrections making this bound obsolete as a terminating criterion.

It seems that the search for theoretical results is a worthwile task. Could we exclude a (maybe small) class of (maybe bad anyway) linear congruential generators such that the majority of the remaining (interesting) generators obey a practically useful terminating criterion? I would expect such a result. Is there a way to control the distribution of the number of steps? Or at least the mean value?

Anyway the known results suffice to disqualify linear congruential generators for direct cryptographic application.

For an implementation of this algorithm in C see `https://www.staff.uni-mainz.de/pommeren/Cryptology/Bitstream/2_Analysis/LCGcrack.html`.

2.6 A General Prediction Method

The method of BOYAR (née PLUMSTEAD) admits a broad generalization by the **BK algorithm** (named after BOYAR and KRAWCZYK): It applies to recursive formulas that have an expression in terms of (unknown) linear combinations of known functions. A suitable language for its description is commutative algebra, that is, rings and modules.

So let R be a commutative ring (with $1 \neq 0$), and X, Z be R-modules. Let

$$\Phi^{(i)} : X^i \longrightarrow Z \quad \text{for } i \geq h$$

be a family of maps that we consider as known, and

$$\alpha : Z \longrightarrow X$$

be a linear map considered as secret. From these data we generate a sequence $(x_n)_{n\in\mathbb{N}}$ in X by the following algorithm, see Figure 2.2:

- Set $x_0, \ldots, x_{h-1} \in X$ as initial values.
- After generating $x_0, \ldots, x_{n-1}$ for some $n \geq h$ let

$$\begin{aligned} z_n &:= \Phi^{(n)}(x_0, \ldots, x_{n-1}) \in Z, \\ x_n &:= \alpha(z_n) \in X. \end{aligned}$$

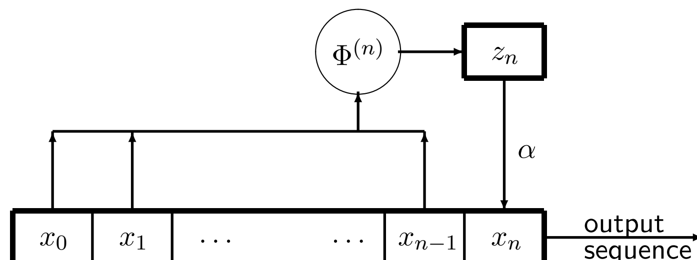


Figure 2.2: A very general generator

Here, in greater generality as before, we allow that each element of the sequence depends on *all* of its predecessors, that is, on the complete "past". A reasonable use for pseudorandom generation of course supposes that the $\Phi^{(i)}$ are efficiently computable. In the sample case $R = \mathbb{Z}/m\mathbb{Z}$, $X = R^k$, the cost should grow at most polynomially with $\log(m)$, h, and k.

Examples

1. The linear congruential generator: $R = \mathbb{Z}/m\mathbb{Z} = X$, $Z = R^2$, $h = 1$, $x_n = ax_{n-1} + b$,

$$\Phi^{(i)}(x_0, \ldots, x_{i-1}) = \begin{pmatrix} x_{i-1} \\ 1 \end{pmatrix},$$

$$\alpha \begin{pmatrix} s \\ t \end{pmatrix} = as + bt.$$

2. The linear-inversive congruential generator: R, X, Z, h, α as above, $x_n = ax_{n-1}^{-1} + b$,

$$\Phi^{(i)}(x_0, \ldots, x_{i-1}) = \begin{pmatrix} x_{i-1}^{-1} \bmod m \\ 1 \end{pmatrix}.$$

   (Set the first component to 0 if x_{i-1} is not invertible mod m.)

3. Congruential generators of higher degree: $R = \mathbb{Z}/m\mathbb{Z} = X$, $Z = R^{d+1}$, $h = 1$, $x_n = a_d x_{n-1}^d + \cdots + a_0$,

$$\Phi^{(i)}(x_0, \ldots, x_{i-1}) = \begin{pmatrix} x_{i-1}^d \\ \vdots \\ x_{i-1} \\ 1 \end{pmatrix},$$

$$\alpha \begin{pmatrix} t_0 \\ \vdots \\ t_d \end{pmatrix} = a_d t_0 + \cdots + a_0 t_d.$$

4. Arbitrary congruential generators: $R = \mathbb{Z}/m\mathbb{Z}$, $x_n = s(x_{n-1})$, $h = 1$. If m is prime, then each function $s \colon R \longrightarrow R$ has an expression as a polynomial of degree $< m$, as in Example 3. For a more general module m we may use the basis $\{e_0, \ldots, e_{m-1}\}$ with $e_i(j) = \delta_{ij}$ of R^R. The basis representation is $s = \sum_{i=0}^{m-1} s(i)e_i$. Thus we set $X = R$, $Z = R^m$, and

$$\Phi^{(i)}(x_0, \ldots, x_{i-1}) = \begin{pmatrix} e_0(x_{i-1}) \\ \vdots \\ e_{m-1}(x_{i-1}) \end{pmatrix},$$

$$\alpha \begin{pmatrix} t_0 \\ \vdots \\ t_{m-1} \end{pmatrix} = s(0)t_0 + \cdots + s(m-1)t_{m-1}.$$

5. For multistep congruential generators set h equal the recursion depth.

6. For nonlinear feedback shift registers see the next section 2.7.

For cryptanalysis we assume that the $\Phi^{(i)}$ are known, but α is unknown. (Later on, in the case $R = \mathbb{Z}/m\mathbb{Z}$, we'll also treat m as unknown.) The question is: Given an initial segment $x_0, \ldots, x_{n-1}$ ($n \geq h$) of the output sequence, is there a method to predict the next element x_n?

To this end we consider the ascending chain $Z_h \subseteq Z_{h+1} \subseteq \ldots \subseteq Z$ of submodules with

$$Z_n = Rz_h + \cdots + Rz_n.$$

If $Z_n = Z_{n-1}$, then $z_n = t_h z_h + \cdots + t_{n-1} z_{n-1}$ with $t_h, \ldots, t_{n-1} \in R$, and applying α we get the formula

$$x_n = t_h x_h + \cdots + t_{n-1} x_{n-1}$$

that predicts x_n from $x_0, \ldots, x_{n-1}$ without using knowledge of α.

If Z is a Noetherian R-module, then we encounter a stationary situation after finitely many steps: $Z_n = Z_l$ for $n \geq l$. Beginning with this index the complete sequence x_n is predictable by the following "algorithm":

1. Calculate $z_n = \Phi^{(n)}(x_0, \ldots, x_{n-1})$.
2. Find a linear combination $z_n = t_h z_h + \cdots + t_{n-1} z_{n-1}$.
3. Set $x_n = t_h x_h + \cdots + t_{n-1} x_{n-1}$.

The Noetherian principle allows the prediction by a linear relation (that however might change from step to step).

To transform the "algorithm" into a true algorithm we need a procedure that explicitly finds a linear combination in step 2, solving a system of linear equations in Z.

For our standard example of a congruential generator with module $m = 8397$ (here assumed to be known), $x_0 = 2134$, $x_1 = 2160$, $x_2 = 6905$, we calculate

$$z_1 = \begin{pmatrix} 2134 \\ 1 \end{pmatrix}, \quad z_2 = \begin{pmatrix} 2160 \\ 1 \end{pmatrix}, \quad z_3 = \begin{pmatrix} 6905 \\ 1 \end{pmatrix}.$$

Trying to write z_3 as a linear combination $t_1 z_1 + t_2 z_2$ we get the system

$$(1) \qquad \begin{aligned} 2134 t_1 + 2160 t_2 &= 6905 \\ t_1 + t_2 &= 1 \end{aligned}$$

of linear equations in $R = \mathbb{Z}/m\mathbb{Z}$. By elimination we find

$$26 t_1 = -4745 = 3652.$$

The inverse of 26 mod 8397 is 323, and thus we get $t_1 = 4016$, $t_2 = 4382$. This result correctly predicts $x_3 = 3778$.

Proceeding in this way we correctly predict the complete output sequence. The reason is that $Z_2 = Z$:

$$z_2 - z_1 = \begin{pmatrix} 26 \\ 0 \end{pmatrix}, \quad e_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \in Z_2, \quad e_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = z_1 - 2134 \cdot e_1 \in Z_2.$$

This example contains a partial answer to the question of when the chain of submodules Z_n becomes stationary: At least when $Z_l = Z$. But in a more general case this might never happen. Note also that from $Z_l = Z_{l+1}$ we can't conclude that the chain is stationary at Z_l—later on it could ascend again. For a bound on the number of proper increments see Proposition 5.

In each single loop of the prediction algorithm there are two possible alternative events:

- $z_n \notin Z_{n-1}$. Then predicting x_n is impossible, and Z_{n-1} properly extends to $Z_n = Z_{n-1} + Rz_n$.
- $z_n \in Z_{n-1}$. Then the algorithm correctly predicts x_n.

By Proposition 5 the first of these two events may happen at most $\log_2(\#Z)$ times (or $\operatorname{Dim} Z$ times if R is a field). For each of these events we need access to the next element x_n of the output sequence to get ahead. On first sight this looks disappointing, but some thought brings to mind that it is a realistic situtation for cryptanalysis: In the process of breaking a cipher the cryptanalyst works with a supposed key until she gets nonsense "plaintext". Then she tries to guess the following plaintext characters by context knowledge, corrects the supposed key and goes on with deciphering. Remember that we already encountered this effect in the last section. And note that the present algorithm is fairly simple but contents itself with predicting elements instead of determining the unknown parameters of the random generator.

2.7 Nonlinear Feedback Shift Registers

As another example of the general prediction method we consider arbitrary, not necessarily linear, feedback shift registers as illustrated in Figure 2.3.

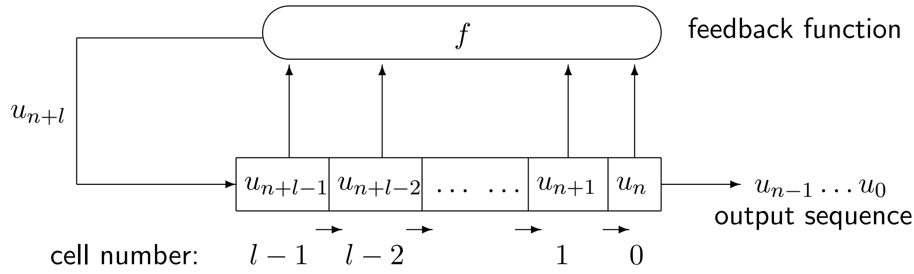


Figure 2.3: A feedback shift register (FSR) of length l

Here the feedback function is an arbitrary Boolean function $f\colon \mathbb{F}_2^l \longrightarrow \mathbb{F}_2$ whose algebraic normal form is a polynomial

$$f(y_1,\dots,y_l) = \sum_{I\subseteq\{1,\dots,l\}} a_I y^I \qquad \text{with } y^I = \prod_{j\in I} y_j.$$

We want to apply the prediction method with $R = X = \mathbb{F}_2$, $h = l$, $Z = \mathbb{F}_2^{2^l}$. For $i \geq l$

$$\Phi^{(i)}\colon \mathbb{F}_2^i \longrightarrow Z$$

is given by

$$z_i := \Phi^{(i)}(x_1,\dots,x_i) = (y^I)_{I\subseteq\{1,\dots,l\}} \qquad \text{with } y = (x_{i-l+1},\dots,x_i).$$

And finally we set

$$\alpha\colon Z \longrightarrow X, \quad \alpha((t_I)_{I\subseteq\{1,\dots,l\}}) = \sum a_I t_I.$$

First we treat two concrete examples:

Examples

1. $l = 2$, $f = T_1T_2 + T_2$. From the initial values $u_0 = 1$, $u_1 = 0$ we generate the sequence (manually or by Sage example 2.2)

$$u_0 = 1,\ u_1 = 0,\ u_2 = 1,\ u_3 = 0,\ \dots$$

(that evidently has period 2). We have

$$Z = \mathbb{F}_2^4, \qquad z_n = \begin{pmatrix} u_{n-1}u_{n-2} \\ u_{n-1} \\ u_{n-2} \\ 1 \end{pmatrix},$$

$$z_2 = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}, \quad z_3 = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}, \quad z_4 = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} = z_2, \quad \ldots$$

From this the cryptanalyst recognizes the linear recursion

$$z_n = z_{n-2} = 0 \cdot z_{n-1} + 1 \cdot z_{n-2} \quad \text{for } n \geq 4.$$

She even recognizes the period, and correctly predicts

$$u_n = 0 \cdot u_{n-1} + 1 \cdot u_{n-2} = u_{n-2} \quad \text{for } n \geq 4.$$

Note that the very same sequence can be generated by a *linear* FSR of length 2. The analysis used the elements u_0, u_1, u_2, u_3.

2. $l = 3$, $f = T_1T_3 + T_2$. From the initial values $u_0 = 0$, $u_1 = 1$, $u_2 = 1$ we generate the elements (manually or by Sage example 2.3)

$$u_3 = 1,\ u_4 = 0,\ u_5 = 1,\ u_6 = 1,\ u_7 = 1,\ u_8 = 0,\ u_9 = 1,\ \ldots$$

of the output sequence. We have

$$Z = \mathbb{F}_2^8, \qquad z_n = \begin{pmatrix} u_{n-1}u_{n-2}u_{n-3} \\ u_{n-1}u_{n-2} \\ u_{n-1}u_{n-3} \\ u_{n-2}u_{n-3} \\ u_{n-1} \\ u_{n-2} \\ u_{n-3} \\ 1 \end{pmatrix},$$

$$z_3 = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \end{pmatrix}, \quad z_4 = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}, \quad z_5 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{pmatrix}, \quad z_6 = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{pmatrix}, \quad z_7 = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \end{pmatrix} = z_3,$$

and so on. Hence the supposed linear recursion is

$$z_n = z_{n-4} \quad \text{for } n \geq 4,$$

again it reflects the periodicity. We get the correct prediction formula

$$u_n = u_{n-4} \quad \text{for } n \geq 4.$$

We needed the elements from u_0 to u_6; and again we found an "equivalent" LFSR, this time of length 4.

Sage Example 2.2 $f_1 = T_1T_2 + T_2$—monomials with exponent pairs `[1,1]` $\hat{=}\,3$ and `[0,1]` $\hat{=}\,1$, hence ANF bitblock `[0,1,0,1]`

```
f1 = BoolF([0,1,0,1],method="ANF")
y = f1.getTT(); y
[0, 1, 0, 0]
start = [0,1]
seq = fsr(f1,start,10); seq
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
```

Sage Example 2.3 $f_2 = T_1T_3 + T_2$—monomials with exponent triples `[1,0,1]` $\hat{=}\,5$ and `[0,1,0]` $\hat{=}\,2$, hence ANF bitblock `[0,0,1,0,0,1,0,0]`

```
f2 = BoolF([0,0,1,0,0,1,0,0],method="ANF")
y = f2.getTT(); y
[0, 1, 0, 0]
start = [1,1,0]
seq = fsr(f2,start,10); seq
[0, 1, 1, 1, 0, 1, 1, 1, 0, 1]
```

Since the dimension of Z grows exponentially with the register length the prediction algorithm reaches its limits soon. In the worst case the stationary state of the ascending chain of subspaces—and the needed linear relation—occurs only after 2^l steps. This observation would make shift registers up to a length of about 32 predictable with manageable cost using linear algebra in a binary vector space of dimension 2^{32}.

However in the examples we observed that the linear relation we found is nothing other than the formula for the final periodic repetition. This was not a fortunate coincidence but is a general phenomen that has an easy proof. For details see the paper [6]. Hence instead of solving large systems of linear equations we can apply an algorithm for period search that needs significantly less resources. This approach enables a realistic attack on shift registers of lengths up to about 80.

From a general point of view there is another objection against using arbitrary FSRs: The feedback function f depends on 2^l parameters. To have f efficiently computable and to deal with a manageable key space we have to restrict the choice of f, say by forcing “almost all” coefficients a_I in the ANF of the “admissible” feedback functions f to 0. Thus we specify a “small” set $\mathcal{M} \subseteq \mathcal{P}(\{1,\ldots l\})$ a priori, and use only functions f whose ANF

$$f(x_1,\ldots,x_l) = \sum_{I\in\mathcal{P}(\{1,\ldots l\})} a_I x^I$$

has coefficients $a_I = 0$ for $I \notin \mathcal{M}$. Then the key space has size $2^{\#\mathcal{M}}$. However the coice of $\mathcal{M}$ is part of the encryption algorithm—in particular for a hardware FSR—, not a part of the key. KERCKHOFFS' principle warns us that the enemy will learn about $\mathcal{M}$ sooner or later. In the model of Figure 2.1 we treat the a priori "monomial supply" $\mathcal{M}$ as public parameter, and the concrete "monomial selection" I as secret parameter.

The necessity of choosing an efficiently computable feedback function and a manageable key space enforces restrictions that make the prediction method efficient too. Expressed in a somewhat sloppy way:

Proposition 9 *Each bit sequence that is generated by an FSR with efficiently computable feedback function is efficiently predictable.*

Our treatment of this problem was quite coarse. To derive mathematically correct statements there are two approaches:

1. Directly estimate the circuit complexity of the prediction algorithm by the circuit complexity of the feedback function.

2. Consider families of Boolean functions—that define families of FSRs—whose complexity grows polynomially with the register length, and show that the costs of the corresponding prediction procedures also grow at most polynomially.

For a comprehensive treatment see the cited paper [6].

We conclude that FSRs, no matter whether linear or nonlinear, are unsuited for generating pseudorandom sequences of cryptographic value—at least if naively applied. The method of BOYAR/KRAWCZYK breaks also nonlinear FSRs in realistic scenarios. And the result of BETH/DAI in Section 3.7 will open another promising way of predicting an FSR using the BERLEKAMP/MASSEY algorithm, see Section 3.3.

2.8 A General Congruential Generator

The prediction procedure becomes somewhat more involved when the module of a congruential generator is unknown. We abandon the general setting of commutative algebra and use special properties of the rings $\mathbb{Z}$ and $\mathbb{Z}/m\mathbb{Z}$, in particular the "canonical" representation of the residue classes of $\mathbb{Z}/m\mathbb{Z}$ by the subset $\{0,\ldots,m-1\} \subseteq \mathbb{Z}$.

Let $X = \mathbb{Z}^r$, $\bar{X} = (\mathbb{Z}/m\mathbb{Z})^r$, $Z = \mathbb{Z}^k$, $\bar{Z} = (\mathbb{Z}/m\mathbb{Z})^k$. The generator uses maps

$$\Phi^{(i)} : X^i \longrightarrow Z \quad \text{for } i \geq h,$$

$$\alpha : \bar{Z} \longrightarrow \bar{X} \quad \text{linear},$$

where α and m are unknown to the cryptanalyst. Identifying the residue classes with their canonical representants we consider $\bar{X}$ as the subset $\{0,\ldots,m-1\}^r$ of X. Then we generate a sequence by the same algorithm as in the previous Section 2.6, and call this procedure a **general congruential generator**, if the evaluation of the maps $\Phi^{(i)}$ is efficient with costs that depend at most polynomially on r, k, and $\log(m)$. In particular there is a bound M for the values of the $\Phi^{(i)}$ on $\{0,\ldots,m-1\}^{ri}$ that is at most polynomial in r, k, and $\log(m)$.

The cryptanalysis proceeds in two phases. In phase one we work over the ring $\mathbb{Z}$ and its quotient field $\mathbb{Q}$, and we determine a multiple $\hat{m}$ of the module m. In phase two we work over the ring $\mathbb{Z}/\hat{m}\mathbb{Z}$. Predicting x_n in this situation can trigger three different events:

- $z_n \notin Z_{n-1}$. Then the module Z_{n-1} (over $\mathbb{Q}$ or $\mathbb{Z}/\hat{m}\mathbb{Z}$) must be enlarged to Z_n, and no prediction is possible for x_n. The cryptanalyst needs some more plaintext.
- The prediction of x_n is correct.
- The prediction of x_n is false. Then the module $\hat{m}$ has to be adjusted.

In phase one Z_{n-1} is the vector space over $\mathbb{Q}$ that is spanned by $z_h,\ldots,z_{n-1}$ (omitting redundant z_i's).

Case 1: $z_n \notin Z_{n-1}$. Then set $Z_n = Z_{n-1} + \mathbb{Q}z_n$. This case can occur at most k times.

Case 2: [Linear relation] $z_n = t_h z_h + \cdots + t_{n-1} z_{n-1}$. Then predict $x_n = t_h x_h + \cdots + t_{n-1} x_{n-1}$ (as element of $\mathbb{Q}^r$).

Case 3: We have an analogous linear relation, but $\hat{x}_n = t_h x_h + \cdots + t_{n-1} x_{n-1}$ differs from x_n. Let $d \in \mathbb{N}$ be the common denominator of $t_h,\ldots,t_{n-1}$. Then

$$d\hat{x}_n = \alpha(dt_h z_h + \cdots + dt_{n-1} z_{n-1}) = \alpha(dz_n) = dx_n$$

in $\bar{X}$, that is $\bmod m$. This shows:

Lemma 8 *(*BOYAR*) The greatest common divisor $\hat{m}$ of the components of $d\hat{x}_n - dx_n$ in case 3 is a multiple of the module m.*

The result of phase one is a multiple $\hat{m} \neq 0$ of the true module m. The expense is:

- at most $k+1$ trials of solving a system of linear equations for up to k unknowns over $\mathbb{Q}$,
- one determination of the greatest common divisor of r integers.

Along the way the procedure correctly predicts a certain number of elements x_n, each time solving a system of linear equations of the same type.

How large can $\hat{m}$ be? For an estimate we need an upper bound M for all components of all $\Phi^{(i)}$ on $\{0, \ldots, m-1\}^{ri} \subseteq X^i$. We use HADAMARD's inequality: For arbitrary vectors $x_1, \ldots, x_k \in \mathbb{R}^k$ we have

$$|\operatorname{Det}(x_1, \ldots, x_k)| \leq \|x_1\|_2 \cdots \|x_k\|_2$$

where $\| \bullet \|_2$ is the Euclidean norm.

Lemma 9 *$\hat{m} \leq (k+1) \cdot m \cdot \sqrt{k^k} \cdot M^k$. In particular* $\log(\hat{m})$ *is bounded by a polynomial in k,* $\log(m)$, $\log(M)$.

Proof. The coefficient vector t is the solution of a system of at most k linear equations for the same number of unknowns. The coefficients z_i of this system are bounded by M. By HADAMARD's inequality and CRAMER's rule the numerators dt_i and denominators d of the solution are bounded by

$$\prod_{i=1}^{k} \sqrt{\sum_{j=1}^{k} M^2} = \prod_{i=1}^{k} \sqrt{kM^2} = \sqrt{k^k} \cdot M^k.$$

Hence the components of $d\hat{x}_n$ are bounded by

$$\|d\hat{x}_n\|_\infty = \|\sum dt_i x_i\|_\infty \leq \sqrt{k^k} \cdot M^k \cdot \sum \|x_i\|_\infty \leq km \cdot \sqrt{k^k} \cdot M^k$$

because m bounds the components of the x_i. We conclude

$$\|d\hat{x}_n - dx_n\|_\infty \leq km \cdot \sqrt{k^k} \cdot M^k + \sqrt{k^k} \cdot M^k \cdot m = (k+1) \cdot m \cdot \sqrt{k^k} \cdot M^k,$$

as claimed. $\diamond$

How does this procedure look in the example of an ordinary linear congruential generator? Here we have

$$z_1 = \begin{pmatrix} x_0 \\ 1 \end{pmatrix}, z_2 = \begin{pmatrix} x_1 \\ 1 \end{pmatrix}, z_3 = \begin{pmatrix} x_2 \\ 1 \end{pmatrix}, \ldots$$

If $x_1 = x_0$, then we have the trivial case of a constant sequence. Otherwise z_3 is a rational linear combination $t_1z_1 + t_2z_2$. Solving the system

$$\begin{aligned} x_0t_1 + x_1t_2 &= x_2, \\ t_1 + t_2 &= 1 \end{aligned}$$

yields

$$t = \frac{1}{d} \cdot \begin{pmatrix} -x_2 + x_1 \\ x_2 - x_0 \end{pmatrix} \quad \text{with } d = x_1 - x_0.$$

From this we derive the prediction

$$\hat{x}_3 = t_1x_1 + t_2x_2 = \frac{-x_2x_1 + x_1^2 + x_2^2 - x_2x_0}{x_1 - x_0} = \frac{(x_2 - x_1)^2}{x_1 - x_0} + x_2.$$

Hence $d(\hat{x}_3 - x_3) = (x_2 - x_1)^2 - (x_1 - x_0)(x_3 - x_2) = y_2^2 - y_1y_3$ where (y_i) is the sequence of differences. If $\hat{x}_3 = x_3$, then we must continue this way. Otherwise we get, see Lemma 6,

$$m|\hat{m} = |y_1y_3 - y_2^2|.$$

For our concrete standard example, where $x_0 = 2134$, $x_1 = 2160$, $x_2 = 6905$, $x_3 = 3778$, $y_1 = 26$, $y_2 = 4745$, $y_3 = -3127$, this general approach gives

$$\hat{m} = 4745^2 + 26 \cdot 3127 = 22596327.$$

A closer look, using Lemma 8 directly, even yields

$$t_1 = -\frac{365}{2},\ t_2 = \frac{367}{2},\ \hat{x}_3 = \frac{1745735}{2},\ \hat{m} = 2 \cdot (\hat{x}_3 - x_3) = 1738179.$$

In phase two of the algorithm we execute the same procedure but over the ring $\hat{R} = \mathbb{Z}/\hat{m}\mathbb{Z}$. However we can't simply reduce $\bmod \hat{m}$ the rational numbers from phase one. Hence we restart at z_h. Again we distinguish three cases for each single step:

Case 1: $z_n \notin \hat{Z}_{n-1} = \hat{R}z_h + \cdots + \hat{R}z_{n-1}$. Then set $\hat{Z}_n = \hat{Z}_{n-1} + \hat{R}z_n$ (and represent this $\hat{R}$-module by a non-redundant system $\{z_{j_1}, \ldots, z_{j_l}\}$ of generators where $z_{j_l} = z_n$). We can't predict x_n (but have to get it from somewhere else).

