Example 50. (bonus!) $p = 29137$ is an example of a left-truncatable prime: the number itself as well as all truncations 9137, 137, 37, 7 are prime. By simply exhausting all possibilities (start with a single digit and keep adding (nonzero) digits on the left until no choice results in a prime), we find that there is a largest left-truncatable prime, namely, 357686312646216567629137.

https://www.youtube.com/watch?v=azL5ehbw_24

Challenge. Find the largest left-truncatable prime which does not have 1 as a digit.

Send me the prime, and an explanation how you found it, by next week for a bonus point!

Comment. You can play the same game in bases different from 10. We expect that (based on the prime number theorem), for every base, there always are just a finite number of truncatable primes (an extra bonus if you can point me to a proof of that claim!), though the number tends to increase with larger bases. The largest truncatable prime for base 30, for instance, is not known (it is estimated to have about 82 digits in base 30).

https://oeis.org/A103463

Example 51. One thing that makes the one-time pad difficult to use is that the key needs to be the same length as the plaintext. What if we have a shorter key and just repeat it until it has the length we need?

That's essentially the Vigenere cipher (in a different alphabet).

Solution. Assuming the attacker knows the length of our key (if she doesn't she can just try all possibilities), this is equivalent to using the one-time pad several times with the same key. That should never be done! Even using a key twice means that we become susceptible to a ciphertext only attack (see Example 48).

So, repeating the key is a terrible idea. However, the idea to create a longer (random) key out of a shorter (random) key is good (we will discuss pseudorandom generators next).

Let us emphasize that, in order to be perfectly confidential, the key for a one-time pad must be chosen completely at random (otherwise, an attacker can make assumptions on the used keys).

Indeed, the need to generate random numbers shows in every modern cipher.

Stream ciphers

Once we have a way to generate **pseudorandom numbers**, we can use the idea of the one-time pad to create a **stream cipher**.

Start with key of moderate size (say, 128 bits).

Use the key k and a PRG (**pseudorandom generator**) to generate a much longer **pseudorandom keystream** $\mathrm{PRG}(k)$. Then encrypt $E_k(m) = m \oplus \mathrm{PRG}(k)$.

We lost perfect confidentiality. Security relies on choice of PRG (must be unpredictable).

As with the one-time pad, we must never reuse the same keystream! That does not mean that we cannot reuse the key: we can do that using a **nonce**: $E_k(m) = m \oplus \mathrm{PRG}((\mathrm{nonce}, k))$, where the seed is produced by combining the nonce and k (for instance, just concatenating them).

The nonce is then passed (unencrypted) along with the message.

To make sure that we never reuse the same keystream, we must never use the same nonce with the same key.

Remark. A nonce can only be used once, as is in its name. Apparently, according to Urban Dictionary, it is also common as a British insult, roughly equivalent to wanker.

Armin Straub
straub@southalabama.edu

How to generate random numbers?

Natural randomness is surprisingly difficult to harness.

You can for instance play around with a Geiger counter but our department is short on these and getting lots of random numbers is again challenging.

Linear congruential generators

(linear congruential generator) Let a, b, m be chosen parameters.
From the seed x_0, we produce the sequence $x_{n+1} \equiv a x_n + b \pmod{m}$.

The choice of a, b, m is crucial for this to generate acceptable pseudorandom numbers.

For instance, glibc uses $a = 1103515245$, $b = 12345$, $m = 2^{31}$. (This is one of two implementations.) In that case, each x_i is represented by precisely 31 bits. [Note that the choice of m makes this very fast.]

https://en.wikipedia.org/wiki/Linear_congruential_generator

Linear congruential generators (LCG) are easy to predict and must not be used for cryptographic purposes. More generally, all polynomial generators are cryptographically insecure. They are still used in practice, because they are fast and easy to implement and have decent statistical properties. (For instance, our online homework is generated using random numbers, and there is no need for crypto-level security there.)

Statistical trouble. Can you see why the sequences produced by the glibc LCG alternate between even and odd numbers? (Similarly, other low bits are much less "random" than the higher bits.) Because of this defect, some programs (and other implementations of `rand()` based on LCGs) throw away the low bits entirely.

Comment. The particular choices of a and b in glibc are somewhat mysterious. See, for instance:
https://stackoverflow.com/questions/8569113/why-1103515245-is-used-in-rand

Example 52. Generate values using the linear congruential generator $x_{n+1} \equiv 5x_n + 3 \pmod{8}$, starting with the seed $x_0 = 6$.

