An efficient state recovery attack on X-FCSR-256

Stankovski, Paul; Hell, Martin; Johansson, Thomas

Published in:
Fast Software Encryption/Lecture Notes in Computer Science

DOI:
10.1007/978-3-642-03317-9_2

2009

An Efficient State Recovery Attack on X-FCSR-256

Paul Stankovski, Martin Hell, Thomas Johansson

Dept. of Electrical and Information Technology, Lund University,
P.O. Box 118, 221 00 Lund, Sweden

Abstract. We describe a state recovery attack on the X-FCSR-256 stream cipher of total complexity at most $2^{57.6}$. This complexity is achievable by requiring $2^{49.3}$ output blocks with an amortized calculation effort of at most $2^{8.3}$ table lookups per output block using no more than 2^{33} table entries of precomputational storage.

Keywords: stream cipher, FCSR, X-FCSR, cryptanalysis, state recovery

1 Introduction

A common building block in stream ciphers is the Linear Feedback Shift Register (LFSR). The bit sequence produced by an LFSR has several cryptographically interesting properties, such as long period, low autocorrelation and balancedness. LFSRs are inherently linear, so additional building blocks are needed in order to introduce nonlinearity. A Feedback with Carry Shift Register (FCSR) is an alternative construction, similar to an LFSR, but with a distinguishing feature, namely that the update of the register is in itself nonlinear. The idea of using FCSRs to generate sequences for cryptographic applications was initially proposed by Klapper and Goresky in [8].

Recently, we have seen several new constructions based on the concept of FCSRs. The class of F-FCSRs, Filtered FCSRs, was proposed by Arnault and Berger in [1]. These constructions were cryptanalyzed in [7], using a weakness in the initialization function. Also a time/memory tradeoff attack was demonstrated in the same paper.

Another similar construction targeting hardware environments is F-FCSR-H, which was submitted to the eSTREAM project [4]. F-FCSR-H was later updated to F-FCSR-H v2 because of a weakness demonstrated in [6]. F-FCSR-H v2 was one of the four ciphers targeting hardware that were selected for the final portfolio at the end of the eSTREAM project. Inspired by the success, Arnault, Berger, Lauradoux and Minier presented a new construction at Indocrypt 2007, now targeting software implementations. It is named X-FCSR [3]. The main idea was to use two FCSRs instead of one, and to also include an additional nonlinear extraction function inspired by the Rijndael round function. Adding this would allow more output bits per register update and thus increase throughput significantly. Two versions, X-FCSR-256 and X-FCSR-128, were defined producing 256 and 128 bits per register update, respectively. According to the specification X-FCSR-256 runs at 6.5 cycles/byte and X-FCSR-128 runs at 8.2 cycles/byte. As this is comparable to the fastest known stream ciphers, it makes them very interesting in software environments. For the security of X-FCSR-256 and X-FCSR-128 we note that there have been no published attacks faster than exhaustive key search.

In [5] a new property inside the FCSR was discovered, namely that the update was sometimes temporarily linear for a number of clocks. This resulted in a very efficient attack on F-FCSR-H v2 and led to its removal from the eSTREAM portfolio.

In this paper we present a state recovery attack on X-FCSR-256. We use the observation in [5]. The fact that two registers are used, together with the extraction function, makes it impossible to immediately use this observation to break the cipher. However, several additional non-trivial observations will allow a successful cryptanalysis. The keystream is produced using state variables 16 time instances apart. By considering consecutive output blocks, and assuming that the update is linear, we are able to partly remove the dependency of several state variables. A careful analysis of the extraction function then allows us to treat parts of the state independently and brute force these parts separately, leading to an efficient state recovery attack. It is shown that the state can be recovered using $2^{49.3}$ keystream output blocks and a computational complexity of $2^{8.3}$ table lookups per output block. Note that table lookup operations are *much* cheaper than testing a single key.

The paper is organized as follows. In Section 2 we give an overview of the FCSR construction in general and the X-FCSR-256 stream cipher in particular. In Section 3 we describe the different parts of the attack. Each part of the attack is described in a separate subsection and in order to simplify the description we will deliberately base the attack on assumptions and methods that are not optimal for the cryptanalyst. Then, additional observations and more efficient algorithms are discussed in Section 4, leading to a more efficient attack. Finally, some concluding remarks are given in Section 5.

2 Background

This section will review the necessary prerequisites for understanding the details of the attack. FCSRs are presented separately as they are used as core components of the X-FCSR-256 stream cipher. The X-FCSR-256 stream cipher itself is outlined in sufficient detail for understanding the presented attack. For remaining details, the reader is referred to the specification [3].

2.1 Recalling the FCSR Automaton

An FCSR is a device that computes the binary expansion of a 2-adic number p/q, where p and q are some integers, with q odd. For simplicity one may assume that $q < 0 < p < |q|$. Following the notation from [2], the size n of the FCSR is the bitlength of $|q|$ less one. In stream ciphers, p usually depends on the secret key and the IV, and q is a public parameter. The choice of q induces a number of FCSR properties, the most important one being that it completely determines the length of the period T of the keystream.

The FCSR automaton as described in [2] efficiently implements generation of a 2-adic expansion sequence. It contains two registers, a main register M and a carries register C. The main register M contains n cells. Let $M = (m_{n-1}, m_{n-2}, \ldots, m_1, m_0)$ and associate M to the integer $M = \sum_{i=0}^{n-1} m_i \cdot 2^i$.

Let the binary representation of the positive integer $d = (1 + |q|)/2$ be given by $d = \sum_{i=0}^{n-1} d_i \cdot 2^i$. The carries register contains l active cells, $l + 1$ being the number of nonzero binary digits d_i in d. The active carry cells are the ones in the interval $0 \le i \le n - 2$ for which $d_i = 1$, and d_{n-1} must always be set.