Case 2: $z_n = t_hz_h + \cdots + t_{n-1}z_{n-1}$. Then predict $x_n = t_hx_h + \cdots + t_{n-1}x_{n-1}$ (as an element of $\hat{X} = (\mathbb{Z}/\hat{m}\mathbb{Z})^r$). The prediction turns out to be correct.

Case 3: The same, but now the predicted value $\hat{x}_n = t_hx_h + \cdots + t_{n-1}x_{n-1}$ differs from x_n in $\hat{X}$. Then considering $\hat{x}_n - x_n$ as an element of $\mathbb{Z}^r$ we show:

Lemma 10 *In case 3 the greatest common divisor d of the coefficients of $\hat{x}_n - x_n$ is a multiple of m, but not a multiple of $\hat{m}$.*

Proof. It is a multiple of m since $\hat{x}_n \bmod m = x_n$. It is not a multiple of $\hat{m}$ since otherwise $\hat{x}_n = x_n$ in $\hat{X}$. $\diamond$

In case 3 we replace $\hat{m}$ by the greatest common divisor of d and $\hat{m}$ and reduce $\bmod \hat{m}$ all the former z_j. The lemma tells us that the new $\hat{m}$ is properly smaller than the old one.

By Lemma 9 case 3 can't occur too often, the number of occurences is polynomially in k, $\log(m)$, and $\log(M)$. If we already hit the true m this case can't occur any more. Case 1 may occur at most $\log_2(\#(\mathbb{Z}/\hat{m}\mathbb{Z})^k) = k \cdot \log_2(\hat{m})$ times in phase 2 by Proposition 5, and this bound is polynomial in k, $\log(m)$, and $\log(M)$.

> **Note.** There is a common aspect of phases one and two: In both cases we use the full quotient ring. The full quotient ring of $\mathbb{Z}$ is the quotient field $\mathbb{Q}$. In a residue class ring $\mathbb{Z}/m\mathbb{Z}$ the non-zero-divisors are exactly the elements that are coprime with m, hence the units. Thus $\mathbb{Z}/m\mathbb{Z}$ is its own full quotient ring.

For the concrete standard example we had $\hat{m} = 1738179$ after phase one, and now have to solve $\bmod \hat{m}$ the system (1) of linear equations. Since the determinant -26 is coprime with $\hat{m}$ we already have $Z_2 = \hat{R}^2$, and know that case 1 will never occur. The inverse of -26 is 66853 (in $\mathbb{Z}/\hat{m}\mathbb{Z}$), so from $-26\,t_1 = 4745$ we get $t_1 = 868907$. Hence $t_2 = 1 - t_1 = 869273$, and $\hat{x}_3 = 1_1 x_1 + t_2 x_2 = 3778$ is a correct prediction.

In the next step we calculate new coefficients t_1 and t_2 for the linear combination $z_4 = t_1 z_1 + t_2 z_2$. We solve (in $\mathbb{Z}/\hat{m}\mathbb{Z}$)

$$\begin{aligned} 2134\,t_1 + 2160\,t_2 &= 3778, \\ t_1 + t_2 &= 1. \end{aligned}$$

Eliminating t_2 yields $-26\,t_1 = 1618$, hence $t_1 = 401056$, and thus $t_2 = 1337124$, as well as $\hat{x}_4 = 1_1 x_1 + t_2 x_2 = 302190$. Since $x_4 = 8295$ we are in case 3 and must adjust $\hat{m}$:

$$\gcd(\hat{x}_4 - x_4, \hat{m}) = \gcd(293895, 1738179) = 8397.$$

Now $\hat{m} < 2x_2$. Thus from now on only case 2 will occur. This means that we'll predict all subsequent elements correctly.

A **prediction method** for a general congruential generator is an algorithm that gets the initial values $x_0, \ldots, x_{h-1}$ as input, then successively produces predictions of $x_h, x_{h+1}, \ldots$, and compares them with the true values; in the case of a mistake it adjusts the parameters using the respective true value.

A prediction method is **efficient** if

1. the cost of predicting each single x_n is polynomial in r, k, and $\log(m)$,

2. the number of false predictions is bounded by a polynomial in r, k, and $\log(m)$, as is the cost of adjusting the parameters in the case of a mistake.

The BOYAR/KRAWCZYK algorithm that we considered in this section fulfils requirement 2. It also fulfils requirement 1 since solving systems of linear equations over residue class rings $\mathbb{Z}/m\mathbb{Z}$ is efficient (as shown in Section 9.2 of Part I). Thus we have shown:

Theorem 2 *For an arbitrary (efficient) general congruential generator the* BOYAR/KRAWCZYK *algorithm is an efficient prediction method.*

A simple concrete example shows the application to a non-linear congruential generator. Suppose a quadratic generator of the form

$$x_n = ax_{n-1}^2 + bx_{n-1} + c \bmod m$$

outputs the sequence

$$x_0 = 63,\ x_1 = 96,\ x_2 = 17,\ x_3 = 32,\ x_4 = 37,\ x_5 = 72.$$

We set $X = \mathbb{Z}$, $Z = \mathbb{Z}^3$, $h = 1$. In phase one the vectors

$$z_1 = \begin{pmatrix} 3969 \\ 63 \\ 1 \end{pmatrix} z_2 = \begin{pmatrix} 9216 \\ 96 \\ 1 \end{pmatrix} z_3 = \begin{pmatrix} 289 \\ 17 \\ 1 \end{pmatrix}$$

span $\mathbb{Q}^3$ since the coefficient matrix is the VANDERMONDE matrix with determinant 119922. Solving

$$z_4 = \begin{pmatrix} 1024 \\ 32 \\ 1 \end{pmatrix} = t_1 z_1 + t_2 z_2 + t_3 z_3$$

yields

$$t_1 = \frac{160}{253}, \quad t_2 = -\frac{155}{869}, \quad t_3 = \frac{992}{1817},$$

with common denominator $d = 11 \cdot 23 \cdot 79 = 19987$. The algorithm predicts

$$\hat{x}_4 = \frac{1502019}{19987} \neq x_4.$$

Hence the first guessed module is

$$\hat{m} = d\hat{x}_4 - dx_4 = 762500,$$

and phase one is completed. Now we have to solve the same system of linear equations over $\mathbb{Z}/\hat{m}\mathbb{Z}$. Here the determinant is a zero divisor. We get two solutions, one of them being

$$t_1 = 156720, \quad t_2 = 719505, \quad t_3 = 648776.$$

Thus we predict the correct value

$$\hat{x}_4 = 156720 \cdot 96 + 719505 \cdot 17 + 648776 \cdot 32 \bmod 763500 = 37.$$

We are in case 2, and continue with predicting x_5: The system

$$z_5 = \begin{pmatrix} 1369 \\ 37 \\ 1 \end{pmatrix} = t_1 z_1 + t_2 z_2 + t_3 z_3$$

has two solutions, one of them being

$$t_1 = 2010\,, \quad t_2 = 558640\,, \quad t_3 = 201851\,,$$

hence

$$\hat{x}_5 = 136572\,, \quad \hat{x}_5 - x_5 = 136500\,.$$

We are in case 3 and adjust $\hat{m}$ to

$$\gcd(762500, 136500) = 500.$$

This exhausts the known values. Because all z_i are elements of $\hat{Z}_3 = \hat{R}z_1 + \hat{R}z_2 + \hat{R}z_3 \neq \hat{R}^3$ case 1 remains a possibility for the following steps. Since $x_0, \ldots, x_5$ are smaller than half the current module $\hat{m}$ also case 3 remains possible. In particular maybe we have to adjust the module furthermore.

Trying to predict x_6 we get $(\bmod 500)$

$$t_1 = 240\,, \quad t_2 = 285\,, \quad t_3 = 476\,, \quad x_6 = 117\,.$$

Exercise. What happens in the concrete standard example if after phase 1 we continue with the value $\hat{m} = 22596327$?

2.9 Analysis of Congruential Generators with Truncated Output

Cryptanalysis is significantly harder for pseudorandom generators that don't output all bits of the generated numbers. Then the sequence of differences is known at most approximately, greatest common divisors cannot be determined, and the algorithms of PLUMSTEAD-BOYAR or BOYAR/KRAWCZYK break down.

If the parameters of the pseudorandom generator are known, the cryptanalyst may try an exhaustion. The following consideration lacks mathematical strength. It doesn't presuppose that the pseudorandom generator is linear.

Suppose the generator produces n-bit integers but outputs only q bits (from fixed known positions) and suppresses $n-q$ bits. Then for each q-bit fragment of the output there exist 2^{n-q} possible complete values. In other words, a pseudorandom n-bit integer has the given bits at the given positions with probability $1/2^q$.

To continue we assume for simplicity that the q output bits are the most significant bits. So we decompose the integer x into $x = x_0 2^{n-q} + x_1$ where $0 \leq x_1 < 2^{n-q}$. The value x_0, the first q bits, is known. The cryptanalyst runs through the 2^{n-q} different possibilities for x_1. For each choice of x_1 she forms $x = x_0 2^{n-q} + x_1$ and sets $y = s(x)$ with the generating function s of the pseudorandom generator. She compares y with the next q bits of the output that she knows. If the pseudorandom generator is statistically good, then the probability of a hit is $1/2^q$. Thus from the 2^{n-q} test values of x_0 there survive about 2^{n-2q} ones. In the case $q \geq \frac{n}{2}$ she expects exactly one hit. Otherwise she proceeds. After using k substrings of q bits the expected number of hits is about 2^{n-kq}. The expected necessary number of q-bit substrings exceeds k only if $kq \leq n$, or $q \geq \frac{n}{k}$. For $q = \frac{1}{4}$ (as in the example $n = 32$, $q = 8$, that is an output of 8 bits of a 32-bit integer) four q-bit fragments suffice (where the exhaustion runs through 2^{24} integers). This trial-and-error procedure is manageable for small modules m. But note that the expense grows exponentially with m (assume the ratio $r = \frac{q}{n}$ of output bits is bounded away from 1).

For linear congruential generators with unknown module FRIEZE/KENNAN/LAGARIAS, HÅSTAD/SHAMIR, and J. STERN developed a better (probabilistic) procedure whose first step—finding the module—is summarized in the statement: *The cryptanalyst finds m with high probability if the generator outputs more than* $2/5$ *of the leading bits.* (without proof).

In the second step the cryptanalyst finds the multiplier a under the assumption that the module m is already known. In the third step she determines the complete integers x_i, or the differences y_i. Also with these

tasks she succeeds except for a negligeable subset of multipliers, and for the "good" multipliers she needs slightly more than one third of the leading bits of x_0, x_1, x_2, and x_3, to derive the complete integers. This enables her to predict all further output of the generator. A similar, somewhat weaker result by J. Stern holds for the case where instead of leading bits the generator outputs "inner bits" of the generated integers.

Thus the cryptanalysis of linear congruential generators reveals fundamental weaknesses, independently of the quality of their statistical properties.

Nevertheless linear congruential generators are useful for statistical applications. It is extremely unlikely that an application procedure "by accident" contains the steps that break a linear congruential generator and reveal its determinism. On the other hand linear congruential generators are disqualified for cryptographic applications once and for all, even with truncated output. However it is an open problem whether the objections also hold for a truncation strategy that outputs "very few" bits, say a quarter (note $\frac{1}{4} < \frac{2}{5}$), or only $\log\log(m)$ bits.

References

- J. Stern: Secret linear congruential generators are not cryptographically secure. FOCS 28 (1987), 421–426.
- Frieze/Håstad/Kannan/Lagarias/Shamir: Reconstructing truncated integer variables satisfying linear congruences. SIAM J. Comput. 17 (1988), 262–280.
- J. Boyar: Inferring sequences produced by a linear congruential generator missing low-order bits. J. Cryptology 1 (1989), 177–184.

2.10 Summary

In Sections 2.1 to 2.8 we developed a prediction method whose overall workflow is depicted in Figure 2.4.

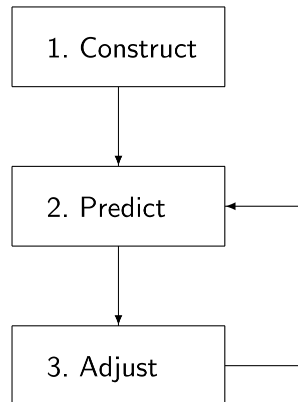


Figure 2.4: Workflow for prediction

1. By guessing plaintext the cryptanalyst finds subsequences of the key stream until she succeeds in constructing a linear relation for the state vectors (Noetherian principle).

2. Using this linear relation she predicts some more key bits.

3. If the predicted key bits are false (the plaintext ceases from making sense), then the cryptanalyst has to guess some more plaintext and to use it to adjust the parameters. Then she continues predicting bits.

This procedure is efficient for the "classical" pseudorandom generators, in particular for congruential generators—even with unknown module—and for feedback shift registers—even nonlinear ones. "Efficient" means that the computational cost is low, and also implies that the needed amount of known or correctly guessed plaintext is small.

One lesson learnt from these results is that for cryptographically secure pseudorandom generation we never should directly use the state of the generator as output. Rather we should insert a transformation between state and output that conceals the state—the output function of Figure 2.1. Section 2.9 illustrates that simply suppressing some bits —"truncating" or "decimating" the output—might be to weak as an output transformation. In the following section we'll learn about better output transformations.

There is a large twilight zone between pseudorandom generators that promise advantage to the cryptanalyst, and pseudorandom generators that

put the cipher designer's mind at ease. In any case we should prefer pseudorandom generators for which both of the procedures

- state transition,
- output function,

are nonlinear. The twilight zone where we don't know useful results on security contains (among others) quadratic congruential generators with slightly truncated output.

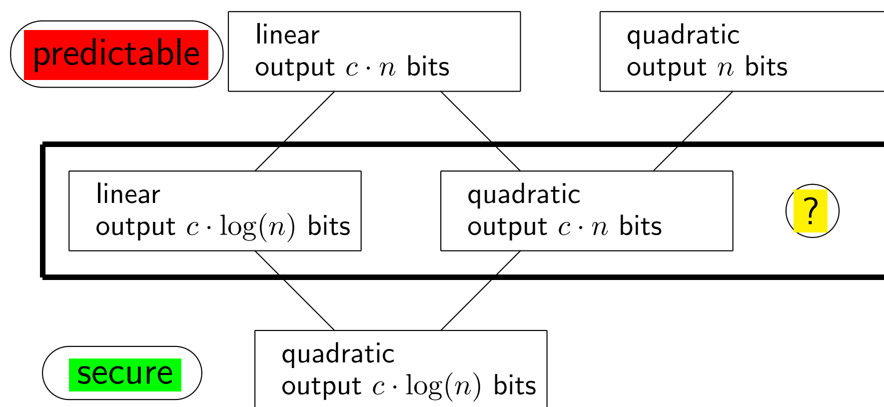


Figure 2.5: Predictable and secure congruential pseudorandom generators for n-bit integers (c a constant factor)

The following chapters present two approaches that are believed to lead to secure pseudorandom generators:

- combinations of LFSRs with a nonlinear output transformation (Chapter 3),
- nonlinear congruential generators with substantially truncated output (Chapter 4).

Chapter 3

Feedback Shift Registers and Linear Complexity

As we saw in the last chapter LFSRs are cryptographically weak if naively used. Also nonlinear FSRs admit an efficient prediction algorithm via the Noetherian principle undermining their security.

In this chapter we'll look at LFSRs from the opposite direction: Given a bit sequence, how to generate it by an LFSR in an optimal way? The minimal length of such an LFSR will turn out to be a useful measure of predictability—even a very good measure except for a few outliers.

3.1 The Linear Complexity of a Bit Sequence

We consider bit sequences $u = (u_i)_{i\in\mathbb{N}} \in \mathbb{F}_2^{\mathbb{N}}$—for the moment infinite ones. We search an LFSR of smallest length that produces the sequence.

If the sequence is generated by an LFSR, it must be periodic. On the other hand every periodic sequence is generated by an LFSR whose length is the sum of the lengths of preperiod and period—namely by the **circular LFSR** that feeds back the bit where the period begins: If $u_{l+i} = u_{k+i}$ for $i \geq 0$, then the taps are $a_{l-k} = 1$, $a_i = 0$ else, as in Figure 3.1. This consideration shows:

Lemma 11 *A bit sequence $u \in \mathbb{F}_2^{\mathbb{N}}$ is generated by an LFSR if and only if it is (eventually) periodic.*

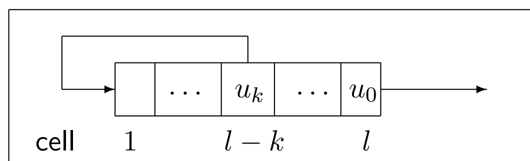


Figure 3.1: A circular LFSR

Definition The **linear complexity** $\lambda(u)$ of a bit sequence $u \in \mathbb{F}_2^{\mathbb{N}}$ is the minimal length of an LFSR that generates u.

For u constant 0 let $\lambda(u) = 0$, for a non-periodic u set $\lambda(u) = \infty$.

This concept of complexity uses the quite special machine model of an LFSR.

Remarks and examples

1. Let $\tau(u)$ be the sum of the lengths of the preperiod and the period of u. Assume that u is generated by an LFSR of length l. Then
$$\lambda(u) \leq \tau(u) \leq 2^l - 1 \quad \text{and} \quad \lambda(u) \leq l.$$
2. The periodically repeated sequence $0, \ldots, 0, 1$ ($l-1$ zeroes) has period l and linear complexity l. An LFSR of length $< l$ would start with the null vector as initial value and thus force the complete output sequence to zero.

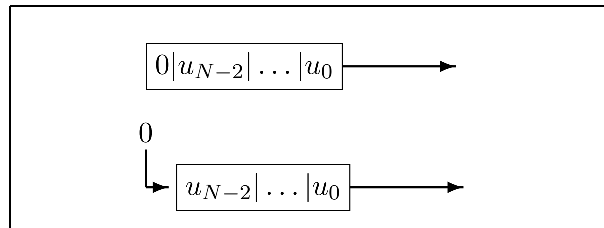
For a finite bit sequence $u = (u_0, \ldots, u_{N-1}) \in \mathbb{F}_2^N$ the linear complexity is analogously defined. In particular $\lambda(u)$ is the minimum integer l for which there exist $a_1, \ldots, a_l \in \mathbb{F}_2$ with

$$u_i = a_1u_{i-1} + \cdots + a_lu_{i-l} \quad \text{for } i = l, \ldots, N-1.$$

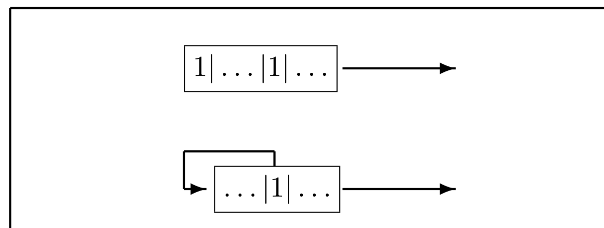
3. For $u \in \mathbb{F}_2^N$ we have $0 \leq \lambda(u) \leq N$.

4. $\lambda(u) = 0 \Longleftrightarrow u_0 = \cdots = u_{N-1} = 0$.

5. $\lambda(u) = N \Longleftrightarrow u = (0, \ldots, 0, 1)$. The implication "$\Longleftarrow$" follows as in remark 2. For the reverse direction assume $u_{N-1} = 0$. Then we can take the LFSR of length $N - 1$ with feedback constant 0—the two LFSRs



   both generate the same output of length N. This contradiction shows that $u_{N-1} = 1$. Assume there is a 1 at an earlier position. Then we can take the LFSR of length $N-1$ that feeds back exactly this position—the two LFSRs



   both generate the same output up to length N.

6. From the first $2\lambda(u)$ bits of the sequence u all the following bits are predictable. (Note that the cryptanalyst who knows that many bits of the sequence, but no further bits, also doesn't know $\lambda(u)$. Therefore she doesn't know that her predictions will be correct from now on. This ignorance doesn't prevent her from correctly predicting bit for bit!)

3.2 Synthesis of LFSRs

In this section we treat the problem of how to find an LFSR of shortest length that generates a given finite bit sequence. In section 2.6 we described a method of finding linear relations for sequence elements from a quite general generator. This might result in an LFSR, but anyway the linear relations might change from step to step and there appears no easy way of getting an optimal LFSR.

Here we follow another approach that solves our problem in a surprisingly easy way: the BM-algorithm, named after Berlekamp (1968 in a different context) and Massey (1969).

We don't use any special properties of the field $\mathbb{F}_2$, so we work over an arbitrary field K. Our goal is to construct a homogeneous linear generator of the smallest possible recursion depth l that generates a given finite sequence $u \in K^N$.

We consider a homogeneous linear generator whose recursion formula is

$$u_k = a_1 u_{k-1} + \cdots + a_l u_{k-l} \quad \text{for } k = l, \ldots, N-1. \tag{1}$$

Its coefficient vector is $(a_1, \ldots, a_l) \in K^l$. The polynomial

$$\varphi = 1 - a_1 T - \cdots - a_l T^l \in K[T]$$

is called **feedback polynomial**.

Note Don't confuse this polynomial with the feedback function

$$s(u_0, \ldots, u_{l-1}) = a_1 u_{l-1} + \cdots + a_l u_0.$$

The feedback polynomial is the reciprocal polynomial of the characteristic polynomial

$$\chi = \mathrm{Det}(T \cdot 1 - A) = T^l - a_1 T^{l-1} - \cdots - a_l$$

of the companion matrix

$$A = \begin{pmatrix} 0 & 1 & \ldots & 0 \\ & \ddots & \ddots & \\ & & & 1 \\ a_l & a_{l-1} & \ldots & a_1 \end{pmatrix}.$$

These two polynomials are related by the formula

$$\varphi = T^l \cdot \chi(\frac{1}{T}).$$

Lemma 12 *Let the sequence $u = (u_0, \ldots, u_{n-1}) \in K^n$ be a segment of the output of the linear generator (1), but not the sequence $\hat{u} = (u_0, \ldots, u_n) \in K^{n+1}$. Then every homogeneous linear generator of length $m \geq 1$ that generates $\hat{u}$ has $m \geq n + 1 - l$.*

Proof. **Case 1:** $l \geq n$. Then obviously $l + m \geq n + 1$.

Case 2: $l \leq n - 1$. Assume $m \leq n - l$. We have

$$u_j = a_1 u_{j-1} + \cdots + a_l u_{j-l} \quad \text{for } l \leq j \leq n-1.$$

Let $(b_1, \ldots, b_m)$ be the coefficient vector of a homogeneous linear generator that produces $\hat{u}$. Then

$$u_j = b_1 u_{j-1} + \cdots + b_m u_{j-m} \quad \text{for } m \leq j \leq n.$$

We deduce

$$\begin{aligned} u_n &\neq a_1 u_{n-1} + \cdots + a_l u_{n-l} \\ &= \sum_{i=1}^{l} a_i \cdot \underbrace{\sum_{k=1}^{m} b_k u_{n-i-k}}_{u_{n-i}} \quad [\text{since } n - l \geq m] \\ &= \sum_{k=1}^{m} b_k \cdot \underbrace{\sum_{i=1}^{l} a_i u_{n-k-i}}_{u_{n-k}} = u_n, \end{aligned}$$

contradiction. ◇

Consider a sequence $u \in K^N$. For $0 \leq n \leq N$ let $\lambda_n(u) = \lambda_n$ be the smallest recursion depth for which a homogeneous linear generator exists that produces $(u_0, \ldots, u_{n-1})$.

Lemma 13 *For every sequence $u \in K^N$ we have:*

(i) $\lambda_{n+1} \geq \lambda_n$ *for all n.*

(ii) *There is a homogeneous linear generator of recursion depth λ_n that produces $(u_0, \ldots, u_n)$ if and only if $\lambda_{n+1} = \lambda_n$.*

(iii) *If there is no such generator, then*

$$\lambda_{n+1} \geq n + 1 - \lambda_n.$$

Proof. (i) Every generator that produces $(u_0, \ldots, u_n)$ a forteriori produces $(u_0, \ldots, u_{n-1})$.