Solution. $x_1 \equiv 1$, $x_2 \equiv 0$, $x_3 \equiv 3$, $x_4 \equiv 2$, $x_5 \equiv 5$, $x_6 \equiv 4$, $x_7 \equiv 7$, $x_8 \equiv 6$. This is the value x_0 again, so the sequence will now repeat. Note that we went through all 8 residues before repeating. Period 8.

Note. Because $8 = 2^3$ we can represent each x_i using exactly 3 bits. Then $x_1, x_2, x_3, \ldots = 1, 0, 3, \ldots$ corresponds to the bit stream $(001\ 000\ 011\ \ldots)_2$.

Example 53. (extra) Observe that the sequence produced by the linear congruential generator $x_{n+1} \equiv a x_n + b \pmod{m}$ must repeat, at the latest, after m terms. (Why?!)

One can give precise conditions on a, b, m to achieve a full period m. Namely, this happens if and only if $\gcd(b, m) = 1$ and $a - 1$ is divisible by all primes (as well as 4) dividing m.

(a) Generate values using a linear congruential generator $x_{n+1} \equiv 2x_n + 1 \pmod{10}$, starting with the seed $x_0 = 5$. When do they repeat? Is that consistent with the mentioned condition?

(b) What are possible values for a so that the LCG $x_{n+1} \equiv a x_n + 11 \pmod{100}$ has period 100?

(c) glibc uses $a = 1103515245$, $b = 12345$, $m = 2^{31}$. After how many terms will the sequence repeat?

Solution.

(a) $x_1 \equiv 1$, $x_2 \equiv 3$, $x_3 \equiv 7$, $x_4 \equiv 5$. This is the value x_0 again, so the sequence will repeat. Period 4.
[The period is less than 10. This is as predicted by the mentioned condition, because $a - 1$ is not divisible by 2 and 5.]

(b) We need that $a - 1$ is divisible by 4 and 5. Equivalently, $a \equiv 1 \pmod{20}$. Hence, possible values are $a = 1, 21, 41, 61, 81$.

(c) Clearly, $\gcd(b, m) = 1$. Also, $a - 1$ is divisible by 4 (and no primes other than 2 divide m). Hence, for every seed, values repeat only after going through all 2^{31} residues.

Example 54. Let's use the PRG $x_{n+1} \equiv 5x_n + 3 \pmod{8}$ as a stream cipher with the key $k = 4 = (100)_2$. The key is used as the seed x_0 and the keystream is $\mathrm{PRG}(k) = x_1 x_2 \ldots$ (where each x_i is 3 bits). Encrypt the message $m = (101\ 111\ 001)_2$.

Solution. We first use the PRG with seed $x_0 = k$ to produce the keystream $\mathrm{PRG}(k) = 7,\ 6,\ 1,\ \ldots = (111\ 110\ 001\ \ldots)_2$.

We then encrypt and get $c = E_k(m) = m \oplus \mathrm{PRG}(k) = (101\ 111\ 001)_2 \oplus (111\ 110\ 001)_2 = (010\ 001\ 000)_2$.

Decryption. Observe that decryption works in the exact same way:

$D_k(c) = c \oplus \mathrm{PRG}(k) = (010\ 001\ 000)_2 \oplus (111\ 110\ 001)_2 = (101\ 111\ 001)_2$.

Note. The keystream continues as $\mathrm{PRG}(k) = 7, 6, 1, 0, 3, 2, 5, 4, \ldots$ At this point it repeats itself because we obtained the value 4, which was our seed. Since the state of this PRG only depends on the value of x_n, and there are 8 possible values for x_n, the period 8 is the longest possible. The previous (extra) example gave conditions on the PRG that guarantee that the period is as long as possible.

Example 55. Can you think of a way in which the numbers produced by a linear congruential generator differ from truly random ones?

Solution. An easy observation for our small examples is the following: by construction, $x_{n+1} \equiv a x_n + b \pmod{m}$, individual values don't repeat unless a full period is reached and everything repeats. Truly random numbers do repeat every now and then (however, if m is large, then this observation is not exactly practical).

Of course, knowing the parameters $a,\ b,\ m$, the numbers generated by the PRG are terribly **predictable**. Knowing just one number, we can produce all the next ones (as well as the ones before). A PRG that is safe for cryptographic purposes should not be predictable like that! (See next example.)