(ii) follows from (i).

(iii) The precondition of Lemma 12 is true for every generator of $(u_0, \ldots, u_{n-1})$. ◇

Proposition 10 [Massey] *Let $u \in K^N$ and $0 \leq n \leq N-1$. Let $\lambda_{n+1}(u) \neq \lambda_n(u)$. Then*

$$\lambda_n(u) \leq \frac{n}{2} \quad \textit{and} \quad \lambda_{n+1}(u) = n+1-\lambda_n(u).$$

Thus the linear complexity may jump only if λ_n (we often omit u in the notation) is “below the diagonal,” and then it jumps to the symmetric position “above the diagonal.” An illustration is in Figure 3.2.

Proof. First we consider the easy case $\lambda_n = 0$: Here $u_0 = \ldots = u_{n-1} = 0$. If $u_n = 0$, then $\lambda_{n+1} = \lambda_n = 0$, leaving nothing to prove. Otherwise $u_n \neq 0$, and then $\lambda_{n+1} = n+1 = n+1-\lambda_n$ by remark 5 in 3.1.

In general the first statement follows from the second one: We have $\lambda_n < \lambda_{n+1}$, hence $2\lambda_n < \lambda_n + \lambda_{n+1} = n+1$.

Now we prove the second statement by induction on n. In the case $n = 0$ we have $\lambda_0 = 0$—this case is already settled.

Now let $n \geq 1$. We may assume $l := \lambda_n \geq 1$. Let

$$u_j = a_1 u_{j-1} + \cdots + a_l u_{j-l} \quad \text{for } j = l, \ldots, n-1;$$

hence the feedback polynomial is

$$\varphi := 1 - a_1 T - \cdots - a_l T^l \in K[T].$$

Let the “n-th discrepancy” be defined as

$$d_n := u_n - a_1 u_{n-1} - \cdots - a_l u_{n-l}.$$

If $d_n = 0$, then the generator outputs u_n as the next element, and there is nothing to prove. Otherwise let $d_n \neq 0$. Let r be the length of the segment before the last increase of linear complexity, thus

$$t := \lambda_r < l, \quad \lambda_{r+1} = l.$$

By induction $l = r+1-t$. We have a relation

$$u_j = b_1 u_{j-1} + \cdots + b_t u_{j-t} \quad \text{for } j = t, \ldots, r-1,$$

the corresponding feedback polynomial is

$$\psi := 1 - b_1 T - \cdots - b_t T^t \in K[T],$$

and the corresponding r-th discrepancy,

$$d_r := u_r - b_1 u_{r-1} - \cdots - b_t u_{r-t} \neq 0.$$

In the case $t = 0$ we have $\psi = 1$ and $d_r = u_r$. Now we form the polynomial

$$\eta := \varphi - \frac{d_n}{d_r} \cdot T^{n-r} \cdot \psi = 1 - c_1 T - \cdots - c_m T^m \in K[T]$$

with $m = \deg \eta$. What is the output of the corresponding homogeneous linear generator? We have

$$\begin{aligned} u_j - \sum_{i=1}^{m} c_i u_{j-i} &= u_j - \sum_{i=1}^{l} a_i u_{j-i} - \frac{d_n}{d_r} \cdot \left[u_{j-n+r} - \sum_{i=1}^{t} b_i u_{j-n+r-i} \right] \\ &= 0 \quad \text{for } j = m, \ldots, n; \end{aligned}$$

for $j = m, \ldots, n-1$ this follows directly, for $j = n$ via the intermediate result $d_n - [d_n/d_r] \cdot d_r$. Hence the output is $(u_0, \ldots, u_n)$. Now we have

$$\lambda_{n+1} \leq m \leq \max\{l, n - r + t\} = \max\{l, n + 1 - l\}.$$

Since linear complexity grows monotonically we conclude $m > l$, and by Lemma 12 we get $m \geq n + 1 - l$. Hence $m = n + 1 - l$ and $\lambda_{n+1} = m$. This proves the proposition. $\diamond$

Corollary 1 *If $d_n \neq 0$ and $\lambda_n \leq \frac{n}{2}$, then*

$$\lambda_{n+1} = n + 1 - \lambda_n > \lambda_n.$$

Proof. By Lemma 12 we have $\lambda_{n+1} \geq n + 1 - \lambda_n$, thus $\lambda_{n+1} \geq \frac{n}{2} + 1 > \lambda_n$. By Proposition 10 we conclude $\lambda_{n+1} = n + 1 - \lambda_n$. $\diamond$

During the successive construction of a linear generator in the proof of the proposition, in each iteration step one of two cases occurs:

- $d_n = 0$: then $\lambda_{n+1} = \lambda_n$.
- $d_n \neq 0$: then
  - $\lambda_{n+1} = \lambda_n$ if $\lambda_n > \frac{n}{2}$,
  - $\lambda_{n+1} = n + 1 - \lambda_n$ if $\lambda_n \leq \frac{n}{2}$.

In particular we always have:

- If $\lambda_n > \frac{n}{2}$, then $\lambda_{n+1} = \lambda_n$.
- If $\lambda_n \leq \frac{n}{2}$, then $\lambda_{n+1} = \lambda_n$ or $\lambda_{n+1} = n + 1 - \lambda_n$.

By the way we found an alternative method of predicting LFSRs:

Corollary 2 *If $u \in \mathbb{F}_2^N$ is generated by an LFSR of length $\leq l$, then one such LFSR may be determined from $u_0, \ldots, u_{2l-1}$.*

Proof. Assume n is the first index $\geq 2l$ such that $d_n \neq 0$. Then $\lambda_n \leq l \leq \frac{n}{2}$, thus $\lambda_{n+1} = n + 1 - \lambda_n \geq l + 1$, contradiction. $\diamond$

3.3 The Berlekamp-Massey Algorithm

The proof of Proposition 10 is constructive: It contains an algorithm that successively builds a linear generator. For the step from length n to length $n+1$ three cases (1, 2a, 2b) are possible:

Case 1 $d_n = 0$, hence the generator with feedback polynomial φ next outputs u_n: Then φ and l remain unchanged, and so remain ψ, t, r, d_r.

Case 2 $d_n \neq 0$, hence the generator with feedback polynomial φ doesn't output u_n as next element: Then we form a new feedback polynomial η whose corresponding generator outputs $(u_0, \ldots, u_n)$. We distinguish between:

a) $l > \frac{n}{2}$: Then $\lambda_{n+1} = \lambda_n$. We replace φ by η and leave l, ψ, t, r, d_r unchanged.

b) $l \leq \frac{n}{2}$: Then $\lambda_{n+1} = n + 1 - \lambda_n$. We replace φ by η, l by $n + 1 - l$, ψ by φ, t by l, r by n, d_r by d_n.

So a semi-formal description of the Berlekamp-Massey algorithm (or BM algorithm) is:

Input: A sequence $u = (u_0, \ldots, u_{N-1}) \in K^N$.

Output: The linear complexity $\lambda_N(u)$,

the feedback polynomial φ of a linear generator of length $\lambda_N(u)$ that produces u.

Auxiliary variables: n = current index, initialized by $n := 0$,

l = current linear complexity, initialized by $l := 0$,

φ = current feedback polynomial $= 1 - a_1T - \cdots - a_lT^l$, initialized by $\varphi := 1$,

invariant condition: $u_i = a_1u_{i-1} + \cdots + a_lu_{i-l}$ for $l \leq i < n$,

d = current discrepancy $= u_n - a_1u_{n-1} - \cdots - a_lu_{n-l}$,

r = previous index, initialized by $r := -1$,

t = previous linear complexity,

ψ = previous feedback polynomial $= 1 - b_1T - \cdots - b_tT^t$, initialized by $\psi := 1$,

invariant condition: $u_i = b_1u_{i-1} + \cdots + b_tu_{i-t}$ for $t \leq i < r$,

d' = previous discrepancy $= u_r - b_1u_{r-1} - \cdots - b_tu_{r-t}$, initialized by $d' := 1$,

η = new feedback polynomial,

m = new linear complexity.

Iteration steps: For $n = 0, \ldots, N-1$:

$d := u_n - a_1 u_{n-1} - \cdots - a_l u_{n-l}$
If $d \neq 0$
  $\eta := \varphi - \frac{d}{d'} \cdot T^{n-r} \cdot \psi$
  If $l \leq \frac{n}{2}$ [linear complexity increases]
    $m := n + 1 - l$
    $t := l$
    $l := m$
    $\psi := \varphi$
    $r := n$
    $d' := d$
  $\varphi := \eta$
Output: $\lambda_N(u) := l$ and φ

Of course we may output also the complete sequence (λ_n).

As an **example** we apply the algorithm to the sequence 001101110. The steps where $d \neq 0$, $l \leq \frac{n}{2}$, are tagged by "[!]".

preconditions of the step	actions
$n = 0$ $u_0 = 0$ $l = 0$ $\varphi = 1$ $r = -1$ $d' = 1$ $t =$ $\psi = 1$	$d := u_0 = 0$
$n = 1$ $u_1 = 0$ $l = 0$ $\varphi = 1$ $r = -1$ $d' = 1$ $t =$ $\psi = 1$	$d := u_1 = 0$
$n = 2$ $u_2 = 1$ $l = 0$ $\varphi = 1$ $r = -1$ $d' = 1$ $t =$ $\psi = 1$	$d := u_2 = 1$ [!] $\eta := 1 - T^3$ $m := 3$
$n = 3$ $u_3 = 1$ $l = 3$ $\varphi = 1 - T^3$ $r = 2$ $d' = 1$ $t = 0$ $\psi = 1$	$d := u_3 - u_0 = 1$ $\eta := 1 - T - T^3$
$n = 4$ $u_4 = 0$ $l = 3$ $\varphi = 1 - T - T^3$ $r = 2$ $d' = 1$ $t = 0$ $\psi = 1$	$d := u_4 - u_3 - u_1 = -1$ $\eta := 1 - T + T^2 - T^3$
$n = 5$ $u_5 = 1$ $l = 3$ $\varphi = 1 - T + T^2 - T^3$ $r = 2$ $d' = 1$ $t = 0$ $\psi = 1$	$d := u_5 - u_4 + u_3 - u_2 = 1$ $\eta := 1 - T + T^2 - 2T^3$

From now on the results differ depending on the characteristic of the base field K. First assume $\operatorname{char} K \neq 2$. Then the procedure continues as follows:

preconditions of the step	actions
$n = 6 \quad u_6 = 1 \quad l = 3$ $\varphi = 1 - T + T^2 - 2T^3$ $r = 2 \quad d' = 1 \quad t = 0 \quad \psi = 1$	$d := u_6 - u_5 + u_4 - 2u_3 = -2$ [!] $\eta = 1 - T + T^2 - 2T^3 + 2T^4$ $m := 4$
$n = 7 \quad u_7 = 1 \quad l = 4$ $\varphi = 1 - T + T^2 - 2T^3 + 2T^4$ $r = 6 \quad d' = -2 \quad t = 3$ $\psi = 1 - T + T^2 - 2T^3$	$d := u_7 - u_6 + u_5 - 2u_4 + 2u_3 = 3$ $\eta = 1 + \frac{1}{2}T - \frac{1}{2}T^2 - \frac{1}{2}T^3 - T^4$
$n = 8 \quad u_8 = 0 \quad l = 4$ $\varphi = 1 + \frac{1}{2}T - \frac{1}{2}T^2 - \frac{1}{2}T^3 - T^4$ $r = 6 \quad d' = -2 \quad t = 3$ $\psi = 1 - T + T^2 - 2T^3$	$d := u_8 + \frac{1}{2}u_7 - \frac{1}{2}u_6 - \frac{1}{2}u_5 - u_4 = -\frac{1}{2}$ [!] $\eta := 1 + \frac{1}{2}T - \frac{3}{4}T^2 - \frac{1}{4}T^3 - \frac{5}{4}T^4 + \frac{1}{2}T^5$ $m := 5$

The resulting sequence of linear complexities is

$$\lambda_0 = 0, \lambda_1 = 0, \lambda_2 = 0, \lambda_3 = 3, \lambda_4 = 3, \lambda_5 = 3, \lambda_6 = 3, \lambda_7 = 4, \lambda_8 = 4, \lambda_9 = 5,$$

and the generating formula is

$$u_i = -\frac{1}{2}u_{i-1} + \frac{3}{4}u_{i-2} + \frac{1}{4}u_{i-3} + \frac{5}{4}u_{i-4} - \frac{1}{2}u_{i-5} \quad \text{for } i = 5, \ldots, 8.$$

For $\operatorname{char} K = 2$ the last three iteration steps look differently:

preconditions of the step	actions
$n = 6 \quad u_6 = 1 \quad l = 3$ $\varphi = 1 - T - T^2$ $r = 2 \quad d' = 1 \quad t = 0 \quad \psi = 1$	$d := u_6 - u_5 - u_4 = 0$
$n = 7 \quad u_7 = 1 \quad l = 3$ $\varphi = 1 - T - T^2$ $r = 2 \quad d' = 1 \quad t = 0 \quad \psi = 1$	$d := u_7 - u_6 - u_5 = 1$ [!] $\eta = 1 - T - T^2 - T^5$ $m := 5$
$n = 8 \quad u_8 = 0 \quad l = 5$ $\varphi = 1 - T - T^2 - T^5$ $r = 7 \quad d' = 1 \quad t = 3 \quad \psi = 1 - T - T^2$	$d := u_8 - u_7 - u_6 - u_3 = 1$ $\eta := 1 - T^3 - T^5$

In this case the sequence of linear complexities is

$$\lambda_0 = 0, \lambda_1 = 0, \lambda_2 = 0, \lambda_3 = 3, \lambda_4 = 3, \lambda_5 = 3, \lambda_6 = 3, \lambda_7 = 3, \lambda_8 = 5, \lambda_9 = 5,$$

and the generating formula is

$$u_i = u_{i-3} + u_{i-5} \quad \text{for } i = 5, \ldots, 8.$$

A Sage program for the char 2 case is in Sage Example 3.1. It uses the function `bmAlg` from Appendix C.2.

Figure 3.2 shows the growth of the linear complexities.

Sage Example 3.1 Applying the BM-algorithm

```
sage: u = [0,0,1,1,0,1,1,1,0]
sage: res = bmAlg(u)
sage: res
[[0, 0, 0, 3, 3, 3, 3, 3, 5, 5], T^5 + T^3 + 1]
```

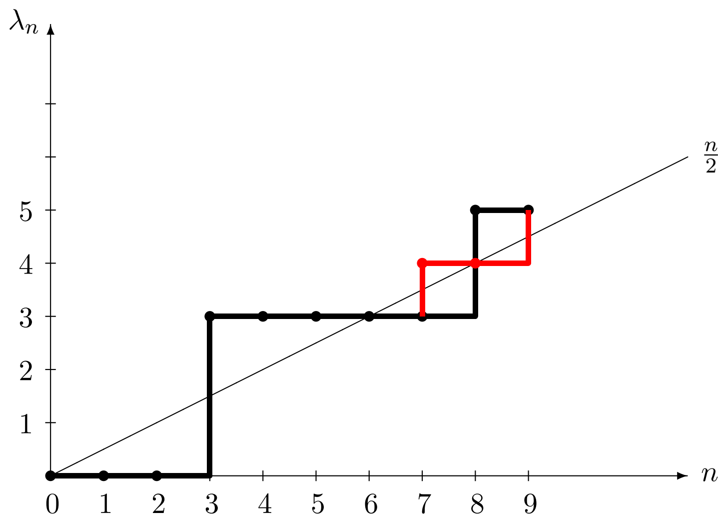


Figure 3.2: The sequence of linear complexities. The red line is for char $K \neq 2$.

The cost of the BM algorithm is $\mathrm{O}(N^2 \log N)$.

The sequence $(\lambda_n)_{n\in\mathbb{N}}$ or (for finite output sequences) $(\lambda_n)_{0\leq n\leq N}$ is called the **linearity profile** of the sequence u.

Here is the linearity profile of the first 128 bits of the sequence that we generated by an LFSR in Section 1.10:

$$(0, 1, 1, 2, 2, 3, 3, 4, 4, 4, 4, 7, 7, 7, 7, 8, 8, 9, 9, 10, 10, 11, 11, 12,$$

$$12, 13, 13, 13, 13, 16, 16, 16, 16, \ldots),$$

its graphic representation is in Figure 3.3:

In Section 4.1 we'll generate a "perfect" pseudorandom sequence. The linearity profile of its first 128 bits is:

$$(0, 1, 1, 1, 1, 4, 4, 4, 4, 5, 5, 5, 5, 8, 8, 8, 8, 8, 8, 8, 12, 12, 12, 12,$$

$$12, 12, 12, 12, 12, 17, 17, 17, 17, 17, 17, 18, 18, 18, 20, 20, 20, 21, 21,$$

$$22, 22, 22, 24, 24, 24, 24, 24, 24, 28, 28, 28, 28, 28, 29, 29, 30, 30, 31,$$

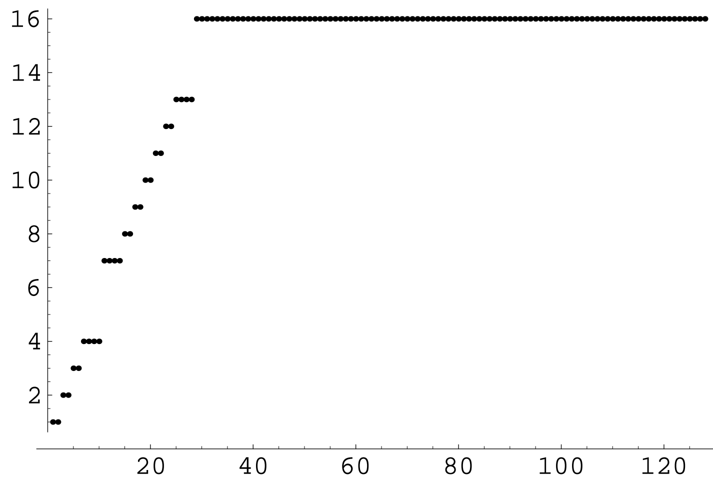


Figure 3.3: Linearity profile of an LFSR sequence

$$31, 32, 32, 32, 34, 34, 34, 34, 36, 36, 36, 37, 37, 38, 38, 39, 39, 40, 40,$$
$$41, 41, 41, 41, 41, 41, 46, 46, 46, 46, 46, 46, 47, 47, 48, 48, 49, 49, 50,$$
$$50, 50, 52, 52, 52, 53, 53, 54, 54, 54, 54, 54, 54, 54, 54, 61, 61, 61, 61,$$
$$61, 61, 61, 61, 61, 63, 63, 63, 64, 64),$$

graphically illustrated by Figure 3.4.

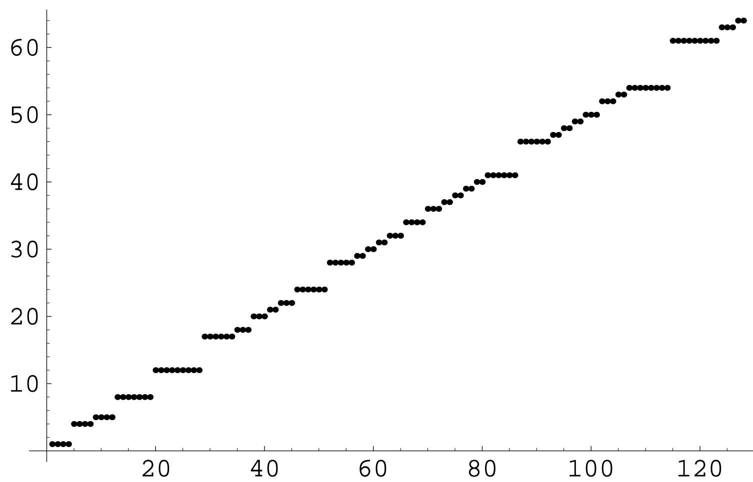


Figure 3.4: Linearity profile of a perfect pseudorandom sequence

In the second example we see a somewhat irregular oscillation around the diagonal, as should be expected for a "good" random sequence. The first example also shows a similar behaviour, but only until the linear complexity of the sequence is reached.

3.4 The BM Algorithm as a Cryptanalytic Tool

We revisit the cryptanalysis of an XOR ciphertext in Section 2.3 and explore how well the BM algorithm performs in this example following the cycle "construct – predict – adjust" as in Section 2.10. Remember the ciphertext:

```
10011100 10100100 01010110 10100110 01011101 10101110
01100101 10000000 00111011 10000010 11011001 11010111
00110010 11111110 01010011 10000010 10101100 00010010
11000110 01010101 00001011 11010011 01111011 10110000
10011111 00100100 00001111 01010011 11111101
```

For use with SageMath we provisionally fix its first 48 bits:

```
ciphtext = [1,0,0,1,1,1,0,0,1,0,1,0,0,1,0,0,0,1,0,1,0,
   1,1,0,1,0,1,0,0,1,1,0,0,1,0,1,1,1,0,1,1,0,1,0,1,1,1,0]
```

As in Section 2.3 we suspect that the cipher is XOR with a key stream from an LFSR, but now *of unknown length*. As before we guess that the text is in German and might begin with the word "Treffpunkt". To solve the cryptogram we need some bits of plaintext, say the first t letters (assumed in the 8-bit ISO 8859-1 character set), making up $8t$ bits of the key stream.

Let us tentatively start with two letters of plaintext: `Tr`, and the corresponding 16 keystream bits

```
        10011100 10100100  (ciphertext)
 Tr =   01010100 01110010  (assumed plaintext)
        -------- --------
        11001000 11010110  (keystream)
```

After attaching the Sage modules `Bitblock.sage`, `FSR.sage`, and `bmAlg.sage` from Appendix C (or Part II, Appendix E.1) we use the interactive commands

```
sage: kbits = [1,1,0,0,1,0,0,0,1,1,0,1,0,1,1,0]
sage: res = bmAlg(kbits)
sage: fbpol = res[1]; fbpol
T^8 + T^7 + T^5 + T^4 + T^3 + T^2 + T + 1
```

This result tells us that the shortest LFSR that generates our 16 keystream bits has length 8 and the taps 1, 2, 3, 4, 5, 7, 8 set. Next we initialize this LFSR in SageMath (note the reverse order of the bits in the initial state):

```
sage: coeff = [1,1,1,1,1,0,1,1]
sage: reg = LFSR(coeff)
sage: start = [0,0,0,1,0,0,1,1]
sage: reg.setState(start)
```

Using this LFSR we predict 32 more, hence altogether 48 tentative keystream bits:

```
sage: testkey = reg.nextBits(48); testkey
[1,1,0,0,1,0,0,0,1,1,0,1,0,1,1,0,1,0,0,1,1,1,0,0,
 0,0,0,0,0,1,1,0,0,0,0,1,0,1,1,1,0,1,1,1,1,0,0,0]
```

These tentative key bits yield 48 bits of experimental plaintext, represented by 6 bytes in decimal notation:

```
sage: testplain = xor(ciphtext,testkey)
sage: testtext = []
sage: for i in range(6):
    block = testplain[8*i:8*i+8]
    nr = bbl2int(block)
    testtext.append(nr)
sage: testtext
[84, 114, 202, 160, 74, 214]
```

or, written as ISO 8859-1 characters, "TrÊ̊⊔JÖ" (where ⊔ represents the non-breaking space)—a definitive failure.

So let us guess one more letter of plaintext: `Tre`, and use the corresponding 24 keystream bits

```
          10011100 10100100 01010110  (ciphertext)
    Tre = 01010100 01110010 01100101  (assumed plaintext)
          -------- -------- --------
          11001000 11010110 00110011  (keystream)
```

As above we apply the BM algorithm interactively and get an LFSR of length 12 with feedback polynomial $T^{12}+T^{10}+T^9+T^8+T^6+T^5+T^3+T+1$, hence taps $1, 3, 5, 6, 8, 9, 10, 12$. Setting up the LFSR and predicting 48 keystream bits:

```
sage: coeff = [1,0,1,0,1,1,0,1,1,1,0,1]
sage: reg = LFSR(coeff)
sage: start = [1,0,1,1,0,0,0,1,0,0,1,1]
sage: reg.setState(start)
sage: testkey = reg.nextBits(48); testkey
[1,1,0,0,1,0,0,0,1,1,0,1,0,1,1,0,0,0,1,1,0,0,1,1,
 0,1,0,1,0,0,0,0,1,0,0,0,1,0,1,1,0,1,0,1,0,0,0,1]
```

we again get 48 bits of experimental plaintext, as bytes in decimal notation: `[84, 114, 101, 246, 214, 255]`. The translation to ISO 8859-1 yields the next flop: "TreöÖÿ".