The next example illustrates the vulnerability of stream ciphers, based on predictable PRGs.

Recall that it is common to know or guess pieces of plaintexts; for instance every PDF begins with $\%\mathrm{PDF}$.

Example 56. Eve intercepts the ciphertext $c = (111\ 111\ 111)_2$. It is known that a stream cipher with PRG $x_{n+1} \equiv 5x_n + 3 \pmod{8}$ was used for encryption. Eve also knows that the plaintext begins with $m = (110\ 1\ldots)_2$. Help her crack the ciphertext!

Solution. Since $c = m \oplus \mathrm{PRG}$, we learn that the initial piece of the keystream is $\mathrm{PRG} = m \oplus c = (110\ 1\ldots)_2 \oplus (111\ 1\ldots)_2 = (001\ 0\ldots)_2$. Since each x_n is 3 bits, we conclude that $x_1 = (001)_2 = 1$.

Because the PRG is predictable, we can now recreate the entire keystream! Using $x_{n+1} \equiv 5x_n + 3 \pmod{8}$, we find $x_2 \equiv 0$, $x_3 \equiv 3$, ... In other words, $\mathrm{PRG} = 1, 0, 3, \ldots = (001\ 000\ 011\ \ldots)_2$.

Hence, Eve can decrypt the ciphertext and obtain $m = c \oplus \mathrm{PRG} = (111\ 111\ 111)_2 \oplus (001\ 000\ 011)_2 = (110\ 111\ 100)_2$.

Review.

- A **pseudorandom generator** (PRG) takes a seed x_0 and produces a stream $\mathrm{PRG}(x_0) = x_1\, x_2\, x_3 \ldots$ of numbers, which should "look like" random numbers.

  For cryptographic purposes, these numbers should be indistinguishable from random numbers. Even for somebody who knows everything about the PRG except the seed. (See Example 60.)

- Once we have a PRG, we can use it as a **stream cipher**: Using the key k, we encrypt $E_k(m) = m \oplus \mathrm{PRG}(k)$. [Here, the key stream $\mathrm{PRG}(k)$ is assumed to be in bits.]

  As with the one-time pad, we must never reuse the same keystream!

- To reuse the key, we can use a **nonce**: $E_k(m) = m \oplus \mathrm{PRG}((\mathrm{nonce}, k))$, where the seed is produced by combining the nonce and k (for instance, just concatenating them).

  The nonce is then passed (unencrypted) along with the message.

  To never reuse the same keystream, we must never use the same nonce with the same key.

Linear feedback shift registers

Here is another basic idea to generate pseudorandom numbers:

(linear feedback shift register (LFSR) Let ℓ and $c_1, c_2, \ldots, c_\ell$ be chosen parameters.
From the seed $(x_1, x_2, \ldots, x_\ell)$, where each x_i is one bit, we produce the sequence

$$x_{n+\ell} \equiv c_1 x_{n+\ell-1} + c_2 x_{n+\ell-2} + \ldots + c_\ell x_n \pmod{2}.$$

This method is particularly easy to implement in hardware (see Example 58), and hence suited for applications that value speed over security (think, for instance, encrypted television).

Example 57. Which sequence is generated by the LFSR $x_{n+2} \equiv x_{n+1} + x_n \pmod{2}$, starting with the seed $(x_1, x_2) = (0, 1)$?

Solution. $(x_1, x_2, x_3, \ldots) = (0, 1, 1, 0, 1, 1, \ldots)$ has period 3.

Note. Observe that the two previous values determine the state, so there are $2^2 = 4$ states of the LFSR. The state $(0, 0)$ is special (it generates the zero sequence $(0, 0, 0, 0, \ldots)$), so there are 3 other states. Hence, it is clear that the generated sequence has to repeat after at most 3 terms.

Comment. Of course, if we don't reduce modulo 2, then the sequence $x_{n+2} = x_{n+1} + x_n$ generates the Fibonacci numbers $0, 1, 1, 2, 3, 5, 8, 13, \ldots$

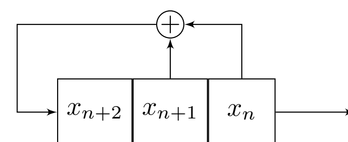
Example 58. Which sequence is generated by the LFSR $x_{n+3} \equiv x_{n+1} + x_n \pmod{2}$, starting with the seed $(x_1, x_2, x_3) = (0, 0, 1)$? What is the period?

[Let us first note that the LFSR has $2^3 = 8$ states. Since the state $(0, 0, 0)$ remains zero forever, 7 states remain. This means that the generated sequence must be periodic, with period at most 7.]

Solution. $(x_1, x_2, x_3, \ldots) = (0, 0, 1, 0, 1, 1, 1, 0, 0, 1, \ldots)$ has period 7.

Again, this is not surprising: 3 previous values determine the state, so there are $2^3 = 8$ states. The state $(0, 0, 0)$ is special, so there are 7 other states.

Note that this LFSR can be implemented in hardware using three registers (labeled x_n, x_{n+1}, x_{n+2} in the sketch to the right). During each cycle, the value of x_n is read off as the next value produced by the LFSR.



Note. In the part $0, 0, 1, 0, 1, 1, 1$ that repeats, the bit 1 occurs more frequently than 0.

The reason for that is that the special state $(0, 0, 0)$ cannot appear.

For the same reason, the bit 1 will always occur slightly more frequently than 0 in LFSRs. However, this becomes negligible if the period is huge, like $2^{31} - 1$ in Example 59.

Armin Straub
straub@southalabama.edu

Example 59. The recurrence $x_{n+31} \equiv x_{n+28} + x_n \pmod{2}$, with a nonzero seed, generates a sequence that has period $2^{31} - 1$.

Note that this is the maximal possible period: this LFSR has 2^{31} states. Again, the state $(0, 0, ..., 0)$ is special (the entire sequence will be zero), so that there are $2^{31} - 1$ other states. This means that the terms must be periodic with period at most $2^{31} - 1$.

Comment. glibc (the second implementation) essentially uses this LFSR.

Advanced comment. One can show that, if the characteristic polynomial $f(T) = x^\ell + c_1 x^{\ell-1} + c_2 x^{\ell-2} + ... + c_\ell$ is irreducible modulo 2, then the period divides $2^\ell - 1$. Here, $f(T) = T^{31} + T^{28} + 1$ is irreducible modulo 2, so that the period divides $2^{31} - 1$. However, $2^{31} - 1$ is a prime, so that the period must be exactly $2^{31} - 1$.

Example 60. Eve intercepts the ciphertext $c = (1111\ 1011\ 0000)_2$ from Alice to Bob. She knows that the plaintext begins with $m = (1100\ 0...)_2$. Eve thinks a stream cipher using a LFSR with $x_{n+3} \equiv x_{n+2} + x_n \pmod{2}$ was used. If that's the case, what is the plaintext?

Solution. The initial piece of the keystream is $\mathrm{PRG} = m \oplus c = (1100\ 0...)_2 \oplus (1111\ 1...)_2 = (0011\ 1...)_2$.

Each x_n is a single bit, and we have $x_1 \equiv 0$, $x_2 \equiv 0$, $x_3 \equiv 1$. The given LFSR produces $x_4 \equiv x_3 + x_1 \equiv 1$, $x_5 \equiv x_4 + x_2 \equiv 1$, $x_6 \equiv 0$, $x_7 \equiv 1$, and so on. Continuing, we obtain $\mathrm{PRG} = x_1 x_2 ... = (0011\ 1010\ 0111)_2$.

Hence, the plaintext would be $m = c \oplus \mathrm{PRG} = (1111\ 1011\ 0000)_2 \oplus (0011\ 1010\ 0111)_2 = (1100\ 0001\ 0111)_2$.

A PRG is **predictable** if, given the stream it outputs (but not the seed), one can with some precision predict what the next bit will be (i.e. do better than just guessing the next bit).

In other words: the bits generated by the PRG must be indistinguishable from truly random bits, even in the eyes of someone who knows everything about the PRG except the seed.

The PRGs we discussed so far are entirely predictable because the state of the PRGs is part of the random stream they output.

For instance, for a given LFSR, it is enough to know any ℓ consecutive outputs $x_n, x_{n+1}, ..., x_{n+\ell-1}$ in order to predict all future output.

We have seen two simple examples of PRGs so far:

- linear congruential generators $x_{n+1} \equiv a x_n + b \pmod{m}$
- LFSRs $x_{n+\ell} \equiv c_1 x_{n+\ell-1} + c_2 x_{n+\ell-2} + ... + c_\ell x_n \pmod{2}$

Of course, we could also combine LFSRs and linear congruential generators (i.e. look at recurrences like for LFSRs but modulo any parameter m).