As next step we use four letters of known plaintext `Tref` (as in Section 2.3) and derive 32 tentative keystream bits:

```
        10011100 10100100 01010110 10100110  (ciphertext)
Tref =  01010100 01110010 01100101 01100110  (assumed plaintext)
        -------- -------- -------- --------
        11001000 11010110 00110011 11000000  (keystream)
```

The BM algorithm yields an LFSR of length 16 with feedback polynomial $T^{16} + T^5 + T^3 + T^2 + 1$, hence taps $2, 3, 5, 16$. It predicts 48 keystream bits:

```
sage: coeff = [0,1,1,0,1,0,0,0,0,0,0,0,0,0,0,1]
sage: reg = LFSR(coeff)
sage: start = [0,1,1,0,1,0,1,1,0,0,0,1,0,0,1,1]
sage: reg.setState(start)
sage: testkey = reg.nextBits(48); testkey
[1,1,0,0,1,0,0,0,1,1,0,1,0,1,1,0,0,0,1,1,0,0,1,1,
 1,1,0,0,0,0,0,0,0,0,1,1,1,0,1,1,1,0,0,0,1,1,1,0]
```

and the experimental plaintext `[84, 114, 101, 102, 102, 32]` that looks promising: "Treff␣" (where ␣ here represents simple space character).

Sure of victory we decipher the complete text:

```
sage: cstream = "10011100101...1111111101"
sage: fullcipher = str2bbl(cstream)
sage: start = [0,1,1,0,1,0,1,1,0,0,0,1,0,0,1,1]
sage: reg.setState(start)
sage: keystream = reg.nextBits(232)
sage: fullplain = xor(fullcipher,keystream)
sage: fulltext = []
sage: for i in range(232/8):
    block = fullplain[8*i:8*i+8]
    nr = bbl2int(block)
    fulltext.append(nr)
sage: fulltext
[84,114,101,102,102,32,109,111,114,103,101,110,32,56,32,85,104,114,
 32,66,97,104,110,104,111,102,32,77,90]
 T  r   e   f   f   _  m   o   r   g   e   n   _  8  _  U  h   r
 _  B  a  h   n   h   o   f   _  M  Z
```

"Meeting tomorrow at 8 p. m. train station Mainz".

Remark

The success of this cryptanalytic approach crucially depends on the LFSR scenario, or in other words on a linearity profile like that in Figure 3.3 for the keystream. If the keystream comes from another kind of source we expect a linearity profile as in Figure 3.4 and shall not be able to make a stable prediction before the plaintext is exhausted.

We could also try nonlinear FSRs in an analoguous way as in Appendix B. Unfortunately most trials—even if the recursive profile stabilizes—will find a trivial FSR that allows no prediction beyond the end of the already known partial key sequence, see Appendix B. Then the approach "construct – predict – adjust" cannot work better than by guessing more keystream bits in a purely random way.

3.5 The Distribution of Linear Complexity

The distribution of the linear complexities of bit sequences of a fixed length can be determined exactly.

A given sequence $u = (u_0, \ldots, u_{N-1}) \in \mathbb{F}_2^N$ has two possible extensions $\tilde{u} = (u_0, \ldots, u_N) \in \mathbb{F}_2^{N+1}$ by 1 bit. The relation between $\lambda(\tilde{u})$ and $\lambda(u)$ is given by the Massey recursion: Let

$$\delta = \begin{cases} 0 & \text{if the prediction is correct,} \\ 1 & \text{otherwise.} \end{cases}$$

Here "prediction" refers to the next outpit bit from the LFSR we constructed for u. Then

$$\lambda(\tilde{u}) = \begin{cases} \lambda(u) & \text{if } \delta = 0, \\ \lambda(u) & \text{if } \delta = 1 \text{ and } \lambda(u) > \frac{N}{2}, \\ N + 1 - \lambda(u) & \text{if } \delta = 1 \text{ and } \lambda(u) \leq \frac{N}{2}. \end{cases}$$

In the middle case we need a new LFSR, but of the same length.

From these relations we derive a formula for the number $\mu_N(l)$ of all sequences of length N that have a given linear complexity l. To this end let

$$\begin{aligned} M_N(l) &:= \{u \in \mathbb{F}_2^N \mid \lambda(u) = l\} \quad \text{for } N \geq 1 \text{ and } l \in \mathbb{N}, \\ \mu_N(l) &:= \#M_N(l). \end{aligned}$$

The following three statements are immediately clear:

- $0 \leq \mu_N(l) \leq 2^N$,
- $\mu_N(l) = 0$ for $l > N$,
- $\sum_{l=0}^{N} \mu_N(l) = 2^N$.

From these we find explicit rules for the recursion from $\mu_{N+1}(l)$ to $\mu_N(l)$:

Case 1, $0 \leq l \leq \frac{N}{2}$. Every $u \in \mathbb{F}_2^N$ may be continued in two different ways: $u_N = 0$ or 1. Exactly one of them matches the prediction and leads to $\tilde{u} \in M_{N+1}(l)$. The other one leads to $\tilde{u} \in M_{N+1}(N+1-l)$. Since there are no other contributions to $M_{N+1}(l)$ we conclude $\mu_{N+1}(l) = \mu_N(l)$.

Case 2, $l = \frac{N+1}{2}$ (may occur only for odd N). The correctly predicted u_N leads to $\tilde{u} \in M_{N+1}(l)$, however the same is true for the mistakenly predicted one because of the Massey recursion. Hence $\mu_{N+1}(l) = 2 \cdot \mu_N(l)$.

Case 3, $l \geq \frac{N}{2} + 1$. Both possible continuations lead to $\tilde{u} \in M_{N+1}(l)$. Additionally we have one element from each of of the wrong predictions of all $u \in M_{N+1-l}(l)$ from case 1. Hence $\mu_{N+1}(l) = 2 \cdot \mu_N(l) + \mu_{N+1-l}(l)$.

The following lemma summarizes these considerations:

Lemma 14 *The frequency $\mu_N(l)$ of bit sequences of length N and linear complexity l complies with the recursion*

$$\mu_{N+1}(l) = \begin{cases} \mu_N(l) & \text{if } 0 \leq l \leq \frac{N}{2}, \\ 2 \cdot \mu_N(l) & \text{if } l = \frac{N+1}{2}, \\ 2 \cdot \mu_N(l) + \mu_{N+1-l}(l) & \text{if } l \geq \frac{N}{2} + 1. \end{cases}$$

From this recursion we get an explicit formula:

Proposition 11 [Rueppel] *The frequency $\mu_N(l)$ of bit sequences of length N and linear complexity l is given by*

$$\mu_N(l) = \begin{cases} 1 & \text{if } l = 0, \\ 2^{2l-1} & \text{if } 1 \leq l \leq \frac{N}{2}, \\ 2^{2(N-l)} & \text{if } \frac{N+1}{2} \leq l \leq N, \\ 0 & \text{if } l > N. \end{cases}$$

Proof. For $n = 1$ we have $M_1(0) = \{(0)\}$, $M_1(1) = \{(1)\}$, hence $\mu_1(0) = \mu_1(1) = 1$.

Now we proceed by induction from N to $N+1$. The case $l = 0$ is trivial since $M_{N+1}(0) = \{(0, \ldots, 0)\}$, $\mu_{N+1}(0) = 1$. As before we distinguish three cases:

Case 1, $1 \leq l \leq \frac{N}{2}$. A forteriori $1 \leq l \leq \frac{N+1}{2}$, and

$$\mu_{N+1}(l) = \mu_N(l) = 2^{2l-1}.$$

Case 2, $l = \frac{N+1}{2}$ (N odd). Here $\mu_N(l) = 2^{2(N-l)}$, and the exponent is $2N - 2l = 2N - N - 1 = N - 1 = 2l - 2$, hence

$$\mu_{N+1}(l) = 2 \cdot 2^{2(N-l)} = 2^{2l-2+1} = 2^{2l-1}.$$

Case 3, $l \geq \frac{N}{2} + 1$. Again $\mu_N(l) = 2^{2(N-l)}$. For $l' = N + 1 - l$ we have $l' \leq N + 1 - \frac{N}{2} - 1 = \frac{N}{2}$, hence $\mu_N(l') = 2^{2l'-1}$, and

$$\begin{aligned} \mu_{N+1}(l) &= 2\mu_N(l) + \mu_N(l') = 2^{2N-2l+1} + 2^{2N-2l+1} \\ &= 2^{2N-2l+2} = 2^{2(N+1-l)}. \end{aligned}$$

This completes the proof. $\diamond$

Table 3.1 gives an impression of the distribution.

	1	2	3	4	5	6	7	8	9	10	$N \rightarrow$
0	1	1	1	1	1	1	1	1	1	1	
1	1	2	2	2	2	2	2	2	2	2	
2		1	4	8	8	8	8	8	8	8	
3			1	4	16	32	32	32	32	32	
4				1	4	16	64	128	128	128	
5					1	4	16	64	256	512	
6						1	4	16	64	256	
7							1	4	16	64	
8								1	4	16	
9									1	4	
10										1	
l											
$\downarrow$											

Table 3.1: The distribution of linear complexity

Observations

- Row l is constant from $N = 2l$ on (red numbers), the diagonals, from $N = 2l - 1$ on (blue numbers).
- Each column N, from row $l = 1$ to row $l = N$, contains the powers 2^k, $k = 0, \ldots, N - 1$, each one exactly once—first the odd powers in ascending order (red), followed by the even powers (blue) in descending order.
- For every length N there is exactly one sequence of linear complexity 0 and N each: From Section 3.1 we know that these are the sequences $(0, \ldots, 0, 0)$ and $(0, \ldots, 0, 1)$.

Figure 3.5 shows the histogram of this distribution for $N = 10$, Figure 3.6, for $N = 100$. The second histogram looks strikingly small. We'll clarify this phenomen in the following Section 3.6.

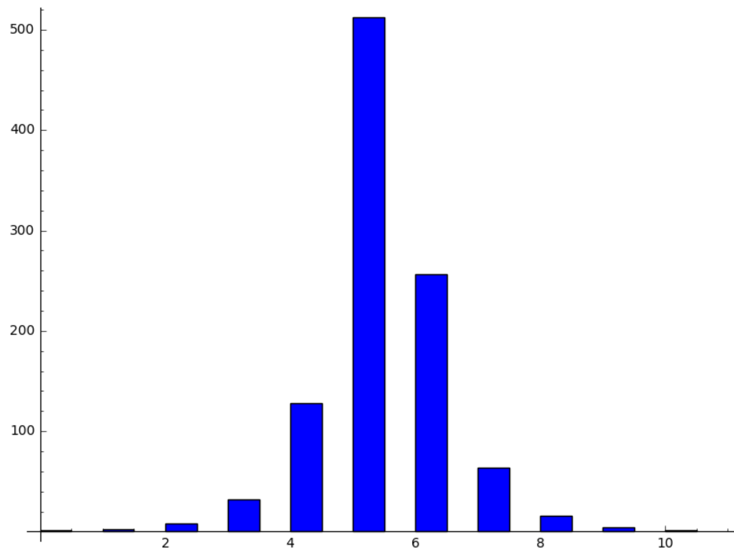


Figure 3.5: The distribution of linear complexity for bitsequences of length $N = 10$

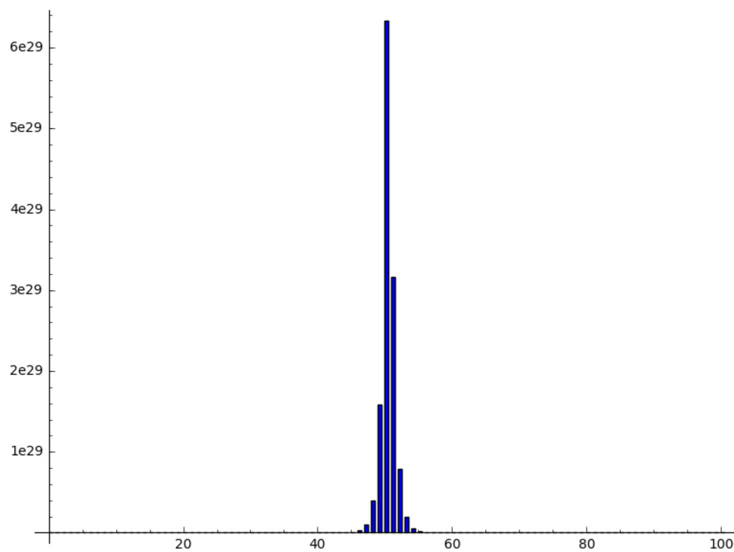


Figure 3.6: The distribution of linear complexity for bitsequences of length $N = 100$

3.6 The Mean Value of the Linear Complexity

From the exact distribution of the linear complexity we also can exactly determine the mean value and the variance (for fixed length N):

Theorem 3 (Rueppel) *Explicit formulas for the mean value*

$$E_N = \frac{1}{2^N} \cdot \sum_{u \in \mathbb{F}_2^N} \lambda(u)$$

and the variance V_N of the linear complexity of all bit sequences of length N are:

$$\begin{aligned} E_N &= \frac{N}{2} + \frac{2}{9} + \frac{\varepsilon}{18} - \frac{N}{3 \cdot 2^N} - \frac{2}{9 \cdot 2^N} \approx \frac{N}{2}, \\ V_N &= \frac{86}{81} - \frac{14-\varepsilon}{27} \cdot \frac{N}{2^N} - \frac{82-2\varepsilon}{81} \cdot \frac{1}{2^N} - \frac{9N^2+12N+4}{81} \cdot \frac{1}{2^{2N}} \approx \frac{86}{81} \end{aligned}$$

where $\varepsilon = 0$ for N even, $\varepsilon = 1$ for N odd (ε is the parity of N).

Remarkably the variance is almost independent of N. Thus almost all linear complexities vary around the mean value in a small strip only that is (almost) independent of N and becomes *relatively* more narrow with increasing N as illustrated by Figures 3.5 and 3.6.

For the proof we have to make a small detour. We'll encounter sums that have a nice expression using a well-known trick from calculus.

Lemma 15 *For the derivatives of the function*

$$f \colon \mathbb{R} - \{1\} \longrightarrow \mathbb{R}, \quad f(x) = \frac{x^{r+1} - x}{x - 1},$$

we have the formulas:

$$f'(x) = \frac{1}{(x-1)^2} \cdot \left[r x^{r+1} - (r+1)x^r + 1 \right],$$

$$f''(x) = \frac{1}{(x-1)^3} \cdot \left[(r^2 - r)x^{r+1} - 2(r^2-1)x^r + (r^2+r)x^{r-1} - 2 \right],$$

$$x^2 f''(x) + x f'(x) = \frac{x}{(x-1)^3} \cdot \left[r^2 x^{r+2} - (2r^2+2r-1)x^{r+1} + (r+1)^2 x^r - x - 1 \right].$$

Proof. By direct calculation. $\diamond$

Using these formulas for f we explicitly calculate some sums:

Corollary 1 *For all $x \in \mathbb{R}$, $x \neq 1$, we have:*

$$\begin{aligned}
\sum_{i=1}^{r} x^i &= \frac{1}{x-1} \cdot \left[x^{r+1} - x\right], \\
\sum_{i=1}^{r} i x^i &= \frac{x}{(x-1)^2} \cdot \left[r x^{r+1} - (r+1)x^r + 1\right], \\
\sum_{i=1}^{r} i^2 x^i &= \frac{x}{(x-1)^3} \cdot \left[r^2 x^{r+2} - (2r^2+2r-1)x^{r+1} + (r+1)^2 x^r - x - 1\right].
\end{aligned}$$

Proof. From the sum formula for the geometric series we conclude

$$\sum_{i=1}^{r} x^i = x \cdot \sum_{i=0}^{r-1} x^i = x \cdot \frac{x^r - 1}{x-1} = f(x),$$

$$\sum_{i=1}^{r} i x^i = x \cdot \sum_{i=1}^{r} i x^{i-1} = x \cdot f'(x),$$

$$\sum_{i=1}^{r} i^2 x^i = \sum_{i=1}^{r} i(i-1)x^i + \sum_{i=1}^{r} i x^i = x^2 \cdot f''(x) + x \cdot f'(x).$$

Therefore the claimed formulas follow from Lemma 15. $\diamond$

Corollary 2

$$\begin{aligned}
\sum_{i=1}^{r} i\, 2^{2i-1} &= \frac{3r-1}{9} \cdot 2^{2r+1} + \frac{2}{9}, \\
\sum_{i=1}^{r} i^2\, 2^{2i-1} &= \frac{3r^2-2r}{9} \cdot 2^{2r+1} + \frac{5}{27} \cdot 2^{2r+1} - \frac{10}{27}.
\end{aligned}$$

Proof.

$$\sum_{i=1}^{r} i 2^{2i-1} = \frac{1}{2} \cdot \sum_{i=1}^{r} i 4^i = \frac{1}{2} \cdot \frac{4}{9} \cdot \left[r 4^{r+1} - (r+1)4^r + 1\right] = \frac{2}{9} \cdot \left[3r4^r - 4^r + 1\right],$$

$$\begin{aligned}
\sum_{i=1}^{r} i^2 2^{2i-1} &= \frac{1}{2} \cdot \sum_{i=1}^{r} i^2 4^i = \frac{1}{2} \cdot \frac{4}{27} \cdot \left[r^2 4^{r+2} - (2r^2+2r-1)4^{r+1} + (r+1)^2 4^r - 5\right] \\
&= \frac{2}{27} \cdot \left[(9r^2 - 6r + 5) \cdot 4^r - 5\right].
\end{aligned}$$

$\diamond$

Now the mean value of the linear complexity is

$$E_N = \frac{1}{2^N} \cdot \sum_{u \in \mathbb{F}_2^N} \lambda(u) = \frac{1}{2^N} \cdot \sum_{l=0}^{N} l \cdot \mu_N(l),$$

$$2^N E_N = \underbrace{\sum_{l=1}^{\lfloor \frac{N}{2} \rfloor} l \cdot 2^{2l-1}}_{S_1} + \underbrace{\sum_{l=\lceil \frac{N+1}{2} \rceil}^{N} l \cdot 2^{2(N-l)}}_{S_2}.$$

First let N be even. Then

$$\begin{aligned}
S_1 &= \sum_{l=1}^{\frac{N}{2}} l \cdot 2^{2l-1} = \frac{3N-2}{18} \cdot 2^{N+1} + \frac{2}{9} = \frac{N}{3} \cdot 2^N - \frac{2}{9} \cdot 2^N + \frac{2}{9}, \\
S_2 &= \sum_{l=\frac{N}{2}+1}^{N} l \cdot 4^{N-l} \stackrel{k=N-l}{=} \sum_{k=0}^{\frac{N}{2}-1} (N-k) \cdot 4^k = N \cdot \sum_{k=0}^{\frac{N}{2}-1} 4^k - \sum_{k=0}^{\frac{N}{2}-1} k \cdot 4^k \\
&= N \cdot \frac{4^{N/2}-1}{3} - \frac{4}{9} \cdot \left[(\frac{N}{2} - 1) \cdot 4^{\frac{N}{2}} - \frac{N}{2} \cdot 4^{\frac{N}{2}-1} + 1 \right] \\
&= \frac{N}{3} \cdot 2^N - \frac{N}{3} - \frac{4}{9} \cdot \left[\frac{N}{2} \cdot 2^N - 2^N - \frac{N}{8} \cdot 2^N + 1 \right] \\
&= \left(\frac{N}{6} + \frac{4}{9} \right) \cdot 2^N - \frac{N}{3} - \frac{4}{9}.
\end{aligned}$$

Taken together this yields

$$2^N E_N = \frac{N}{2} \cdot 2^N + \frac{2}{9} \cdot 2^N - \frac{N}{3} - \frac{2}{9},$$

proving the first formula of Theorem 3 for N even.

For odd N we have

$$\begin{aligned}
S_1 &= \sum_{l=1}^{\frac{N-1}{2}} l \cdot 2^{2l-1} = \frac{3(N-1)-2}{18} \cdot 2^N + \frac{2}{9} = \frac{3N-5}{18} \cdot 2^N + \frac{2}{9} \\
&= \frac{N}{6} \cdot 2^N - \frac{5}{18} \cdot 2^N + \frac{2}{9},
\end{aligned}$$

$$\begin{aligned}
S_2 &= \sum_{l=\frac{N+1}{2}}^{N} l\cdot 4^{N-l} \stackrel{k=N-l}{=} \sum_{k=0}^{\frac{N-1}{2}} (N-k)\cdot 4^k = N\cdot\sum_{k=0}^{\frac{N-1}{2}} 4^k - \sum_{k=0}^{\frac{N-1}{2}} k\cdot 4^k \\
&= N\cdot\frac{4^{(N+1)/2}-1}{3} - \frac{4}{9}\cdot\left[\frac{N-1}{2}\cdot 4^{\frac{N+1}{2}} - \frac{N+1}{2}\cdot 4^{\frac{N-1}{2}} + 1\right] \\
&= \frac{N}{3}\cdot 2^{N+1} - \frac{N}{3} - \frac{4}{9}\cdot\left[\frac{N-1}{2}\cdot 2^{N+1} - \frac{N+1}{2}\cdot 2^{N-1} + 1\right] \\
&= \frac{2N}{3}\cdot 2^N - \frac{N}{3} - \frac{4N}{9}\cdot 2^N + \frac{4}{9}\cdot 2^N + \frac{N}{9}\cdot 2^N + \frac{1}{9}\cdot 2^N - \frac{4}{9} \\
&= \left(\frac{N}{3}+\frac{5}{9}\right)\cdot 2^N - \frac{N}{3} - \frac{4}{9},
\end{aligned}$$

$$2^N E_N = \frac{N}{2}\cdot 2^N + \frac{5}{18}\cdot 2^N - \frac{N}{3} - \frac{2}{9},$$

proving the first formula of Theorem 3 also for odd N.

Now let's calculate the variance V_N. We start with

$$\begin{aligned}
V_N + 2^N E_N^2 &= \frac{1}{2^N}\cdot\sum_{u\in\mathbb{F}_2^N}\lambda(u)^2 = \frac{1}{2^N}\cdot\sum_{l=0}^{N} l^2\cdot\mu_N(l), \\
&= \underbrace{\sum_{l=1}^{\lfloor\frac{N}{2}\rfloor} l^2\cdot 2^{2l-1}}_{S_3} + \underbrace{\sum_{l=\lceil\frac{N+1}{2}\rceil}^{N} l^2\cdot 4^{N-l}}_{S_4}.
\end{aligned}$$

Again we first treat the case of even N. Then the first sum evaluates as

$$\begin{aligned}
S_3 &= \sum_{l=1}^{\frac{N}{2}} l^2\cdot 2^{2l-1} = \frac{3\cdot\frac{N^2}{4} - 2\cdot\frac{N}{2}}{9}\cdot 2^{N+1} + \frac{5}{27}\cdot 2^{N+1} - \frac{10}{27} \\
&= \frac{N^2}{6}\cdot 2^N - \frac{2N}{9}\cdot 2^N + \frac{10}{27}\cdot 2^N - \frac{10}{27}.
\end{aligned}$$

We decompose the second sum:

$$\begin{aligned}
S_4 &= \sum_{l=\frac{N}{2}+1}^{N} l^2\cdot 4^{N-l} \stackrel{k=N-l}{=} \sum_{k=0}^{\frac{N}{2}-1} (N-k)^2\cdot 4^k \\
&= N^2\cdot\underbrace{\sum_{k=0}^{\frac{N}{2}-1} 4^k}_{S_{4a}} - 2N\cdot\underbrace{\sum_{k=0}^{\frac{N}{2}-1} k\cdot 4^k}_{S_{4b}} + \underbrace{\sum_{k=0}^{\frac{N}{2}-1} k^2\cdot 4^k}_{S_{4c}}
\end{aligned}$$

and separately evaluate the summands:

$$\begin{aligned}
S_{4a} &= N^2 \cdot \frac{4^{\frac{N}{2}} - 1}{3} = \frac{N^2}{3} \cdot 2^N - \frac{N^2}{3}\,, \\
S_{4b} &= N \cdot \frac{4}{9} \cdot \left[(\frac{N}{2} - 1) \cdot 4^{\frac{N}{2}} - \frac{N}{2} \cdot 4^{\frac{N}{2}-1} + 1 \right] \\
&= \frac{4N}{9} \cdot \left[\frac{N}{2} \cdot 2^N - 2^N - \frac{N}{8} \cdot 2^N + 1 \right] = \frac{N^2}{6} \cdot 2^N - \frac{4N}{9} \cdot 2^N + \frac{4N}{9}\,, \\
S_{4c} &= \frac{4}{27} \cdot \left[(\frac{N}{2} - 1)^2 \cdot 4^{\frac{N}{2}+1} - \left(2 \cdot (\frac{N}{2} - 1)^2 + 2 \cdot (\frac{N}{2} - 1) - 1 \right) \cdot 4^{\frac{N}{2}} \right. \\
&\qquad \left. + (\frac{N}{2})^2 \cdot 4^{\frac{N}{2}-1} - 5 \right] \\
&= \frac{4}{27} \cdot \left[2 \cdot (\frac{N^2}{4} - N + 1) \cdot 2^N - N \cdot 2^N + 2 \cdot 2^N + 2^N + \frac{N^2}{16} \cdot 2^N - 5 \right] \\
&= \frac{1}{12} \cdot N^2 \cdot 2^N - \frac{4}{9} \cdot N \cdot 2^N + \frac{20}{27} \cdot 2^N - \frac{20}{27}\,.
\end{aligned}$$

We have to subtract

$$\begin{aligned}
2^N \cdot E_N^2 &= \left[\frac{N}{2} + \frac{2}{9} - \frac{N}{3 \cdot 2^N} - \frac{2}{9 \cdot 2^N} \right]^2 \cdot 2^N \\
&= \frac{N^2}{4} \cdot 2^N + \frac{2N}{9} \cdot 2^N + \frac{4}{81} \cdot 2^N - \frac{N^2}{3} - \frac{10N}{27} - \frac{8}{81} \\
&\quad + \frac{N^2}{9 \cdot 2^N} + \frac{4N}{27 \cdot 2^N} + \frac{4}{81 \cdot 2^N}\,.
\end{aligned}$$

All this fragments together yield

$$2^N \cdot V_N = \frac{86}{81} \cdot 2^N - \frac{14N}{27} - \frac{82}{81} - \frac{N^2}{9 \cdot 2^N} - \frac{4N}{27 \cdot 2^N} - \frac{4}{81 \cdot 2^N},$$

proving the second formula of Theorem 3 for even N.

The corresponding calculation for odd N is:

$$\begin{aligned}
S_3 &= \sum_{l=1}^{\frac{N-1}{2}} l^2 \cdot 2^{2l-1} = \frac{N^2}{12} \cdot 2^N - \frac{5N}{18} \cdot 2^N + \frac{41}{108} \cdot 2^N - \frac{10}{27}\,, \\
S_{4a} &= N^2 \cdot \sum_{k=0}^{\frac{N-1}{2}} 4^k = \frac{2N^2}{3} \cdot 2^N - \frac{N^2}{3}\,, \\
S_{4b} &= N \cdot \sum_{k=0}^{\frac{N-1}{2}} k \cdot 4^k = \frac{N^2}{3} \cdot 2^N - \frac{5N}{9} \cdot 2^N + \frac{4N}{9}\,, \\
S_{4c} &= \sum_{k=0}^{\frac{N-1}{2}} k^2 \cdot 4^k = \frac{N^2}{6} \cdot 2^N - \frac{5N}{9} \cdot 2^N + \frac{41}{54} \cdot 2^N - \frac{20}{27}\,,
\end{aligned}$$

$$\begin{aligned}
2^N \cdot E_N^2 &= \left[\frac{N}{2} + \frac{5}{18} - \frac{N}{3 \cdot 2^N} - \frac{2}{9 \cdot 2^N}\right]^2 \cdot 2^N \\
&= \frac{N^2}{4} \cdot 2^N + \frac{5N}{18} \cdot 2^N + \frac{25}{324} \cdot 2^N - \frac{N^2}{3} - \frac{11N}{27} - \frac{10}{81} \\
&\quad + \frac{N^2}{9 \cdot 2^N} + \frac{4N}{27 \cdot 2^N} + \frac{4}{81 \cdot 2^N}\,.
\end{aligned}$$

Putting the fragments together we get

$$\begin{aligned}
2^N \cdot V_N &= S_3 + S_{4a} - 2 \cdot S_{4b} + S_{4c} - 2^N \cdot E_N^2 \\
&= \frac{86}{81} \cdot 2^N - \frac{13N}{27} - \frac{80}{81} - \frac{9N^2 + 12N + 4}{81 \cdot 2^N}\,.
\end{aligned}$$

This completes the proof of Theorem 3.

3.7 Linear Complexity and TURING Complexity

A **universal** TURING **machine** is able to simulate every other TURING machine by a suitable program. Let **M** be one, and let $u \in \mathbb{F}_2^n$ be a bit sequence of length n. Then the TURING-KOLMOGOROV-CHAITIN **(TKC) complexity** $\chi(u)$ is the length of the shortest program of **M** that outputs u. There is always one such program of length n: Simply take u as input sequence and output it unchanged. (Informally: Move the input tape forward by n steps and stop.)

Remark The function $\chi\colon \mathbb{F}_2^* \longrightarrow \mathbb{N}$ itself is not computable. This means there is no TURING machine that computes χ. Thus the TKC complexity is of low practical value as a measure of complexity. However in the recent years it gained some momentum in a more precise form by the work of VITANYI and others, see for example:

> Ming LI, Paul VITANYI: *An Introduction to Kolmogorov Complexity and Its Applications.* Springer, New York 1993, 1997.

A central result of the theory is:

$$\frac{1}{2^n} \cdot \#\{u \in \mathbb{F}_2^n \mid \chi(u) > n \cdot (1-\varepsilon)\} > 1 - \frac{1}{2^{n\varepsilon-1}}.$$

This result says that almost all sequences have a TKC complexity near the maximum value, there is no significantly shorter description of a sequence than to simply write it down. A common interpretation of this result is: "Almost all sequences are random." This corresponds quite well with the intuitive idea of randomness. Nobody would consider a sequence with a short description such as "alternate one million times between 0 and 1" as random.

> Thomas BETH, Zong-Duo DAI: On the complexity of pseudo-random sequences – or: If you can describe a sequence it can't be random. EUROCRYPT 89, 533–543.

This paper contains some small errors that are corrected in [9].

Also "linear complexity" λ measures complexity, using a quite special machine model: the LFSR. On first sight it suffers from severe deficits. The sequence "999999 times 0, then a single 1" has a very low TKC complexity—corresponding to a very low intuitive randomness—, but the linear complexity is 1 million.

Of course we could also try to use nonlinear FSRs for measuring complexity, see for instance the papers:

- Agnes Hui CHAN, Richard A. GAMES: On the quadratic span of periodic sequences. CRYPTO 89, 82–89.

- Cees J. A. JANSEN, Dick E. BOEKEE: The shortest feedback shift register that can generate a given sequence. CRYPTO 89, 90–96.

and Appendix B. However, as we saw, *a short description by a nonlinear FSR also implies a small linear complexity.*

In any case linear complexity has the advantage of easy explicite computability, and "in general" it characterizes the randomness of a bit sequence very well. This vague statement admits a surprisingly precise wording (stated here without proof). To make a fair comparision note that the description of a sequence by an LFSR needs $2 \times \lambda$ bits: the taps of the register and the starting value. Therefore we should compare χ and $2 \cdot \lambda$:

Proposition 12 (BETH/DAI)

$$\frac{1}{2^n} \cdot \#\{u \in \mathbb{F}_2^n \mid (1-\varepsilon)2\lambda(u) \leq \chi(u)\} \geq 1 - \frac{8}{3 \cdot 2^{\frac{n\varepsilon}{2-\varepsilon}}},$$

$$\frac{1}{2^n} \cdot \#\{u \in \mathbb{F}_2^n \mid (1-\varepsilon)\chi(u) \leq 2\lambda(u)\} \geq 1 - \frac{1}{3} \cdot \frac{1}{2^{n\varepsilon-(1-\varepsilon)(1+\log n)+1}} - \frac{1}{3} \cdot \frac{1}{2^n}.$$

We interpret this as: "For almost all bit sequences the linear complexity and the TKC complexity coincide with only a negligeable discrepancy (up to the obvious factor 2)."

This result confirms that linear complexity—despite its simplicity—is a useful measure of complexity, and that in general bit sequences of high linear complexity have no short description in other machine models. Thus they are cryptographically useful. Every efficient prediction method—in the sense of cryptanalysis of bitstream ciphers—would provide a short description in the sense of TKC complexity. And conversely: If a sequence has a short description, then we even can generate it by a short LFSR. Thus we may summarize: *In general a bit sequence of high linear complexity is not efficiently predictable.*

Note that these results

- are "asymptotic" in character. For the "bounded" world we live in they only yield qualitative statements—a standard phenomen for results on cryptographic security.

- concern probabilities only. There might be $2^r \ll 2^n$ sequences of small TKC complexity that however have high linear complexity—*relatively* very few, but *absolutely* quite a lot! In Chapter 4 we'll construct such sequences, dependent on secret parameters, and show (up to one of the usual hardness assumptions for mathematical problems) that they don't allow an efficient prediction algorithm, in particular not by a "short" LFSR.

3.8 Approaches to Nonlinearity for Feedback Shift Registers

LFSRs are popular—in particular among electrical engineers and military—for several reasons:

- very easy implementation,
- extreme efficiency in hardware,
- good qualification as random generators for statistical applications and simulations,
- unproblematic operation in parallel even in large quantities.

But unfortunately from a cryptological view they are completely insecure if used naively. To capitalize their positive properties while escaping their cryptological weakness there are several approaches.

Approach 1, Nonlinear Feedback

Nonlinear feedback follows the scheme from Figure 1.7 with a nonlinear Boolean function f. There is a general proof that in realistic use cases NLFSRs are cryptographically useless if used in the direct naive way [6]. We won't pursue this approach here.

Approach 2, Nonlinear Output Filter

The nonlinear ouput filter (nonlinear feedforward) realizes the scheme from Figure 3.7. The shift register itself is linear, the Boolean function f, nonlinear.

The nonlinear ouput filter is a special case of a nonlinear combiner.

Approach 3, Nonlinear Combiner

The nonlinear combiner uses a "battery" of n LFSRs—preferably of different lengths—operated in parallel. The output sequences of the LFSRs serve as input of a Boolean function $f\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2$, see Figure 3.8. (Sometimes also called "nonlinear feedforward.") We'll see in Section 3.9 how to cryptanalyze this random generator.

Approach 4, Output Selection/Decimation/Clocking

There are different ways of controlling a battery of n parallel LFSRs by another LFSR:

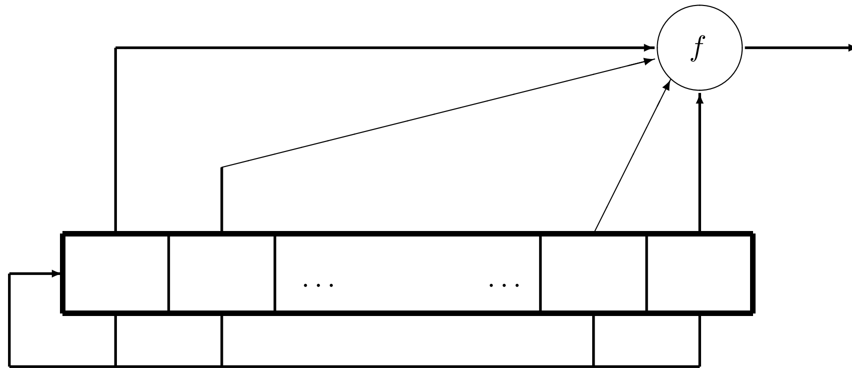


Figure 3.7: Nonlinear ouput filter for an LFSR

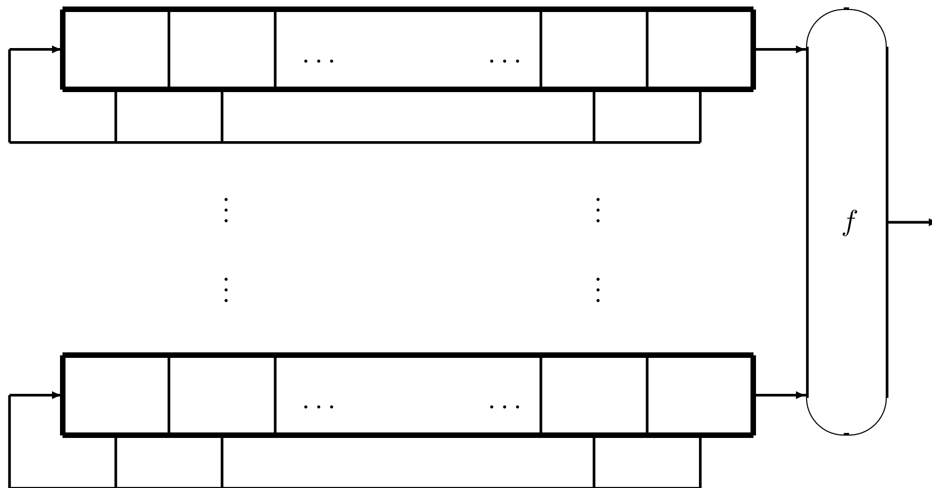


Figure 3.8: Nonlinear combiner

- **Output selection** takes the current output bit of exactly one of the LFSRs from the "battery", depending on the state of the auxiliary register, and outputs it as the next pseudorandom bit. More generally we could choose "r from n".
- For **decimation** one usually takes $n = 1$, and outputs the current bit of the one battery register only if the auxiliary register is in a certain state, for example its own current output is 1. Of course this kind of decimation applies to arbitrary bit sequences in an analogous way.
- For **clocking** we look at the state of the auxiliary register and depending on it decide which of the battery registers to step in the current cycle (and by how many positions), leaving the other registers in their current states (this mimics the control logic of rotor machines in classical cryptography).

These methods turn out to be special cases of nonlinear combiners if properly rewritten. Thus approach 3 represents the most important method of making the best of LFSRs.

The encryption standard A5/1 for mobile communications uses three LFSRs of lengths 19, 22 und 23, each with maximum possible period, and slightly differently clocked. It linearly (by simple binary addition) combines the three output streams. The—even weaker—algorithm A5/2 controls the clocking by an auxiliary register. Both variants can be broken on a standard PC in real-time.

The Bluetooth encryption standard E_0 uses four LFSRs and combines them in a nonlinear way. This method is somewhat stronger than A5, but also too weak for real security [7].

Example: The Geffe generator

The Geffe generator provides a simple example of output selection. Its description is in Figure 3.9. The output is x, if $z = 0$, and y, if $z = 1$. Expressed by a formula:

$$\begin{aligned} u &= \begin{cases} x, & \text{if } z = 0, \\ y, & \text{if } z = 1 \end{cases} \\ &= (1-z)x + zy = x + zx + zy. \end{aligned}$$

This formula shows how to interpret the Geffe generator as a nonlinear combiner with a Boolean function $f\colon \mathbb{F}_2^3 \longrightarrow \mathbb{F}_2$ of degree 2. For later use we implement f in Sage sample 3.2.

For a concrete example we first choose three LFSRs of lengths 15, 16, 17, whose periods are $2^{15} - 1 = 32767$, $2^{16} - 1 = 65535$, and $2^{17} - 1 = 131071$. These are pairwise coprime. Combining their outputs (in each step)

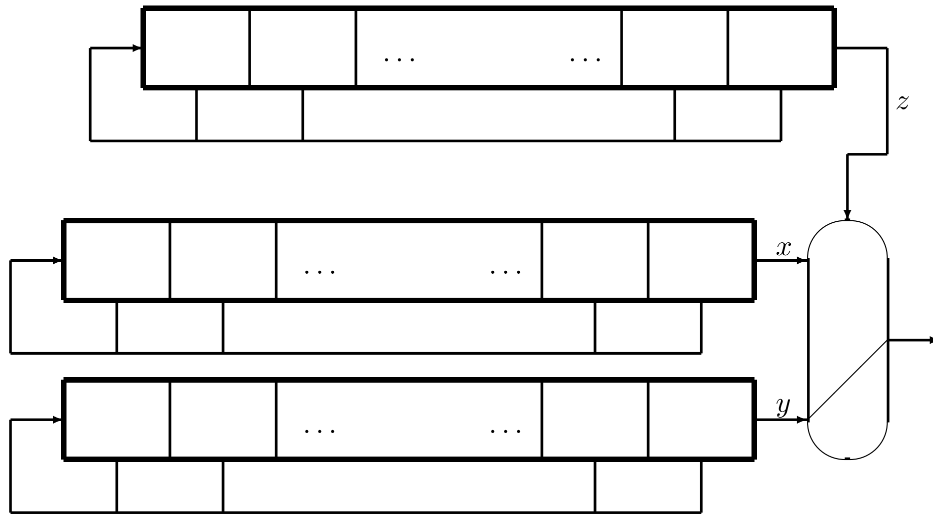


Figure 3.9: GEFFE generator

Sage Example 3.2 The Geffe function

```
sage: geff = BoolF(str2bbl("00011100"),method="ANF")
sage: geff.printTT()
Value at 000 is 0
Value at 001 is 0
Value at 010 is 0
Value at 011 is 1
Value at 100 is 1
Value at 101 is 0
Value at 110 is 1
Value at 111 is 1
```

as bitblocks of length 3 yields a sequence with a period that has an impressive length of 281459944554495, about 300×10^{12} (300 European billions, for Americans this are 300 trillions).

Register 1 recursive formula $u_n = u_{n-1} + u_{n-15}$, taps `100000000000001`, initial state `011010110001001`.

Register 2 recursive formula $u_n = u_{n-2} + u_{n-3} + u_{n-5} + u_{n-16}$, taps `0110100000000001`, initial state `0110101100010011`.

Register 3 recursive formula $u_n = u_{n-3}+u_{n-17}$, taps `00100000000000001`, initial state `01101011000100111`.

Sage sample 3.3 defines the three LFSRs. We let each of the LFSRs generate a sequence of length 100, see Sage sample 3.4.

Sage Example 3.3 Three LFSRs

```
sage: reg15 = LFSR([1,0,0,0,0,0,0,0,0,0,0,0,0,0,1])
sage: reg15.setState([0,1,1,0,1,0,1,1,0,0,0,1,0,0,1])
sage: print(reg15)
Length: 15 | Taps: 100000000000001 | State: 011010110001001
sage: reg16 = LFSR([0,1,1,0,1,0,0,0,0,0,0,0,0,0,0,1])
sage: reg16.setState([0,1,1,0,1,0,1,1,0,0,0,1,0,0,1,1])
sage: print(reg16)
Length: 16 | Taps: 0110100000000001 | State: 0110101100010011
sage: reg17 = LFSR([0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1])
sage: reg17.setState([0,1,1,0,1,0,1,1,0,0,0,1,0,0,1,1,1])
sage: print(reg17)
Length: 17 | Taps: 00100000000000001 | State: 01101011000100111
```

Sage Example 3.4 Three LFSR sequences

```
sage: nofBits = 100
sage: outlist15 = reg15.nextBits(nofBits)
sage: print(outlist15)
[1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0,
 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1,
 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0,
 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1,
 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1]
sage: outlist16 = reg16.nextBits(nofBits)
sage: print(outlist16)
[1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1,
 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1,
 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0,
 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1,
 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1]
sage: outlist17 = reg17.nextBits(nofBits)
sage: print(outlist17)
[1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1,
 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0,
 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0,
 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1,
 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0]
```

The three sequences of length 100 are:

```
10010 00110 10110 11100 00100 11011 01000 00111 01101 10000
00101 10110 11111 11001 00100 10101 01110 00111 00110 01011

11001 00011 01011 00011 00111 10000 00001 11011 10001 11000
00100 01110 11110 10010 01111 00101 10111 10010 11100 10001

11100 10001 10101 10001 00000 01100 11111 10110 11000 00111
00001 10000 00001 11111 10010 01001 01010 10110 01011 00110
```

In Sage sample 3.5 the GEFFE function combines them to the output sequence

```
11010 00111 00011 01101 00100 10011 00001 10011 10101 10000
00100 00110 11110 10010 00110 10101 00110 10011 01100 01001
```

Sage Example 3.5 The combined sequence

```
sage: outlist = []
sage: for i in range(0,nofBits):
....:     x = [outlist15[i],outlist16[i],outlist17[i]]
....:     outlist.append(geff.valueAt(x))
....:
sage: print(outlist)
[1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1,
 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1,
 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0,
 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1,
 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1]
```

3.9 Correlation Attacks—the Achilles Heels of Combiners

Let $f\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2$ be the combining function of a nonlinear combiner. The number

$$K_f := \#\{x = (x_1, \ldots, x_n) \in \mathbb{F}_2^n \mid f(x) = x_1\}$$

counts the coincidences of the value of the function with its first argument. If it is $> 2^{n-1}$, then the probability of a coincidence,

$$p = \frac{1}{2^n} \cdot K_f > \frac{1}{2},$$

is above average, and the combined output sequence "correlates" with the output of the first LFSR more then expected by random. If $p < \frac{1}{2}$, then the correlation deviates from the expected value in the other direction.

The cryptanalyst can exploit this effect in an attack with known plaintext. We suppose that she knows the "hardware", that is the taps of the registers, and also the combining function f. She seeks the initial states of all the LFSRs. We assume she knows the bits $k_0, \ldots, k_{r-1}$ of the key stream. For each of the 2^{l_1} initial states of the first LFSR she generates the sequence $u_0, \ldots, u_{r-1}$, and counts the coincidences. The expected values are

$$\frac{1}{r} \cdot \#\{i \mid u_i = k_i\} \approx \begin{cases} p & \text{for the correct initial state of LFSR 1,} \\ \frac{1}{2} & \text{otherwise.} \end{cases}$$

If r is large enough, she can determine the true initial state of LFSR 1 (with high probability) for a cost of $\sim 2^{l_1}$. She continues with the other registers, and finally identifies the complete key with a cost of $\sim 2^{l_1} + \cdots + 2^{l_n}$. Note that the cost is exponential, but significantly lower than the cost $\sim 2^{l_1} \cdots 2^{l_n}$ of the naive exhaustion of the key space.

In the language of linear cryptanalysis from Part II she made use of the linear relation

$$f(x_1, \ldots, x_n) \stackrel{p}{\approx} x_1$$

for f. Clearly she could use any linear relation as well to reduce the complexity of key search. (A more in-depth analysis of the situation leads to the notion of correlation immunity that is related with the linear potential.)

Correlations from the Geffe generator

From the truth table 3.2 we get the correlations produced by the Geffe generator. Thus the probabilities of coincidences are

$$p = \begin{cases} \frac{3}{4} & \text{for register 1 } (x), \\ \frac{3}{4} & \text{for register 2 } (y), \\ \frac{1}{2} & \text{for register 3 } (z = \text{control bit}). \end{cases}$$

x	0	0	0	0	1	1	1	1
y	0	1	0	1	0	1	0	1
z	0	0	1	1	0	0	1	1
$f(x, y, z)$	0	0	0	1	1	1	0	1

Table 3.2: Truth table of the GEFFE function

linear form	0	z	y	$y+z$	x	$x+z$	$x+y$	$x+y+z$
representation	000	001	010	011	100	101	110	111
potential λ	0	0	1/4	1/4	1/4	1/4	0	0
probability p	1/2	1/2	3/4	1/4	3/4	3/4	1/2	1/2

Table 3.3: Coincidence probabilities of the GEFFE function

A correlation attack easily detects the initial states of registers 1 and 2—the battery registers—given only a short piece of an output sequence. Afterwards exhaustion finds the initial state of register 3, the control register.

We exploit this weakness of the GEFFE generator for an attack in Sage sample 3.6 that continues Sage sample 3.2. Since we defined the linear profile for objects of the class `BoolMap` only, we first of all have to interpret the function `geff` as a Boolean map, that is a one-element list of Boolean functions. Then the linear profile is represented by a matrix of 2 columns and 8 rows. The first column `[64, 0, 0, 0, 0, 0, 0, 0]` shows the coincidences with the linear form 0 in the range. So it contains no useful information, except the denominator 64 that applies to all entries. The second row `[0, 0, 16, 16, 16, 16, 0, 0]` yields the list of coincidence probabilities p (after dividing it by 64) in Table 3.3, using the formula

$$p = \frac{1}{2} \cdot (\pm\sqrt{\lambda} + 1).$$

If $\lambda = 0$, then $p = 1/2$. If $\lambda = 1/4$, then $p = 1/4$ or $3/4$. For deciding between these two values for p we use Table 3.2.

Sage Example 3.6 Linear profile of the Geffe function

```
sage: g = BoolMap([geff])
sage: linProf = g.linProf(); linProf
[[64,0], [0,0], [0,16], [0,16], [0,16], [0,16], [0,0], [0,0]]
```

In Sage sample 3.7 we apply this finding to the 100 element sequence from Sage sample 3.5. The function `coinc` from the Sage module `Bitblock.sage` in Appendix E.1 of Part II counts the coincidences. For the first register we

find 73 coincidences, for the second one 76, for the third one only 41. This confirms the values 75, 75, 50 predicted by our theory.

Sage Example 3.7 Coincidences for the Geffe generator

```
sage: coinc(outlist15,outlist)
73
sage: coinc(outlist16,outlist)
76
sage: coinc(outlist17,outlist)
41
```

Cryptanalysis of the Geffe Generator

These results promise an effortless analysis of our sample sequence. For an assessment of the success probability we consider a bitblock $b \in \mathbb{F}_2^r$ and first ask how large is the probability that a random bitblock $u \in \mathbb{F}_2^r$ coincides with b at exactly t positions. For an answer we have to look at the symmetric binomial distribution (where $p = \frac{1}{2}$ is the probability of coincidence at a single position): The probability of exactly t coincidences is

$$B_{r,\frac{1}{2}}(t) = \frac{\binom{r}{t}}{2^r}.$$

Hence the cumulated probability of up to T coincidences is

$$\sum_{t=0}^{T} B_{r,\frac{1}{2}}(t) = \frac{1}{2^r} \cdot \sum_{t=0}^{T} \binom{r}{t}.$$

If r is not too large, then we may explicitly calculate this value for a given bound T. If on the other hand r is not too small, then we approximate the value using the normal distribution. The mean value of the number of coincidences is $r/2$, the variance, $r/4$, and the standard deviation, $\sqrt{r}/2$.