However, much of the appeal of an LFSR comes from its extremely simple hardware realization, as the sketch in Example 58 indicates.

Example 61. (extra) One can also consider nonlinear recurrences (it mitigates some issues). Our book mentions $x_{n+3} \equiv x_{n+2} x_n + x_{n+1} \pmod{2}$. Generate some numbers.

Solution. For instance, using the seed $0, 0, 1$, we generate $\overbrace{0, 0, 1}^{\text{seed}}, 0, 1, 1, 1, 0, 1, ...$ which now repeats (with period 4) because the state $1, 0, 1$ appeared before. Observe that the generated sequences is only what is called eventually periodic (it is not strictly periodic because $0, 0, 1$ never shows up again).

Example 62. (bonus!) The LFSR $x_{n+31} \equiv x_{n+28} + x_n \pmod{2}$ from Example 59, which is used in glibc, is entirely predictable because observing $x_1, x_2, \ldots, x_{31}$ we know what $x_{32}, x_{33}, \ldots$ are going to be. Alice tries to reduce this predictability by using only $x_3, x_6, x_9, \ldots$ as the output of the LFSR. Demonstrate that this PRG is still perfectly predictable by showing the following:

Challenge. Find a simple LFSR which produces $x_3, x_6, x_9, \ldots$

Send me the LFSR, and an explanation how you found it, by next week for a bonus point!

Comment. There is nothing special about this LFSR. Moreover, a generalization of this argument shows that only outputting every Nth bit of an LFSR is always going to result in an entirely predictable PRG.

A popular way to reduce predictability is to combine several LFSRs (in a nonlinear fashion):

Example 63. The CSS (content scramble system) is based on 2 LFSRs and used for the encryption of DVDs. Before discussing the actual CSS let us consider a baby version of CSS. Our PRG uses the LFSR $x_{n+3} \equiv x_{n+1} + x_n \pmod{2}$ as well as the LFSR $x_{n+4} \equiv x_{n+2} + x_n \pmod{2}$. The output of the PRG is the output of these two LFSRs added with carry.

Adding with carry just means that we are adding bits modulo 2 but add an extra 1 to the next bits if the sum exceeded 1. This is the same as interpreting the output of each LFSR as the binary representation of a (huge) number, then adding these two numbers, and outputting the binary representation of the sum.

If we use $(0,0,1)$ as the seed for LFSR-1, and $(0,1,0,1)$ for LFSR-2, what are the first 10 bits output by our PRG?

Solution. With seed $0,0,1$ LSFR-1 produces $0,1,1,1,0,0,1,0,1,1,\ldots$

With seed $0,1,0,1$ LSFR-2 produces $0,0,0,1,0,1,0,0,0,1,\ldots$

We now add these two:

	0	1	1	1	0	0	1	0	1	1	$\cdots$
+	0	0	0	1	0	1	0	0	0	1	$\cdots$
carry					1						1
	0	1	1	0	1	1	1	0	1	0	$\cdots$

Hence, the output of our PRG is $0,1,1,0,1,1,1,0,1,0,\ldots$.

Important comment. Make sure you realize in which way this CSS PRG is much less predictable than a single LFSR! A single LFSR with ℓ registers is completely predictable since knowing ℓ bits of output (determines the state of the LFSR and) allows us to predict all future output. On the other hand, it is not so simple to deduce the state of the CSS PRG from the output. For instance, the initial $(0,1,\ldots)$ output could have been generated as $(0,0,\ldots)+(0,1,\ldots)$ or $(0,1,\ldots)+(0,0,\ldots)$ or $(1,0,\ldots)+(1,0,\ldots)$ or $(1,1,\ldots)+(1,1,\ldots)$.

[In this case, we actually don't learn anything about the registers of each individual LFSR. However, we do learn how their values have to match up. That's the correlation that is exploited in **correlation attacks**, like the one described next class for the actual CSS scheme.]

Advanced comment. Is the carry important? Yes! Let $a_1, a_2, \ldots$ and $b_1, b_2, \ldots$ be the outputs of LFSR-1 and LFSR-2. Suppose we sum without carry. Then the output is $a_1+b_1, a_2+b_2, \ldots$ (with addition mod 2). If Eve assigns variables $k_1, k_2, \ldots, k_7$ to the $3+4$ seed bits (the key in the stream cipher), then the output of the combined LFSR will be linear in these seven variables (because the a_i and b_i are linear combinations of the k_i). Given just a few more than 7 output bits, a little bit of linear algebra (mod 2) is therefore enough to solve for $k_1, k_2, \ldots, k_7$.