In any case for $r = 100$ the probability of finding at most (say) 65 coincidences is 0.999, the probability of surpassing this number is 1 ‰. For the initial state of register 1 we have to try $2^{15} = 32786$ possibilities (generously including the zero state $0 \in \mathbb{F}_2^{15}$ into the count). So we expect about 33 oversteppings with at least 66 coincidences. One of these should occur for the true initial state of register 1 that we expect to produce about 75 coincidences. Maybe it even produces the maximum number of coincidences.

Sage sample 3.8 shows that this really happens. However the maximum number of coincidences, 73, occurs twice in the histogram. The first occurrence happens at index 13705, corresponding to the initial state 011010110001001, the correct solution. The second occurrence, at index

Sage Example 3.8 Analysis of the Geffe generator—register 1

```
sage: clist = []
sage: histogr = [0] * (nofBits + 1)
sage: for i in range(0,2**15):
....:     start = int2bbl(i,15)
....:     reg15.setState(start)
....:     testlist = reg15.nextBits(nofBits)
....:     c = coinc(outlist,testlist)
....:     histogr[c] += 1
....:     clist.append(c)
....:
sage: print(histogr)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 12, 12, 37, 78, 116, 216,
 329, 472, 722, 1003, 1369, 1746, 1976, 2266, 2472, 2531, 2600,
 2483, 2355, 2149, 1836, 1574, 1218, 928, 726, 521, 343, 228, 164,
 102, 60, 47, 36, 13, 8, 7, 4, 2, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
sage: mm = max(clist)
sage: ix = clist.index(mm)
sage: block = int2bbl(ix,15)
sage: print("Maximum =", mm, "at index", ix, ", start value", block)
Maximum = 73 at index 13705 , start value\
 [0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1]
```

Sage Example 3.9 Analysis of the Geffe generator—continued

```
sage: ix = clist.index(mm,13706); ix
31115
sage: print(int2bbl(ix,15))
[1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1]
```

31115, see Sage sample 3.9, yields the false solution 111100110001011 that eventually leads to a contradiction.

Sage sample 3.10 shows the analogous analysis of register 2. Here the maximum of coincidences, 76, is unique, occurs at index 27411 corresponding to the initial state 0110101100010011, and provides the correct solution.

To complete the analysis we must yet determine the initial state of register 3, the control register. The obvious idea is to exhaust the 2^{17} different possibilities. There is a shortcut since we already know 51 of the first 100 bits of the control register: At a position where the values of registers 1 and

Sage Example 3.10 Analysis of the Geffe generator—register 2

```
sage: clist = []
sage: histogr = [0] * (nofBits + 1)
sage: for i in range(0,2**16):
....:     start = int2bbl(i,16)
....:     reg16.setState(start)
....:     testlist = reg16.nextBits(nofBits)
....:     c = coinc(outlist,testlist)
....:     histogr[c] += 1
....:     clist.append(c)
....:
sage: print(histogr)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 1, 0, 2, 3, 4, 8, 17, 25, 51, 92, 171,
 309, 477, 750, 1014, 1423, 1977, 2578, 3174, 3721, 4452, 4821,
 5061, 5215, 5074, 4882, 4344, 3797, 3228, 2602, 1974, 1419,
 1054, 669, 434, 306, 174, 99, 62, 38, 19, 10, 3, 0, 1, 0, 0,
 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0]
sage: mm = max(clist)
sage: ix = clist.index(mm)
sage: block = int2bbl(ix,16)
sage: print("Maximum =", mm, "at index", ix, ", start value", block)
Maximum = 76 at index 27411 , start value\
 [0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1]
```

2 differ, the control bit is necessarily 0 if the final output coincides with register 1, and 1 otherwise. Only at positions where registers 1 and 2 coincide the corresponding bit of register 3 is undetermined.

```
 register 1: 10010001101011011100001001101101000001110110110000
 register 2: 11001000110101100011001111000000001110111000111000
 register 3: -1-00--0-1101-110001---00-1-00-1--1101--110---0---
bitsequence: 11010001110001101101001001001100001100111010110000

          ... 0010110110111111100100100101010111000111001100101l
          ... 00100011101111010010011110010110111100101110010001
          ... ----110-------1-1-11-0-100----01--01-1-001-1-00-1-
          ... 00100001101111010010001101010100110100110110001001
```

In particular we already know 11 of the 17 initial bits, and are left with only $2^6 = 64$ possibilities to try.

$u_{17} = u_{14} + u_0$	$0 = 1 + u_0$	$u_0 = 1$	
$u_{19} = u_{16} + u_2$	$1 = 0 + u_2$	$u_2 = 1$	
$u_{20} = u_{17} + u_3$	$u_{20} = 0 + 0$	$u_{20} = 0$	
$u_{22} = u_{19} + u_5$	$u_{22} = u_5 + 1$	$u_5 = u_{22} + 1$	
$u_{23} = u_{20} + u_6$	$0 = u_{20} + u_6$	$u_6 = u_{20}$	$u_6 = 0$
$u_{25} = u_{22} + u_8$	$u_{25} = u_{22} + u_8$	$u_8 = u_{22} + u_{25}$	$u_8 = u_{22}$
$u_{27} = u_{24} + u_{10}$	$u_{27} = 0 + 1$	$u_{27} = 1$	
$u_{28} = u_{25} + u_{11}$	$0 = u_{25} + 0$	$u_{25} = 0$	
$u_{30} = u_{27} + u_{13}$	$u_{30} = u_{27} + u_{13}$	$u_{13} = u_{27} + u_{30}$	$u_{13} = u_{30} + 1$
$u_{33} = u_{30} + u_{16}$	$u_{33} = u_{30} + 0$	$u_{30} = u_{33}$	$u_{30} = 1$
$u_{36} = u_{33} + u_{19}$	$0 = u_{33} + 1$	$u_{33} = 1$	
$u_{39} = u_{36} + u_{22}$	$u_{39} = 0 + u_{22}$	$u_{22} = u_{39}$	
$u_{42} = u_{39} + u_{25}$	$0 = u_{39} + u_{25}$	$u_{39} = u_{25}$	$u_{39} = 0$

Table 3.4: Determination of the control register's initial state

But even this may be further simplified, since the known and the unknown bits obey linear relations of the type $u_n = u_{n-3} + u_{n-17}$. The unknown bits of the initial state are u_0, u_2, u_5, u_6, u_8, u_{13}. The solution follows the columns of Table 3.4, that immediately give

$$u_0 = 1, \ u_2 = 1, \ u_6 = 0.$$

The remaining solutions are

$$u_8 = u_{22} = u_{39} = 0, \ u_5 = u_{22} + 1 = u_8 + 1 = 1, \ u_{13} = u_{30} + 1 = 0.$$

Hence the initial state of the control register is `01101011000100111`, and we know this is the correct solution. We don't need to bother with the second possible solution for register 1 since we already found a constellation that correctly reproduces the sequence.

3.10 Design Criteria for Nonlinear Combiners

From the forgoing discussion we derive design criteria for nonlinear combiners:

- The battery registers should be as long as possible.
- The combining function f should have a low linear potential.

How long should the battery registers be? There are some algorithms for "fast" correlation attacks using the Walsh transformation, in particular against sparse linear feedback functions (that use only a small number of taps) [4]. These don't reduce the complexity class of the attack ("exponential in the length of the shortest register") but reduce the cost by a significant factor. So they are able to attack registers with up to 100 coefficients 1 in the feedback function. As a consequence

- The single LFSRs should have a length of at least 200 bits, and use about 100 taps each.

To assess the number n of LFSRs we bear in mind that the combining function should be "correlation immune", in particular have a low linear potential. A well-chosen Boolean function of 16 variables should suffice, but there are no known recommendations in the literature.

Rueppel found an elegant way out to make the correlation attack break down: Use a "time-dependent" combining function, that is a family $(f_t)_{t\in\mathbb{N}}$. The bit u_t of the key stream is calculated by the function f_t. We won't analyze this approach here.

Observing that the correlation attack needs knowledge of the taps, the security could be somewhat better if the taps are secret. Then the attacker has to perform additional exhaustions that multiply the complexity by factors such as 2^{l_1} for the first LFSR alone. This scenario allows choosing LFSRs of somewhat smaller lengths. But bear in mind that for a hardware implementation the taps are parts of the algorithm, not of the key, that is they are public parameters in the sense of Figure 2.1.

Efficiency

LFSRs and nonlinear combiners allow efficient realizations by special hardware that produces one bit per clock cycle. This rate can be enlarged by parallelization. From this point of view estimating the cost of execution on a usual PC processor is somewhat inadequate. Splitting each of the ≥ 200 bit registers into 4 parts of about 64 bits shifting a single register requires at least 4 clock cycles, summing up to 64 clock cycles for 16 registers. Add some clock cycles for the combining function. Thus one single bit would take about 100 clock cycles. A 2-GHz processor, even with optimized implementation, would produce at most $2 \cdot 10^9/100 = 20$ million bits per second.

As a summary we note:

Using LFSRs and nonlinear combining functions we can build useful and fast random generators, especially in hardware.

Unfortunately there is no satisfying theory for the cryptologic security of this type of random generators, even less a mathematical proof. Security is assessed by plausible criteria that—as for bitblock ciphers—are related to the nonlinearity of Boolean functions.

Chapter 4

Perfect Pseudorandom Generators

As we saw the essential cryptologic criterion for pseudorandom generators is unpredictability. In the 1980s cryptographers, guided by an analogy with asymmetric cryptography, found a way of modelling this property in terms of complexity theory: Prediction should boil down to a known "hard" algorithmic problem such as factoring integers or discrete logarithm. This idea established a new quality standard for pseudorandom generators, much stronger than statistical tests, but eventually building on unproven mathematical hypotheses. Thus the situation with respect to the security of pseudorandom generators is comparable to asymmetric encryption.

As an interesting twist it soon turned out that in a certain sense unpredictability is a universal property: For an unpredictable sequence there is *no efficient algorithm at all* that distinguishes it from a true random sequence, a seemingly much stronger requirement. See Theorem 4 (YAO's theorem). This universality justifies the denomination "perfect" for the corresponding pseudorandom generators. In particular there is no efficient statistical test that is able to distinguish the output of a perfect pseudorandom generator from a true random sequence. Thus, on the theoretical side, we have a very appropriate model for pseudorandom generators that are absolutely strong from a statistical viewpoint, and invulnerable from a cryptological viewpoint. In other words:

> *Perfect pseudorandom generators are cryptographically secure and statistically undistinguishable from true random sources—and are fit for* any *efficient application that needs random input.*
>
> *Presumably perfect pseudorandom generators exist, but there is no complete mathematical proof ot their existence.*

The first concrete approaches to the construction of perfect pseudorandom generators yielded algorithms that were too slow for most practical uses

(given the then current CPUs), the best known being the BBS generator (for Lenore Blum, Manuel Blum, Michael Shub). But modified approaches soon provided pseudorandom generators that are passably fast und nevertheless (presumably) cryptographically secure.

Looking back at Section 3.7 we might stumble over the apparent contradiction with the general "rule": "If a sequence has a short description (as the BBS sequence obviously has!), then it can't be random and even has a short description by a linear feedback shift register." In particular this would yield an efficient algorithm that distinguishes it from a random sequence. However as already stated in 3.7 this rule leaves a small loophole—small in relative terms but maybe wide enough in absolute terms. The notion of pseudorandomness tries to slip through this loophole, see Figure 4.1: Pseudorandom sequences

- are not random because they have a short description,
- can't nevertheless be efficiently predicted, or distinguished from random sequences.

For some more theoretical framework see the book [1].

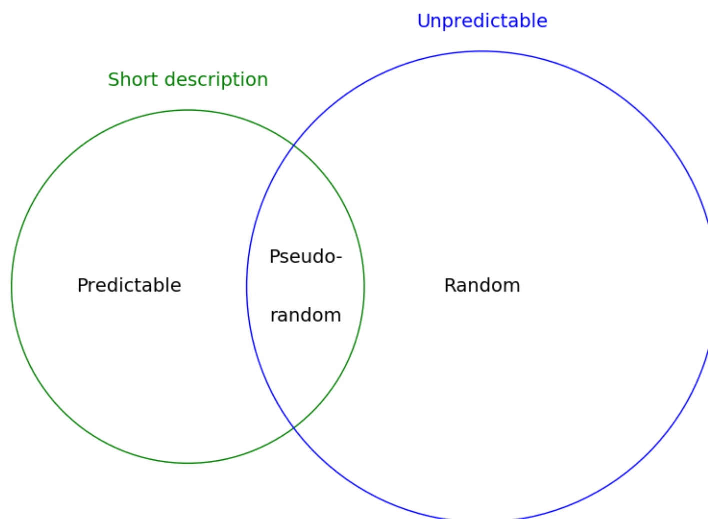


Figure 4.1: "Pseudorandom" as intersection of "Short description" and "Unpredictable"—the size of the areas is by far not to scale.

In the literature we find many tests for randomness:

- Marsaglia's diehard test suite, a collection of statistical tests that check the fitness of a sequence for statistical, but not cryptological applications,
- Golomb's postulates, see Section 1.10 above,

- the linear complexity profile, see Chapter 3 above,
- MAURER's universal test, see [5, 5.4.5],
- the LIL test, see [9].

By definition a perfect pseudorandom generator will pass all these tests. However in practice, since perfectness is an "asymptotic property" only, applying these tests can't harm.

4.1 The BBS Generator

As with the RSA cipher we consider an integer module n that is a product of two large prime numbers. For the BBS generator we choose a Blum **integer**, preferably—but not necessarily—a special or even superspecial one.

> Choosing n superspecial ensures that the sequence of states has a huge period. See the discussion in the Appendices 12–14 of Part III. However the following security proof doesn't depend on this property.

The BBS generator works in the following way that easily fits into the general framework of Figure 2.1: As a first step choose two large random Blum primes p and q, and form their product $n = pq$. The factors p and q are internal (secret) parameters, the product n may be treated as internal or external (public) parameter. As a second step choose a random integer, the "seed", s with $1 \leq s \leq n-1$, and coprime with n.

> The coprimality is efficiently tested with the Euclidean algorithm. If we catch an s not coprime with n, we have factorized n by hazard. This might happen, but is extremely unlikely, and can easily be captured at initialization time.

Then we proceed with generating a pseudorandom sequence: Take $x_0 = s \in \mathbb{M}_n$ as initial state, and form the sequence of inner states of the pseudorandom generator: $x_i = x_{i-1}^2 \bmod n$ for $i = 1, 2, 3, \ldots$ In each step output the last significant bit of the binary representation, that is $u_i = x_i \bmod 2$ for $i = 1, 2, 3 \ldots$, or in other words, the parity of x_i.

> If $x_i < \sqrt{n}$, then $x_i^2 \bmod n = x_i^2$, the integer square, so x_{i+1}^2 has the same parity as x_i. In order to avoid a constant segment at the beginning of the output, often the boundary areas $s < \sqrt{n}$, as well as $s > n - \sqrt{n}$, are excluded. However if we really choose s as a true random value, the probability for s falling into these boundary areas is extremely low. But to be on the safe side we may require $\sqrt{n} \leq s \leq n - \sqrt{n}$.
>
> If the seed s happens to be a quadratic non-residue, the sequence of inner states (the BBS sequence) has a preperiod of length 1.

Example

Of course an example with small numbers is practically irrelevant, but it illustrates the algorithm: Take $p = 7$, $q = 11$, $n = 77$, $s = 53$. Then $s^2 = 2809$, hence $x_1 = 37$, and $u_1 = 1$ since x_1 is odd. The following table shows the beginning of the sequence of states:

i	1	2	3	4	...
x_i	37	60	58	53	...
u_i	1	0	0	1	...

Since $x_4 = 53 = s$ the seed s happens to be a quadratic residue, and the BBS sequence has period 4. Therefore the output "pseudorandom" sequence (u_i) a forteriori has period 4.

Treating the primes p and q as secret is essential for the security of the BBS generator. They serve for forming n only, afterwards they may even be destroyed—in contrast with RSA there is no further use for them (except when you use SageMath, see below). Likewise all the non-output bits of the inner states x_i must be secret. Moreover there is no reason to reveal the product $n = pq$ even if the following security proof doesn't depend on the nondisclosure of n.

SageMath has an implementation of the BBS generator via the methods `random_blum_prime()` and `blum_blum_shub()`. The code sample 4.1 shows how to use them.

Sage Example 4.1 Generating a pseudorandom bit sequence by the BBS generator

```
sage: from sage.crypto.util import random_blum_prime
sage: from sage.crypto.stream import blum_blum_shub
sage: p = random_blum_prime(2^511, 2^512)
sage: q = random_blum_prime(2^511, 2^512)
sage: s = 11.powermod(248,p*q)  # a (not so random) example
sage: prseq = blum_blum_shub(1024,s,p,q)
```

Table 4.1 shows a BLUM integer with 309 decimal places (or 1024 bits) that was an intermediate result of this program. Considering the progress of factoring algorithms we better should use BLUM integers of at least 2048 bits.

```
 4506 15286 74466 50249 26225 14044 26383 22616 74480 10227
69340 10344 80414 96318 08671 21639 63710 30387 17602 25696
53909 02080 09976 45161 76261 91025 59480 62175 49124 86394
40823 70452 14981 62658 94574 67753 74945 83135 16199 61782
07594 51105 16833 44889 30109 66289 10763 64987 90309 41852
27681 66632 02722 32988 57145 85172 07427 89442 30004 31819
83739 34537
```

Table 4.1: A 1024 bit Blum integer

Table 4.2 shows the resulting bitsequence. Be warned that the SageMath

output is of type `StringMonoidElement`. For further use in a stream cipher it might be necessary to convert it to a bitblock or bitstring.

```
1000 1111 1001 0101 1001 0111 0011 0100 0010 1000 1100 0001
1010 0101 1110 1001 1010 1001 0110 0010 1010 1010 0111 0111
1000 1010 1000 1101 1111 1101 1010 1100 1100 0001 0101 1001
0111 1111 0001 0100 1010 0000 1100 1010 0101 1000 1110 0000
0001 1011 0100 0100 1010 0010 1010 1010 0110 1001 0111 1100
1011 0010 0011 0100 1101 1001 0101 0100 0111 0100 0010 0111
1101 1000 0010 0111 1000 0110 1110 0111 1110 1101 0110 1000
0001 0011 1111 0011 0011 0101 0001 0001 1010 0110 0101 1000
1010 1100 1011 0011 1111 1000 1001 0100 0001 1110 1111 1111
1001 0000 0010 0000 0111 0111 1001 0001 1111 0100 1010 0011
1000 0111 1100 0000 1011 0110 1011 1010 0111 0100 1110 1001
1001 0101 0011 1000 0010 0011 1010 1001 1100 0010 1111 1001
1010 1001 0110 0011 1001 0100 1000 1111 1001 1001 0010 1000
0111 0110 1101 0011 0110 0010 1110 0010 0000 1100 1011 1111
0011 0010 0110 1110 1000 1000 1110 1110 0011 0010 0100 0100
1101 1000 0011 0010 1000 1110 1000 1101 1010 0001 0011 1100
1001 0110 1010 0000 0000 0000 1011 0111 1010 0010 1100 1010
0100 0010 0010 0010 0010 1011 0100 0000 1100 1010 1101 0000
1101 1111 0011 0001 1000 0000 0111 0111 1110 1111 0011 1011
1111 0001 0010 1000 0110 1011 0111 0011 1111 1011 0101 0100
0110 1111 1111 0011 1011 0000 1010 0010 1100 0010 1001 0101
1110 1001 1001 1001
```

Table 4.2: 1024 “perfect” pseudorandom bits. Note that generating 1024 pseudorandom bits from a 1024-bit random integer isn’t worth the effort. However we could continue this sequence much further and generate, say, 2^{30} pseudorandom bits.

Figure 4.2 gives an optical impression of the randomness of this sequence, and Figure 4.3, of its linearity profile.

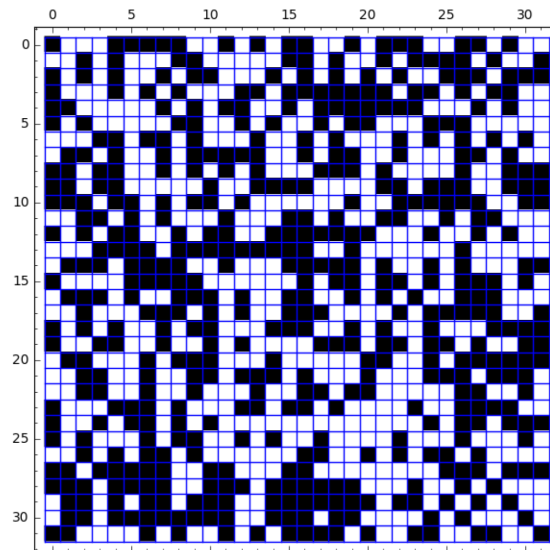


Figure 4.2: Visualization of a “perfect” pseudorandom sequence

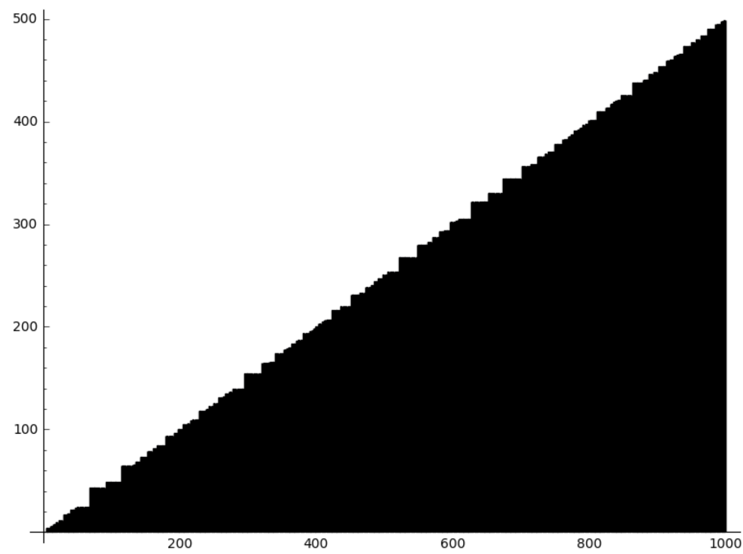


Figure 4.3: Linearity profile of a “perfect” pseudorandom sequence

4.2 The BBS Generator and Quadratic Residuosity

Given a seed $s \in \mathbb{M}_n^+$ the BBS generator outputs a bit sequence $(b_1(s), \ldots, b_r(s))$—by the way the same sequence as the seed $s' = \sqrt{s^2} \bmod n$ that is a quadratic residue. A probabilistic circuit (see Appendix B of Part III)

$$C\colon \mathbb{F}_2^r \times \Omega \longrightarrow \mathbb{F}_2$$

has an ε-advantage for **BBS extrapolation** with respect to n if

$$P(\{(s,\omega) \in \mathbb{M}_n \times \Omega \mid C(b_1(s), \ldots, b_r(s), \omega) = \operatorname{lsb}(\sqrt{s^2} \bmod n)\}) \geq \frac{1}{2} + \varepsilon.$$

In other words: The algorithm implemented by C "predicts" (or extrapolates) the bit preceding a given subsequence with ε-advantage.

> If we seed the generator with a quadratic residue s, then C outputs the parity of s (with ε-advantage). If fed with a later segment $(b_{i+1}, \ldots, b_{i+r})$ (with $i \geq 1$) of a BBS output C extrapolates the preceding bit b_i.

In the following lemmas and proposition let τ_t be the maximum expense of the operation $xy \bmod n$ where n is a t-bit integer and $0 \leq x, y < n$. We know that $\tau_t = \mathrm{O}(t^2)$ (and even know an exact upper bound for the circuit size).