On the other hand, suppose we include the carry. Then the output is $a_1+b_1, a_2+b_2+a_1b_1, \ldots$ (note how a_1b_1 is $1 \pmod{2}$ precisely if both a_1 and b_1 are $1 \pmod{2}$, which is when we have a carry). This is not linear in the a_i and b_i (and, hence, not linear in the k_i), and we cannot solve for $k_1, k_2, \ldots, k_7$ as before.

Armin Straub
straub@southalabama.edu

Example 64. In each case, determine if the stream could have been produced by the LFSR $x_{n+5} \equiv x_{n+2} + x_n \pmod{2}$. If yes, predict the next three terms.

(STREAM-1) $\ldots, 1, 0, 0, 1, 1, 1, 1, 0, 1, \ldots$ (STREAM-2) $\ldots, 1, 1, 0, 0, 0, 1, 1, 0, 1, \ldots$

Solution. Using the LFSR, the values $1, 0, 0, 1, 1$ are followed by $1, 1, 1, 0, \ldots$ Hence, STREAM-1 was not produced by this LFSR.

On the other hand, using the LFSR, the values $1, 1, 0, 0, 0$ are followed by $1, 1, 0, 1, 1, 1, 0, \ldots$ Hence, it is possible that STREAM-2 was produced by the LFSR (for a random stream, the chance is only $1/2^4 = 6.25\%$ that 4 bits matched up). We predict that the next values are $1, 1, 0, \ldots$

Comment. This observation is crucial for the attack on CSS described in Example 65.

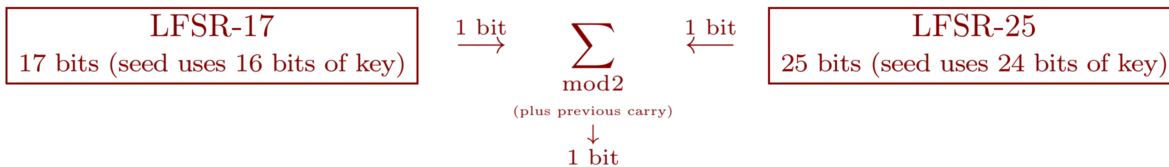
Example 65. (CSS) The CSS (content scramble system) is based on 2 LFSRs and used for the encryption of DVDs. Let us indicate (in a slightly oversimplified way) how to break it.

CSS was introduced in 1996 and first compromised in 1999. One big issue is that its key size is 40 bits. Since $2^{40} \approx 1.1 \cdot 10^{12}$ is small by modern standards, even a direct brute-force attack in time 2^{40} is possible.

However, we will see below that poor design makes it possible to attack it in time 2^{16}.

Historic comment. 40 bits was the maximum allowed by US export limitations at the time.

https://en.wikipedia.org/wiki/Export_of_cryptography_from_the_United_States



CSS PRG combines one 17-bit LFSR and one 25-bit LFSR. The bits output by the CSS PRG are the sum of the bits output by the two LFSRs (this is the usual sum, including carries).

The 40 bit key is used to seed the LFSRs (the 4th bit of each seed is "1", so we need $16 + 24 = 40$ other bits).

Here's how we break CSS in time 2^{16}:

- If a movie is encrypted using MPEG then we know the first few, say x (6-20), bytes of the plaintext.
- As in Example 60, this allows us to compute the first x bytes of the CSS keystream.
- We now go through all 2^{16} possibilities for the seed of LFSR-17. For each seed:
  - We generate x bytes using LFSR-17 and subtract these from the known CSS keystream.
  - This would be the output of LFSR-25. As in Example 64, we can actually easily tell if such an output could have been produced by LFSR-25. If yes, then we found (most likely) the correct seed of LFSR-17 and now also have the correct state of LFSR-25.

This kind of attack is known as a correlation attack.

https://en.wikipedia.org/wiki/Correlation_attack

Comment. Similar combinations of LFSRs are used in GSM encryption (A5/1,2, 3 LFSRs); Bluetooth (E0, 4 LFSRs). Due to certain details, these are broken or have serious weaknesses; so, of course, they shouldn't be used. However, it is difficult to update things implemented in hardware...