Lemma 16 *Let n be a* Blum *integer $< 2^t$. Assume the probabilistic circuit $C\colon \mathbb{F}_2^r \times \Omega \longrightarrow \mathbb{F}_2$ has an ε-advantage for BBS extrapolation with respect to n. Then there is a probabilistic circuit $C' : \mathbb{F}_2^t \times \Omega \longrightarrow \mathbb{F}_2$ of size $\#C' \leq \#C + r\tau_t + 4$ that has an ε-advantage for deciding quadratic residuosity for $x \in \mathbb{M}_n^+$.*

Proof. First we compute the BBS sequence $(b_1, \ldots, b_r)$ for the seed $s \in \mathbb{M}_n^+$ at an expense of $r\tau_t$. Then C computes the bit $\operatorname{lsb}(\sqrt{s^2} \bmod n)$ with advantage ε. Therefore setting

$$C'(s,\omega) := \begin{cases} 1 & \text{if } C(b_1, \ldots, b_r, \omega) = \operatorname{lsb}(s), \\ 0 & \text{otherwise,} \end{cases}$$

we decide the quadratic residuosity of s with ε-advantage by the corollary of Proposition 24 in Appendix A.11 of Part III. The additional costs for comparing bits are at most 4 additional nodes in the circuit. ◇

Now let C: $\mathbb{F}_2^t \times \Omega \longrightarrow \mathbb{F}_2$ be an arbitrary probabilistic circuit. Then for $m \geq 1$ we define the m**-fold circuit** by

$$C^{(m)}\colon \mathbb{F}_2^t \times \Omega^m \longrightarrow \mathbb{F}_2,$$

$$C^{(m)}(s,\omega_1,\ldots,\omega_m) := \begin{cases} 1 & \text{if } \#\{i \mid C(s,\omega_i) = 1\} \geq \frac{m}{2}, \\ 0 & \text{otherwise.} \end{cases}$$

So this circuit represents the "majority decision". Its implementation consists of m parallel copies of C, one integer addition of m bits, and one comparision of $\lceil {}^2\log m \rceil$-bit integers, hence by Appendix B.3 of Part III its size is

$$\#C^{(m)} \leq r \cdot \#C + 2m^2.$$

Lemma 17 (Amplification of advantage) *Let $A \subseteq \mathbb{F}_2^t$, and let C be a circuit that computes the Boolean function $f : A \longrightarrow \mathbb{F}_2$ with an ε-advantage. Let $m = 2h + 1$ be odd.*

Then $C^{(m)}$ computes the function f with an error probability of

$$\leq \frac{(1 - 4\varepsilon^2)^h}{2}.$$

For each $\delta > 0$ there is an

$$m \leq 3 + \frac{1}{2\delta\varepsilon^2}$$

such that $C^{(m)}$ computes the function f with an error probability δ.

Proof. The probability that C gives a correct answer is

$$p := P(\{(s,\omega) \in A \times \Omega \mid C(s,\omega) = f(s)\}) \geq \frac{1}{2} + \varepsilon.$$

Since enlarging ε tightens the assertion we may assume that $p = \frac{1}{2} + \varepsilon$. The complementary value $q := 1 - p = \frac{1}{2} - \varepsilon$ equals the probability that C gives a wrong answer. Hence the probability of getting exactly k correct answers from m independent invocations of C is $\binom{m}{k} p^k q^{m-k}$. Thus the error probability we search is

$$\begin{aligned}
&P(\{(s,\omega_1,\ldots,\omega_m) \in A \times \Omega^m \mid C^{(m)}(s,\omega_1,\ldots,\omega_m) = f(s)\}) \\
&= \sum_{k=0}^{h} \binom{m}{k} (\frac{1}{2} + \varepsilon)^k (\frac{1}{2} - \varepsilon)^{m-k} \\
&= (\frac{1}{2} + \varepsilon)^h (\frac{1}{2} - \varepsilon)^{h+1} \cdot \sum_{k=0}^{h} \binom{m}{k} (\frac{1}{2} + \varepsilon)^{k-h} (\frac{1}{2} - \varepsilon)^{h-k} \\
&= (\frac{1}{4} - \varepsilon^2)^h \cdot (\frac{1}{2} - \varepsilon) \cdot \underbrace{\sum_{k=0}^{h} \binom{m}{k} \underbrace{\left(\frac{\frac{1}{2} - \varepsilon}{\frac{1}{2} + \varepsilon}\right)^{h-k}}_{\leq 1}}_{\leq 2^{m-1} = 4^h} \\
&\leq (1 - 4\varepsilon^2)^h
\end{aligned}$$

which proves the first statement.

For an error probability δ a sufficient condition is:

$$\begin{aligned} (1-4\varepsilon^2)^h &\leq 2\delta, \\ h \cdot \ln(1-4\varepsilon^2) &\leq \ln 2 + \ln \delta, \\ h &\geq \frac{\ln 2 + \ln \delta}{\ln(1-4\varepsilon^2)}. \end{aligned}$$

Therefore we choose

$$h := \left\lceil \frac{\ln 2 + \ln \delta}{\ln(1-4\varepsilon^2)} \right\rceil. \tag{1}$$

Then the error probability of $C^{(m)}$ is at most δ, and

$$\begin{aligned} h &\leq 1 + \frac{\ln 2 + \ln \delta}{\ln(1-4\varepsilon^2)} = 1 + \frac{\ln \frac{1}{\delta} - \ln 2}{\ln \frac{1}{1-4\varepsilon^2}} \\ &\leq 1 + \frac{\frac{1}{\delta} - 1 - \ln 2}{4\varepsilon^2} \leq 1 + \frac{1}{4\delta\varepsilon^2}, \end{aligned}$$

proving the second statement. $\diamond$

By the way the size of $C^{(m)}$ is

$$\#C^{(m)} \leq \left[3 + \frac{1}{2\delta\varepsilon^2}\right] \cdot \#C + 2 \cdot \left[3 + \frac{1}{2\delta\varepsilon^2}\right]^2.$$

Merging the two lemmas we get:

Proposition 13 *Let n be a* Blum *integer $< 2^t$. Assume the probabilistic circuit $C : \mathbb{F}_2^r \times \Omega \longrightarrow \mathbb{F}_2$ has an ε-advantage for BBS extrapolation with respect to n. Then for each $\delta > 0$ there is a probabilistic circuit $C' : \mathbb{F}_2^t \times \Omega' \longrightarrow \mathbb{F}_2$ that decides quadratic residuosity in $\mathbb{M}_n^+$ with error probability δ and has size*

$$\#C' \leq \left[3 + \frac{1}{2\delta\varepsilon^2}\right] \cdot [\#C + r\tau_t + 4] + 2 \cdot \left[3 + \frac{1}{2\delta\varepsilon^2}\right]^2.$$

Note that the size of C' is polynomial in r, $\#C$, $\frac{1}{\delta}$, $\frac{1}{\varepsilon}$, and t, and we even could make this polynomial explicit. Thus:

From an efficient probabilistic BBS extrapolation algorithm for the module n with ε-advantage we can construct an efficient probabilistic decision algorithm for quadratic residuosity for n with arbitrary small error probability.

This complexity bound becomes even more perspicuous, when we specify dependencies from the input complexity, measured by the bit size t. Thus we choose

- $r \leq f(t)$ with a polynomial $f \in \mathbb{Q}[T]$ (that is we generate only "polynomially many" pseudorandom bits),
- $\frac{1}{\delta} \leq g(t)$ (or $\delta \geq 1/g(t)$) with a polynomial $g \in \mathbb{Q}[T]$ (that is we don't choose δ "too small", not like an ambitious $\delta < 1/2^t$),
- $\frac{1}{\varepsilon} \leq h(t)$ (or $\varepsilon \geq 1/h(t)$) with a polynomial $h \in \mathbb{Q}[T]$ (that is ε is reasonably small, not only like a modest $\varepsilon \approx 1/\log(t)$).

Then

$$\begin{aligned} \#C' &\leq \left[3 + \frac{1}{2}\, g(t)\, h(t)^2\right] \cdot [\#C + f(t)\,\tau_t + 4] + 2 \cdot \left[3 + \frac{1}{2}\, g(t)\, h(t)^2\right]^2 \\ &\leq \Phi(t) \cdot \#C + \Psi(t) \end{aligned}$$

with polynomials Φ, $\Psi \in \mathbb{Q}[t]$. In the following section we'll see how this statement makes BBS a "perfect" pseudorandom generator.

The hypothetical decision algorithm for $s \in \mathbb{M}_n^+$ from Proposition 13 runs like this (assuming that n is a public parameter):

1. Construct the BBS-sequence $b_1(s), \ldots, b_r(s)$ (using the public parameter n).
2. Choose the desired error probability δ.
3. Choose $m = 2h + 1$ with h as in Equation 1.
4. Choose random elements $\omega_1, \ldots, \omega_m \in \Omega$ and determine $b_i = C(s, \omega_i) \in \mathbb{F}_2$ for $i = 1, \ldots, r$.
5. Count $z = \#\{i \mid b_i = \mathrm{lsb}(s)\}$.
6. If $z \geq m/2$ output 1 ("quadratic residue"), else output 0 ("quadratic nonresidue").

4.3 Perfect Pseudorandom Generators

A sound definition of the concept "pseudorandom generator" is overdue. Informally we define it as an efficient algorithm that takes a "short" bitstring $s \in \mathbb{F}_2^n$ and converts it into a "long" bitstring $s \in \mathbb{F}_2^r$, compare Appendix A.2.

The terminology of complexity theory as in Appendix B of Part III allows us to give a mathematically exact (but not completey satisfying from a practical point of view) definition by considering parameter-dependent families of Boolean maps (or circuits) $G_n\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^{r(n)}$, and analyzing their behaviour when the parameter n grows to infinity. Such an algorithm—represented by the family $(G_n)_{n\in\mathbb{N}}$ of Boolean circuits—can be efficient only if the "expanding function" $r\colon \mathbb{N} \longrightarrow \mathbb{N}$ grows at most polynomially with the parameter n, otherwise even writing down the output sequence in an efficient way is impossible. We measure the complexity in a meaningful way by the size of the circuit (or by counting the number of needed bit operations) that likewise must grow at most polynomially with n.

To make this idea more precise we consider an infinite parameter set $M \subseteq \mathbb{N}$. We assume that an instance of the generator is defined for each $m \in M$. As an example think of M as a set of Blum integers. Let $M_n = M \cap [2^{n-1} \dots 2^n[$ be the set of n-bit integers in M.

A **pseudorandom generator with parameter set** M **and expansion function** r is a family $G = (G_m)_{m\in M}$ of Boolean circuits

$$G_m\colon A_m \longrightarrow \mathbb{F}_2^{r(n)} \quad \text{with } A_m \subseteq \mathbb{F}_2^n,$$

where n is the bitlength of m, such that there exists a (deterministic) polynomial family of circuits $\tilde{G} = (\tilde{G}_n)_{n\in\mathbb{N}}$, where $\tilde{G}_n$ has $2n$ deterministic input nodes, with $\tilde{G}_n(m, x) = G_m(x)$. (In other words: The pseudorandom bits are efficiently computable. In particular the function r is bounded by a polynomial in n.) A_m is called the set of seeds for the parameter m. Thus each G_m expands an n-bit sequence $x \in A_m$ to a $r(n)$-bit sequence $G_m(x) \in \mathbb{F}_2^{r(n)}$.

To see how the BBS generator fits into this definition let M be the set of Blum integers or an infinite subset of it, $A_m = \mathbb{M}_m$, and $G_m(x) = (b_1(x), \dots, b_{r(n)}(x))$ be the corresponding BBS sequence, $b_i(x) = \mathrm{lsb}(x_i)$ where $x_0 = x$, $x_i = x_{i-1}^2 \bmod m$, for $m \in M$.

A **polynomial test** for the pseudorandom generator G is a polynomial family of (probabilistic) circuits $C = (C_n)_{n\in\mathbb{N}}$,

$$C_n : \mathbb{F}_2^n \times \mathbb{F}_2^{r(n)} \times \Omega_n \longrightarrow \mathbb{F}_2$$

over a probability space $\Omega_n \subseteq \mathbb{F}_2^{s(n)}$ where $s(n)$ is the number of probabilistic inputs of C_n. Thus the test C_n may depend on the parameter m. The probability that the test computes the value 1 for a sequence generated by G is

$$p(G, C, m) = P\{(x, \omega) \in A_m \times \Omega_n \mid C_n(m, G_m(x), \omega) = 1\}.$$

The probability that the test computes the value 1 for an arbitrary ("true random") sequence of the same length is

$$\bar{p}(C,m) = P\{(u,\omega) \in \mathbb{F}_2^{r(n)} \times \Omega_n \mid C_n(m,u,\omega) = 1\}.$$

Ideally (for a "good" generator) these two probabilities should agree approximately: the test should not be an ε-distinguisher for reasonable values of ε and for almost all parameters m. We say the pseudorandom generator G **passes the test** C if for all non-constant polynomials $h \in \mathbb{N}[X]$ the set A of $m \in M$ with

$$|p(G,C,m) - \bar{p}(C,m)| \geq \frac{1}{h(n)}$$

is sparse in M (the set of parameters m for which C is a $1/h(n)$-distinguisher).

Recall from Appendix B.7 of Part III that this means that

$$\frac{\#(A \cap M_n)}{\#M_n} \leq \frac{1}{\eta(n)} \quad \text{for almost all } n \in \mathbb{N}$$

for each non-constant polynomial $\eta \in \mathbb{N}[X]$.

The pseudorandom generator G is called **perfect** if it passes all polynomial tests. In sloppy words:

No efficient statistical test (or algorithm) is able to distinguish a bit sequence generated by G from a "true random" bit sequence.

4.4 YAO's Criterion

At first sight trying to prove the perfectness of a pseudorandom generator G seems hopeless. How to manage "all polynomial tests"? But surprisingly a seemingly much weaker test is sufficient. Let $G_m(x) = (b_1^{(m)}(x), \ldots, b_{r(n)}^{(m)}(x))$ be the bit sequence generated by G_m from the seed x. Let $C = (C_n)_{n\in\mathbb{N}}$ be a polynomial family of circuits,

$$C_n : \mathbb{F}_2^n \times \mathbb{F}_2^{i_n} \times \Omega_n \longrightarrow \mathbb{F}_2$$

with $0 \leq i_n \leq r(n) - 1$, and let $h \in \mathbb{N}[X]$ be a non-constant polynomial. Then we say that C has a $\frac{1}{h}$-advantage for extrapolating G if the set of parameters $m \in M$ with

$$P\{(x, \omega) \in A_m \times \Omega_n \mid C_n(m, b_{j_m+1}^{(m)}(x), \ldots, b_{j_m+i_n}^{(m)}(x), \omega) = b_{j_m}^{(m)}(x)\}$$

$$\geq \frac{1}{2} + \frac{1}{h(n)} \qquad (2)$$

for an index j_m, $1 \leq j_m \leq r(n) - i_n$, is not sparse in M. In other words given a subsequence C extrapolates the preceding bit with a small advantage in sufficiently many cases. We say that G passes the **extrapolation test** if there exists no such polynomial family of circuits with a $\frac{1}{h}$-advantage for extrapolating G for any polynomial $h \in \mathbb{N}[X]$.

For instance the linear congruential generator fails the extrapolation test, as does a linear feedback shift register.

Theorem 4 [YAO's criterion] *The following statements are equivalent for a pseudorandom generator G:*

(i) *G is perfect.*

(ii) *G passes the extrapolation test.*

Proof. "(i) $\Longrightarrow$ (ii)": Assume G fails the extrapolation test. Then there is a polynomial family C of circuits that has a $\frac{1}{h}$-advantage for extrapolating G. Let $A \subseteq M$ be the non-sparse set of parameters for which the inequality (2) holds. We construct a polynomial test $C' = (C'_n)_{n\in\mathbb{N}}$:

$$C'_n(m, u, \omega) = C_n(m, u_{j_m+1}, \ldots, u_{j_m+i_n}, \omega) + u_{j_m} + 1$$

where for $m \in \mathbb{F}_2^n - A$ we set $j_m = 1$ (this value doesn't matter anyway). Hence

$$C'_n(m, u, \omega) = 1 \Longleftrightarrow C_n(m, u_{j_m+1}, \ldots, u_{j_m+i_n}, \omega) = u_{j_m}.$$

For $m \in A$ we get

$$p(G, C', m) = P\{C_n(m, b_{j_m+1}^{(m)}(x), \ldots, b_{j_m+i_n}^{(m)}(x), \omega) = b_{j_m}^{(m)}(x)\} \geq \frac{1}{2} + \frac{1}{h(n)}$$

and have to compare this value with

$$\bar{p}(C', m) = P\{C_n(m, u_{j_m+1}, \ldots, u_{j_m+i_n}, \omega) = u_{j_m}\}$$

$$= P\{C_n(\ldots) = 0 \text{ and } u_{j_m} = 0\} + P\{C_n(\ldots) = 1 \text{ and } u_{j_m} = 1\}.$$

(The sum corresponds to a decomposition into two disjoint subsets.) Each summand denotes the probability that two independent events occur simultaneously. Thus

$$\bar{p}(C', m) = \frac{1}{2}P\{C_n(\ldots) = 0\} + \frac{1}{2}P\{C_n(\ldots) = 1\} = \frac{1}{2}.$$

Hence for $m \in A$

$$p(G, C', m) - \bar{p}(C', m) \geq \frac{1}{h(n)}.$$

We conclude that G fails the test C', and therefore is not perfect.

"(ii) $\Longrightarrow$ (i)": Assume G is not perfect. Then there is a polynomial test C failed by G. Hence there is a non-constant polynomial $h \in \mathbb{N}[X]$ and a $t \in \mathbb{N}$ with

$$|p(G, C, m) - \bar{p}(C, m)| \geq \frac{1}{h(n)}$$

for m from a non-sparse subset $A \subseteq M$ with $\#A_n \geq \#M_n/n^t$ for infinitely many $n \in I$. For at least half of all $m \in A_n$ we have $p(G, C, m) > \bar{p}(C, m)$ or the inverse inequality. First we treat the first of these two cases (for fixed n).

For $k = 0, \ldots, r(n)$ let

$$p_m^k = P\{C_n(m, t_1, \ldots, t_k, b_{k+1}^{(m)}(x), \ldots, b_{r(n)}^{(m)}(x), \omega) = 1\}$$

where $t_1, \ldots, t_k \in \mathbb{F}_2$ are random bits. So we consider the probability in $A_m \times (\mathbb{F}_2^k \times \Omega_n)$. We have

$$p_m^0 = p(G, C, m), \quad p_m^{r(n)} = \bar{p}(C, m),$$

$$\frac{1}{h(n)} \leq p_m^0 - p_m^{r(n)} = \sum_{k=1}^{r(n)} (p_m^{k-1} - p_m^k)$$

for the $m \in A_n$ under consideration. Thus there is an r_m with $1 \leq r_m \leq r(n)$ such that

$$p_m^{r_m-1} - p_m^{r_m} \geq \frac{1}{r(n)h(n)}.$$

One of these values r_m occurs at least $(\#M_n/2n^t r(n))$ times, denote it by k_n.

Let $\Omega'_n = \mathbb{F}_2^{k_n} \times \Omega_n$. The polynomial family C' of circuits whose deterministic inputs are fed from $A_n \times \mathbb{F}_2^{r(n)-k_n}$, and whose probabilistic inputs from Ω'_n, is defined for this n by

$$C'_n(m, u_1, \ldots, u_{r(n)-k_n}, t_1, \ldots, t_{k_n}, \omega) = C_n(m, t, u, \omega) + t_{k_n} + 1.$$

Hence

$$C'_n(m, u, t, \omega) = t_{k_n} \Longleftrightarrow C_n(m, t, u, \omega) = 1.$$

Now

$$C'_n(m, b^{(m)}_{k_n+1}(x), \ldots, b^{(m)}_{r(n)}(x), t, \omega) = b^{(m)}_{k_n}(x)$$

$$\Longleftrightarrow \begin{cases} C_n(m, t, b^{(m)}_{k_n+1}(x), \ldots, b^{(m)}_{r(n)}(x), \omega) = 1 \quad \text{and} \quad t_{k_n} = b^{(m)}_{k_n}(x) \\ \text{or} \\ C_n(m, t, b^{(m)}_{k_n+1}(x), \ldots, b^{(m)}_{r(n)}(x), \omega) = 0 \quad \text{and} \quad t_{k_n} \neq b^{(m)}_{k_n}(x) \end{cases}$$

Both cases describe the occurence of two independent events. Therefore the probability of the second one is $\frac{1}{2}(1 - p_m^{k_n})$. The first one is equivalent with

$$C_n(m, t_1, \ldots, t_{k_n-1}, b^{(m)}_{k_n}(x), \ldots, b^{(m)}_{r(n)}(x), \omega) = 1 \quad \text{and} \quad t_{k_n} = b^{(m)}_{k_n}(x)\,.$$

Its probability is $p_m^{k_n-1}/2$. Together this gives

$$P\{C'_n(m, b^{(m)}_{k_n+1}(x), \ldots, b^{(m)}_{r(n)}(x), t, \omega) = b^{(m)}_{k_n}(x)\}$$

$$= \frac{1}{2} + \frac{1}{2}(p_m^{k_n-1} - p_m^{k_n}) \geq \frac{1}{2} + \frac{1}{2r(n)h(n)}$$

for at least $\#M_n/2n^t r(n)$ of the parameters $m \in M_n$. With $u = t + \deg(r) + 1$ this is $\geq \#M_n/n^u$ for infinitely many $n \in I$.

In the case where $p(G, C, m) < \bar{p}(C, m)$ for at least half of all $m \in A_n$ we analoguously set

$$C'_n(m, u, t, \omega) = C_n(m, t, u, \omega) + t_{k_n}.$$

Then the derivation runs along the same lines.

Therefore G fails the extrapolation test (with $i_n = r(n) - k_n$ and $j_m = k_n$). $\diamond$

By the way the proof made use of the non-uniformity of the computational model: C'_n depends on k_n, and we didn't give an algorithm that determines k_n.

4.5 The Prediction Test

The extrapolation test looks somewhat strange since it extrapolates the bit sequence in reverse direction, a clear contrast with the usual cryptanalytic procedures that try to predict *forthcoming* bits. We'll immediately remedy this quaint effect:

Let $C = (C_n)_{n\in\mathbb{N}}$ be a polynomial family of circuits,

$$C_n : \mathbb{F}_2^n \times \mathbb{F}_2^{i_n} \times \Omega_n \longrightarrow \mathbb{F}_2$$

with $0 \le i_n \le r(n)-1$, and let $h \in \mathbb{N}[X]$ be a non-constant polynomial. Then C has a $\frac{1}{h}$-advantage for predicting G if the subset of parameters $m \in M$ with

$$P\{(x,\omega) \mid C_n(m, b_1^{(m)}(x), \ldots, b_{i_n}^{(m)}(x), \omega) = b_{i_n+1}^{(m)}(x)\} \ge \frac{1}{2} + \frac{1}{h(n)}$$

is not sparse in M. The pseudorandom generator G passes the **prediction test** if no polynomial family of circuits has an advantage for predicting G. The proof of "(i) $\Longrightarrow$ (ii)" in Theorem 4 directly adapts to this situation yielding:

Corollary 1 *Every perfect pseudorandom generator passes the prediction test.*

Corollary 2 *If the quadratic residuosity conjecture is true, then the BBS generator is perfect, in particular passes the prediction test.*

Proof. Otherwise from Proposition 13 we could construct a polynomial family of circuits that decides quadratic residuosity for a non-sparse subset of Blum integers. ◇

The paper

> U. V. Vazirani, V. V. Vazirani: Efficient and secure pseudorandom number generation, Crypto 84, 193–202

contains a stronger result: *If the factoring conjecture is true, i. e. if factoring large integers is hard, then the BBS generator is perfect.*

4.6 Examples and Practical Considerations

We saw that the BBS generator is perfect under a plausible but unproven assumption, the quadratic residuosity hypothesis. However we don't know relevant concrete details, for example what parameters might be inappropriate. We know that certain initial states generate output sequences with short periods. Some examples of this effect are known, but we are far from a complete answer except for superspecial BLUM modules. However the security proof (depending on the quadratic residuosity hypothesis) doesn't require additional assumptions. Therefore we may confidently use the BBS generator with a pragmatic attitude: randomly choosing the parameters (primes and initial state) the probability of hitting "bad" values is extremely low, much lower then finding a needle in a haystack, or even in the universe.

Nevertheless some questions are crucial for getting good pseudorandom sequences from the BBS generator in an efficient way:

- How large should we choose the module m?
- How many bits can we use for a fixed module and initial state without compromising the security?

The provable results—relative to the quadratic residuosity hypothesis—are qualitative only, not quantitative. The recommendation to choose a module that escapes the known factorization methods also rests on heuristic considerations only, and doesn't seem absolutely mandatory for a module that itself is kept secret. The real quality of the pseudorandom bit sequence, be it for statistical or for cryptographic applications, can only be assessed by empirical criteria for the time being. We are confident that the danger of generating a "bad" pseudorandom sequence is extremely small, in any case negligeable, for modules that escape the presently known factorization algorithms, say at least of a length of 2048 bits, and for a true random choice of the module and the initial state.

> Émile BOREL proposed an informal ranking of negligeability of extremely small probabilities: $\leq 10^{-6}$ from a human view; $\leq 10^{-15}$ from a terrestrial view; $\leq 10^{-45}$ from a cosmic view. By choosing a sufficiently large module m for RSA or BBS we easily undercut Borel's bounds by far.

For the length of the useable output sequence we only know the qualitative criterion "at most polynomially many" that is useless in a concrete application. But even if we only use "quadratically many" bits we wouldn't hesitate to take 4 millions bits from the generator with a ≥ 2000 bit module. Should we need substantially more bits we would restart the generator with new parameters after every few millions of bits.

An additional question suggests itself: Are we allowed to output more then a single bit of the inner state in each iteration step to enhance the practical benefit of the generator? At least 2 bits?

Vazirani and Vazirani, and independently Alexi, Chor, Goldreich, and Schnorr gave a partial answer to this question, unfortunately also a qualitative one only: at least $\mathrm{O}(\log_2 \log_2 m)$ of the least significant bits are "safe". Depending on the constants that hide in the "O" we need to choose a sufficiently large module, and trust empirical experience. A common recommendation is using $\lfloor \log_2 \log_2 m \rfloor$ bits per step. Then for a module m of 2048 bits, or roughly 600 decimal places, we can use 11 bits per step. Calculating $x^2 \bmod m$ for a n bit number m takes $(\frac{n}{64})^2$ multiplications of 64-bit integers and subsequently the same number of divisions of the type "128 bits by 64 bits". For $n = 2048$ this makes a total of $2 \cdot (2^5)^2 = 2048$ multiplicative operations to generate 11 bits, or about 200 operations per bit. A well-established rule of thumb says that a modern CPU executes one multiplicative operation per clock cycle. (Special CPUs that use pipelines and parallelism are significantly faster.) Thus on a 2-GHz CPU with 64-bit architecture we may expect roughly $2 \cdot 10^9/200 \approx 10$ million bits per second, provided the algorithm is implemented in an optimized way. This consideration shows that the BBS generator is almost competitive with a software implementation of a sufficiently secure nonlinear combiner of LFSRs, and is fast enough for many purposes if executed on a present day CPU.

The cryptographic literature offers several pseudorandom generators that follow similar principles as BBS:

The RSA generator (Shamir). Choose a random module m of n bits as a product of two large primes p, q, and an exponent d that is coprime with $(p-1)(q-1)$, furthermore a random initial state $x = x_0$. The state transition is $x \mapsto x^d \bmod m$. Thus we calculate $x_i = x_{i-1}^d \bmod m$, and output the least significant bit, or the $\lfloor \log_2 \log_2 m \rfloor$ least significant bits. If the RSA generator is not perfect, then there exists an efficient algorithm that breaks the RSA cipher. Since calculating d-th powers is more expensive by a factor n than squaring the cost is higher then for BBS: for a random d the algorithm needs $\mathrm{O}(n^3)$ cycles per bit.

The index generator (Blum/Micali). As module choose a random large prime p of n bits, and find a primitive root a for p. Furthermore choose a random initial state $x = x_0$, coprime with $p - 1$. Then calculate $x_i = a^{x_{i-1}} \bmod p$, and and ouput the most significant bit of x_i, or the $\lfloor \log_2 \log_2 p \rfloor$ most significant bits. The perfectness of the index generator relies on the hypothesis that calculating discrete logarithms $\bmod p$ is hard. The cost per bit also is $\mathrm{O}(n^3)$.

The elliptic index generator (Kaliski). It works like the index generator, but replacing the group of invertible elements of the field $\mathbb{F}_p$ by

an elliptic curve over $\mathbb{F}_p$ (such a curve is a finite group in a canonical way).

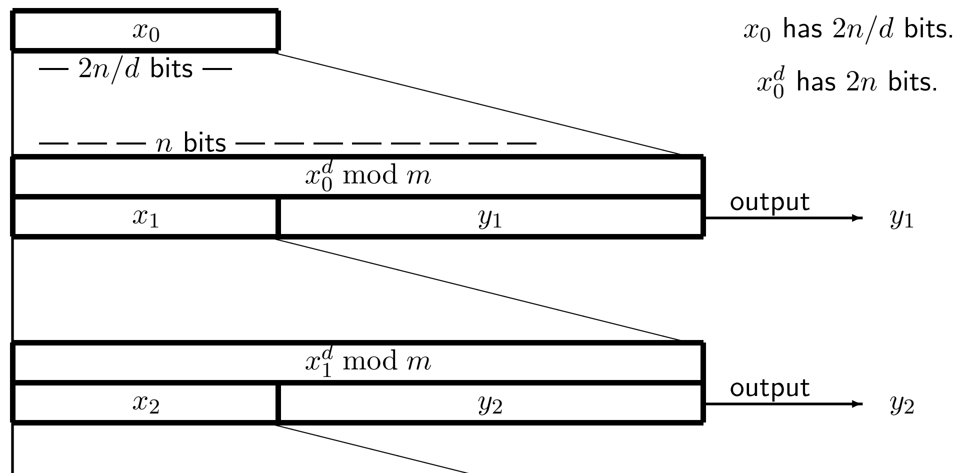


Figure 4.4: Micali-Schnorr generator

4.7 The Micali-Schnorr Generator

Micali and Schnorr proposed a pseudorandom generator that is a descendent of the RSA generator. Fix an odd number $d \geq 3$. The parameter set is the set of all products m of two primes p and q whose bit lengths differ by at most 1, and such that d is coprime with $(p-1)(q-1)$. For an n-bit number m let $h(n)$ be an integer $\approx \frac{2n}{d}$. Then the d-th power of an $h(n)$-bit number is (approximately) a $2n$-bit number.

In the i-th step calculate $z_i = x_{i-1}^d \bmod m$. Take the first $h(n)$ bits as the new state x_i, that is $x_i = \lfloor z_i/2^{n-h(n)} \rfloor$, and output the remaining bits, that is $y_i = z_i \bmod 2^{n-h(n)}$. Thus the bits of the result z_i are partitioned into two *disjoint* parts: the new state x_i, and the output y_i. Figure 4.4 illustrates this scheme.

But why may we hope that this pseudorandom generator is perfect? This depends on the hypothesis: There is no efficient test that distinguishes the uniform distribution on $\{1, \ldots, m-1\}$ from the distribution of $x^d \bmod m$ for uniformly distributed $x \in \{1, \ldots, 2^{h(n)}\}$. If this hypothesis is true, then the Micali-Schnorr generator is perfect. This argument seems tautologic, but heuristic considerations show a relation with the security of RSA and with factorization. Anyway we have to concede that this "proof of security" seems considerably more airy then that for BBS.

How fast do the pseudorandom bits tumble out of the machine? As elementary operations we again count the multiplication of two 64-bit numbers, and the division of a 128-bit number by a 64-bit number with 64-bit quotient. We multiply and divide by the classical algorithms. Thus the product of s (64-bit) words and t words costs st elementary operations. The cost of

division is the same as the cost of the product of divisor and quotient.

The concrete recommendation by the inventors is: $d = 7$, $n = 512$. (Today we would choose a larger n.) The output of each step consists of 384 bits, withholding 128 bits as the new state. The binary power algorithm for a 128-bit number x with exponent 7 costs several elementary operations:

- x has 128 bits, hence 2 words.
- x^2 has 256 bits, hence 4 words, and costs $2 \cdot 2 = 4$ elementary operations.
- x^3 has 384 bits, hence 6 words, and costs $2 \cdot 4 = 8$ elementary operations.
- x^4 has 512 bits, hence 8 words, and costs $4 \cdot 4 = 16$ elementary operations.
- x^7 has 896 bits, hence 14 words, and costs $6 \cdot 8 = 48$ elementary operations.
- $x^7 \bmod m$ has ≤ 512 bits, and likewise costs $6 \cdot 8 = 48$ elementary operations.

This makes a total of 124 elementary operations; among them only one reduction $\bmod m$ (for x^7). Our reward consists of 384 pseudorandom bits. Thus we get about 3 bits per elementary operation, or, by the assumptions in Section 4.6, about 6 milliards bits per second. Compared with the BBS generator this amounts to a factor of about 1000.

Parallelization increases the speed virtually without limit: The MICALI-SCHNORR generator allows complete parallelization. Thus distributing the work among k CPUs brings a profit by the factor k since the CPUs can work indepedently of each other without need of communication.

4.8 The Impagliazzo-Naor Generator

Recall the knapsack problem (or subset sum problem):

Given positive integers $a_1, \ldots, a_n \in \mathbb{N}$ and $T \in \mathbb{N}$.

Wanted a subset $S \subseteq \{1, \ldots, n\}$ with

$$\sum_{i \in S} a_i = T.$$

This problem is believed to be hard. We know it is NP-complete. Building on it Impagliazzo and Naor developed a pseudorandom generator:

Let k and n be (sufficiently large) integers with $n < k < \frac{3n}{2}$. As parameters we choose random $a_1, \ldots, a_n \in [1 \ldots 2^k]$.

Attention: quite a lot of big numbers.

The state space consists of the power set of $\{1, \ldots, n\}$. So the states are subsets $S \subseteq \{1, \ldots, n\}$. We represent them by bit sequences in $\mathbb{F}_2^n$ in the natural way. In each single step we form the sum

$$\sum_{i \in S} a_i \quad \bmod 2^k.$$

This is a k-bit integer. Output the first $k - n$ bits, and retain the last n bits as the new state, see Figure 4.5.

Thus state transition and output function are:

$$\begin{aligned} T(S) &= \sum_{i \in S} a_i \bmod 2^n \\ & \text{(retain the rightmost } n \text{ bits)} \\ U(S) &= \lfloor \frac{\sum_{i \in S} a_i \bmod 2^k}{2^n} \rfloor \\ & \text{output the leftmost } k - n \text{ bits} \end{aligned}$$

If this pseudorandom generator is not perfect, then the knapsack problem admits an efficient solution. Here we omit the proof. See

- R. Impagliazzo, M. Naor: Efficient cryptographic schemes provably as secure as subset sum. J. Cryptology 9 (1996), 199–216.

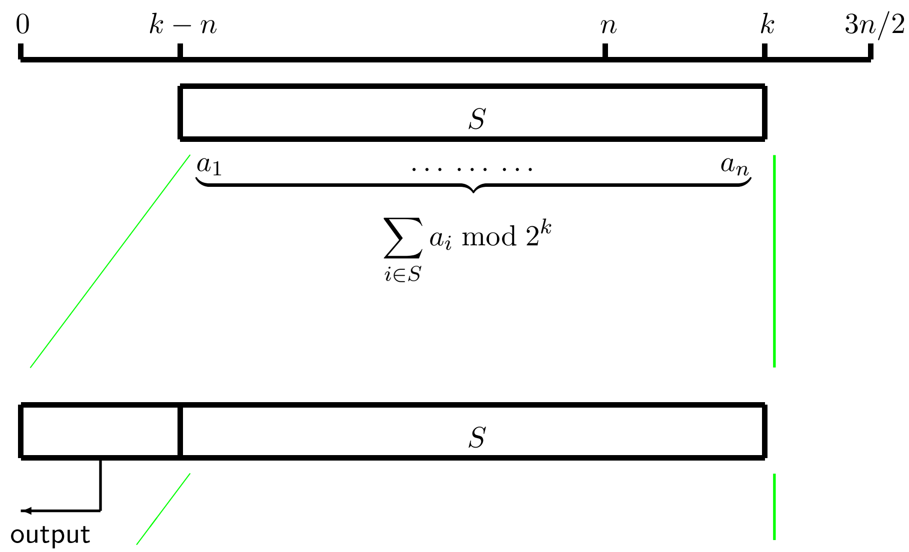


Figure 4.5: The Impagliazzo-Naor generator

Appendix A

Statistical Distinguishers

As usual in these lecture notes we restrict ourselves to finite probability spaces.

A.1 Distingishing Distributions by a Test

Let A be a finite probability space with two probability distributions P_0 and P_1. Accordingly for a real valued function $\Delta \colon A \longrightarrow \mathbb{R}$ we have the mean values (or expectations)

$$\mu_i = \sum_{a \in A} \Delta(a) \cdot P_i(a).$$

For $\varepsilon > 0$ we call Δ an **ε-distinguisher** of P_0 and P_1 if

$$|\mu_1 - \mu_0| \geq \varepsilon.$$

That is, the expectations of Δ with respect to P_0 and P_1 differ considerably.

> Note the analogy with the common statistical test scenario where we decide whether a sample deviates from an assumed distribution by comparing mean values.

This notion has an obvious analogue for bit valued functions (or binary attributes) $\Delta \colon A \longrightarrow \mathbb{F}_2$. Here

$$\mu_i \ = \sum_{a \in \Delta^{-1}(1)} P_i(a) \ = \ P_i(\Delta^{-1}(1))$$

is the probability that $\Delta(a) = 1$ for a randomly chosen $a \in A$. Thus

$$\mu_1 - \mu_0 = P_1(\Delta^{-1}(1)) - P_0(\Delta^{-1}(1)).$$

The "test" Δ ε-distinguishes between the distributions P_1 and P_0 if the probabilities for $\Delta(a) = 1$ with respect to these two distributions differ by at least ε.

> Note that the notion "test" just means "function". However in the present context it suggests a role that this function plays. A similar remark also holds for the notion "randomize".

We may "randomize" our test by more generally considering a function

$$\Delta\colon A \times \Omega \longrightarrow \mathbb{F}_2$$

where Ω is a finite probability space from which we take an additional random input ω, and then consider the probabilities μ_i that $\Delta(a, \omega) = 1$,

$$\mu_i \;=\; \frac{1}{\#A \cdot \#\Omega} \cdot \#\{(a, \omega) \in A \times \Omega \mid \Delta(a, \omega) = 1\}.$$

A.2 Testing Bitsequences

A statistical test for bitsequences of length r is simply a Boolean function $\Delta\colon \mathbb{F}_2^r \longrightarrow \mathbb{F}_2$, a probabilistic statistical test is a function

$$\Delta\colon \mathbb{F}_2^r \times \Omega \longrightarrow \mathbb{F}_2$$

where Ω is a finite probability space.

We want to distinguish between random bitsequences $u \in \mathbb{F}_2^r$, and bitsequences that arise from a "generator map"

$$G\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^r$$

that transforms a randomly chosen $x \in \mathbb{F}_2^n$ (called "seed") to a bitsequence $G(x) \in \mathbb{F}_2^r$. This sequence $G(x)$, if it passes our tests, may qualify as a pseudorandom sequence. In this test scenario the reference distribution P_0 is the uniform distribution on $\mathbb{F}_2^r$,

$$P_0(u) = \frac{1}{2^r} \quad \text{for all } u \in \mathbb{F}_2^r\,.$$

We want to compare it with the induced distribution

$$P_1(u) = \frac{1}{2^n} \cdot \#\{x \in \mathbb{F}_2^n \mid G(x) = u\}\,.$$

Or, somewhat more generally, if G is defined on a subset $A \subseteq \mathbb{F}_2^n$ only,

$$P_1(u) = \frac{1}{\#A} \cdot \#\{x \in A \mid G(x) = u\}\,.$$

A probabilistic statistical test $\Delta : \mathbb{F}_2^r \times \Omega \longrightarrow \mathbb{F}_2$ ε-distinguishes between random bitsequences $u \in \mathbb{F}_2^r$ and sequences generated by $G\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^r$ if

$$|\mu_1 - \mu_0| \geq \varepsilon$$

where

$$\mu_0 = \frac{1}{2^r \cdot \#\Omega} \cdot \#\{(u,\omega) \in \mathbb{F}_2^r \times \Omega \mid \Delta(u,\omega) = 1\}$$

is the probability that the test assigns the value 1 to a random bitsequence $u \in \mathbb{F}_2^r$, and

$$\mu_1 = \frac{1}{2^n \cdot \#\Omega} \cdot \#\{(x,\omega) \in A \times \Omega \mid \Delta(G(x)),\omega) = 1\}$$

is the probability that the test yields the value 1 for a bitstring generated by a random seed $x \in A$.

Examples

We want to distinguish sequences generated by a map $G\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^r$ from random sequences (by deterministic tests, that is $\#\Omega = 1$).

Example 1

First an extremely simple example with the test function

$$\Delta\colon \mathbb{F}_2^r \longrightarrow \mathbb{F}_2, \quad \Delta(u) = \begin{cases} 1 & \text{if } \#\{i \mid u_i = 1\} \geq \frac{r}{2}, \\ 0 & \text{otherwise,} \end{cases}$$

That is Δ decides on the majority of ones in the sequence u. Then obviously $\mu_0 = \frac{1}{2}$.

Case 1a: Let $n = 1$ and $G\colon \mathbb{F}_2 \longrightarrow \mathbb{F}_2^r$ be defined by

$$\begin{aligned} G(0) &= (0,0,0,\ldots), \\ G(1) &= (1,1,1,\ldots). \end{aligned}$$

Then also $\mu_1 = \frac{1}{2}$, yielding $\mu_1 - \mu_0 = 0$. Thus Δ is not an ε-distinguisher for any $\varepsilon > 0$.

Case 1b: We keep the definition of $G(1)$ but change the definition of $G(0)$ to

$$G(0) = (1,0,1,0,1,\ldots).$$

Then $\Delta(G(0)) = \Delta(G(1)) = 1$, hence $\mu_1 = 1$, yielding $\mu_1 - \mu_0 = \frac{1}{2}$. Thus Δ is an ε-distinguisher for $0 < \varepsilon \leq \frac{1}{2}$.

Example 2

For a serious example we consider sequences generated by a linear feedback shift register $G\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^r$ of length n where $2n < r \leq 2^n - 1$. We know that the output of G is distinguished by a low linear complexity $\lambda(u) \leq n$. Therefore we use

$$\Delta\colon \mathbb{F}_2^r \longrightarrow \mathbb{F}_2, \quad \Delta(u) = \begin{cases} 1 & \text{if } \lambda(u) < \frac{r}{2}, \\ 0 & \text{if } \lambda(u) \geq \frac{r}{2}, \end{cases}$$

as test . Since $n < \frac{r}{2}$ this yields

$$\mu_1 \;=\; \frac{1}{2^n} \cdot \#\{x \in \mathbb{F}_2^n \mid \Delta(G(x)) = 1\} \;=\; 1.$$

For arbitrary sequences $u \in \mathbb{F}_2^r$ we know from Theorem 3 that we may expect $\lambda(u) \approx \frac{r}{2}$. A more precise statement follows from the frequency count in Proposition 11:

$$k \;:=\; \#\{u \in \mathbb{F}_2^r \mid \lambda(u)) \leq \frac{r-1}{2}\} \;=\; 1 + \sum_{l=1}^{\lfloor \frac{r-1}{2} \rfloor} 2^{2l-1} \;=\; \frac{1}{2} + \frac{1}{2} \cdot \sum_{l=0}^{\lfloor \frac{r-1}{2} \rfloor} 4^l.$$

Case 2a: Let r be even. Then $\lfloor \frac{r-1}{2} \rfloor = \frac{r}{2} - 1$, and

$$k \;=\; \frac{1}{2} + \frac{1}{2} \cdot \frac{4^{r/2} - 1}{3} \;=\; \frac{1}{2} + \frac{1}{6} \cdot (2^r - 1) \;=\; \frac{1}{3} + \frac{1}{6} \cdot 2^r,$$

$$\mu_0 \;=\; \frac{1}{2^r} \cdot k \;=\; \frac{1}{6} + \frac{1}{3 \cdot 2^r} \;\leq\; \frac{1}{3} \quad \text{for } r \geq 1.$$

Case 2b: Let r be odd. Then $\lfloor \frac{r-1}{2} \rfloor = \frac{r-1}{2}$, and

$$k \;=\; \frac{1}{2} + \frac{1}{2} \cdot \frac{4^{(r+1)/2} - 1}{3} \;=\; \frac{1}{2} + \frac{1}{6} \cdot (2^{r+1} - 1) \;=\; \frac{1}{3} + \frac{1}{3} \cdot 2^r,$$

$$\mu_0 \;=\; \frac{1}{2^r} \cdot k \;=\; \frac{1}{3} + \frac{1}{3 \cdot 2^r} \;\leq\; \frac{1}{2} \quad \text{for } r \geq 1.$$

Hence in any case we have

$$\mu_1 - \mu_0 \;\geq\; \frac{1}{2} \quad \text{for } r \geq 1.$$

Thus Δ is an ε-distinguisher for $0 < \varepsilon \leq \frac{1}{2}$, distinguishing between LFSR sequences and random sequences.

Appendix B

Recursive and Periodic Sequences

How to Find the Shortest Feedback Shift Register That Generates a Given Finite Sequence

This chapter is in the separate document `FindFSR.pdf`

Appendix C

SageMath Code for Bitstream Ciphers

The code snippets can be downloaded from the web page `https://www.staff.uni-mainz.de/pommeren/Cryptology/Bitstream/` where the are grouped in three modules

- `FSR.sage`
- `bmAlg.sage`
- `Periods.sage`

They are written in pure Python, hence execute also in a Python environment. For use with Sage attach them by the instructions:

```
sage: load_attach_path(path="/PATH_TO/Sage", replace=False)
sage: attach('FSR.sage')
sage: attach('bmAlg.sage')
sage: attach('Periods.sage')
```

Remember that `xor` is in the module `Bitblock.sage` introduced with Part II of these lecture notes.

C.1 Feedback Shift Registers

Sage Example C.1 A general feedback shift register. The Boolean function f must be initialized first, using the module `BoolF.sage`, see Part II

```
def fsr(f,x,n):
  """Generate a feedback shift register sequence.
  Parameters: Boolean function f, start vector x,
  number n of output bits."""
  u = x
  outlist = []
  for i in range (0,n):
    b = f.valueAt(u)     # feedback value
    c = u.pop()         # output rightmost bit
    u.insert(0,b)       # feedback the leftmost bit
                        # and shift register to the right
    outlist.append(c)
  return outlist
```

Note the use of the indices:

- start vector: `[x[0],...,x[l-1]] = [u[l-1],...,u[1],u[0]]`
- feedback value: $u_i = f(u_{i-1}, \ldots, u_{i-l})$
- output sequence: `u[0],u[1],...,u[n-1]`

Due to the prominence of LFSRs we concide them a special class that allows for several parallel instances:

Sage Example C.2 Linear feedback shift register.

```
class LFSR(object):
  """Linear Feedback Shift Register
  Attributes: the length of the register
              a list of bits describing the taps of the register
              the state"""

  __max = 1024                                 # max length

  def __init__(self,blist):
    """Initializes a LFSR with a list of taps and the all 0 state."""
    ll = len(blist)
    assert ll <= self.__max, "LFSR_Error: Bitblock too long."
    self.__length = ll
    self.__taplist = blist
    self.__state = [0] * ll

  def __str__(self):
    """Defines a printable string telling the internals of
    the register."""
    outstr = "Length: " + str(self.__length)
    outstr += " | Taps: " + bbl2str(self.__taplist)
    outstr += " | State: " + bbl2str(self.__state)
    return outstr

  def getLength(self):
    """Returns the length of the LFSR."""
    return self.__length

  def setState(self,slist):
    """Sets the state."""
    sl = len(slist)
    assert sl == self.__length, "LFSR_Error: Bitblock has wrong length."
    self.__state = slist

  def nextBits(self,n):
    """Returns the next n bits as a list and updates the state."""
    outlist = []
    a = self.__taplist
    u = self.__state
    for i in range (0,n):
      b = binScPr(a,u)
      c = u.pop()
      u.insert(0,b)
      outlist.append(c)
    self.__state = u
    return outlist
```

C.2 The BM Algorithm

Sage Example C.3 The BM algorithm. It uses `bibScPr` from the module `Bitblock.sage`, see Part II

```
R.<T> = GF(2)[]

def bmAlg(u):
  """Find the shortest linear feedback shift register that generates
  the bit sequence u."""
# Initialization ------------------------------
  lcprof = [0]
  phi = R(1)
  lc = 0                 # linear complexity up to actual index
  r = -1                 # last index
  psi = R(1)             # last feedback polynomial
  nn = len(u)
# End initialization
  for n in range(0,nn):
    b = u[n-lc:n]        # coefficients for feedback
    flist = []           # feedback taps
    for i in range(0,lc):
      if (i < phi.degree()):
        coeff = phi.coefficients(sparse=False)[i+1] # get coefficient of phi at t^(i
      else:
        coeff = 0
      flist.insert(0,coeff)
    d = u[n] - binScPr(flist,b) # discrepancy between predicted bit and true bit
                                #   -- always 0 or 1 in F_2
    if (d == 1):
      eta = phi - T^(n-r) * psi
      if (2*lc <= n):
        m = n+1-lc         # new linear complexity
        t = lc             # linear complexity in last state
        lc = m
        psi = phi
        r = n
      phi = eta
    lcprof.append(lc)
  outlist = [lcprof,phi]
  return outlist
```

Bibliography

[1] Oded Goldreich, *Modern Cryptography, Probabilistic Proofs and Pseudorandomness.* Springer-Verlag, Berlin 1999. ISBN 3-540-64766-X.

[2] Solomon W. Golomb, *Shift Register Sequences.* Revised Edition: Aegean Park Press, Laguna Hills 1982.

[3] Donald E. Knuth, *The Art of Computer Programming, Vol.2, Seminumerical Algorithms.* Addison-Wesley, Reading Mass. 1969, 1981.

[4] W. Meier, O. Staffelbach, Fast correlation attacks on certain stream ciphers, Journal of Cryptology 1 (1989), 159–176.

[5] Alfred J. Menezes, Paul C. van Oorschot, Scott A. Vanstone, *Handbook of Applied Cryptography.* CRC Press, Boca Raton 1997, ISBN 0-8493-8523-7.

[6] Klaus Pommerening, Cryptanalysis of nonlinear shift registers, Cryptologia 40 (2016), 303–315.

[7] Klaus Schmeh, *Cryptography and Public Key Infrastructure on the Internet.* John Wiley, New York 2003.

[8] D. R. Stinson, *Cryptography – Theory and Practice.* CRC Press, Boca Raton 1995.

[9] Yongge Wang, Linear complexity versus pseudorandomness – On Beth and Dai's result. Asiacrypt 1999, 288–298.