Write the carries register as $C = (c_{n-2}, c_{n-3}, \ldots, c_1, c_0)$ and associate it to the integer $C = \sum_{i=0}^{n-2} c_i \cdot 2^i$. Note that l of the bits in C are active and the remaining ones are set to zero.

Representing the integer p as $\sum_{i=0}^{n-1} p_i \cdot 2^i$ where $p_i \in \{0, 1\}$, the 2-adic expansion of the number p/q is computed by the automaton given in Figure 1.

The automaton is referred to as the Galois representation and it is very similar to the Galois representation of an LFSR. For all defined variables we also introduce a time index t, letting $M(t)$ and $C(t)$ denote the content of M and C at time t, respectively.

The addition with carry operation, denoted $\boxplus$ in Figure 1, has a one bit memory, the carry. It operates on three inputs in total, two external

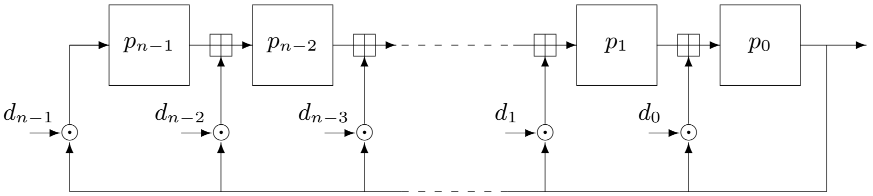

Fig. 1. Automaton computing the 2-adic expansion of p/q.

inputs and the carry bit. It outputs the XOR of the external inputs and sets the new carry value to one if and only if the integer sum of all three inputs is two or three.

In Figure 2 we specifically illustrate (following [2]) the case $q = -347$, which gives us $d = 174 = (10101110)_{binary}$. The X-FCSR family of stream ciphers uses two FCSR automatons at the core of their construction. For the purposes of this paper it is sufficient to recall the FCSR automaton as implemented in Figure 1 and Figure 2.

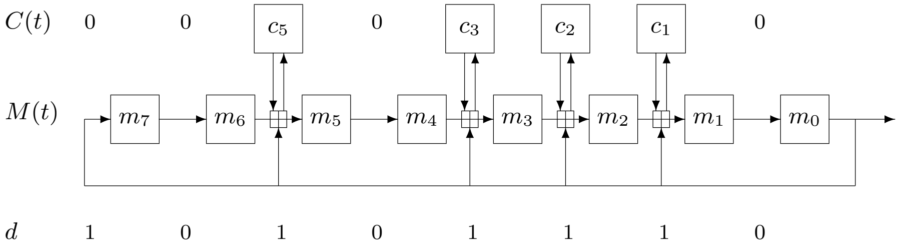

Fig. 2. Example of an FCSR.

The FCSR automaton has n bits of memory in the main register and l bits in the carries register for a total of $n + l$ bits. If (M, C) is our *state*, then many states are equivalent in the sense that starting in equivalent states will produce the same output. As the period is $|q| - 1 \approx 2^n$, the number of states equivalent to a given state is in the order of 2^l.

2.2 Brief Summary of X-FCSR-256 Prerequisites

X-FCSR-256 admits a secret key of 128-bit length and a public initialization vector (IV) of bitlength ranging from 64 to 128 as input. The core of the X-FCSR stream cipher consists of two 256-bit FCSRs with main registers Y and Z which are clocked in opposite directions.

$$\begin{aligned} Y(t) &= (y_{t+255}, \ldots, y_{t+2}, y_{t+1}, y_t), \text{ clocked } \rightarrow \\ Z(t) &= (z_{t-255}, \ldots, z_{t-2}, z_{t-1}, z_t), \text{ clocked } \leftarrow \end{aligned}$$

X-FCSR combines Y and Z to form a 256-bit block $X(t)$ at each discrete time instance t according to

$$X(t) = Y(t) \oplus Z(t),$$

where $\oplus$ denotes bitwise XOR, so that

$$\begin{aligned} X(0) &= (y_{255} \oplus z_{-255}, \ldots, y_2 \oplus z_{-2}, y_1 \oplus z_{-1}, y_0 \oplus z_0) \\ X(1) &= (y_{256} \oplus z_{-254}, \ldots, y_3 \oplus z_{-1}, y_2 \oplus z_0, y_1 \oplus z_1) \\ X(2) &= (y_{257} \oplus z_{-253}, \ldots, y_4 \oplus z_0, y_3 \oplus z_1, y_2 \oplus z_2) \\ &\ldots \end{aligned}$$

Further define

$$W(t) = round_{256}(X(t)) = mix(sr(sl(X(t))), \tag{1}$$

where *sl*, *sr* and *mix* mimic the general structure of the AES round function;

sl is an s-box function applied at byte level,

sr is a row-shifting function operating on bytes,

mix is a column mixing function operating on bytes.

The round functions operate on a 256-bit input, as defined in (1). The general idea behind the round function operations becomes apparent if one considers how the functions operate on the 256-bit input when it is viewed as a 4×8 matrix $\mathbf{A}$ at byte level. Let the byte entries of $\mathbf{A}$ be denoted $a_{i,j}$ with $0 \leq i \leq 3$ and $0 \leq j \leq 7$.

The first transformation layer consists of an S-box function *sl* applied at byte level. The chosen S-box has a number of attractive properties that are described in [3].

The second operation shifts the rows of $\mathbf{A}$, and *sr* is identical to the row shifting operation of Rijndael. *sr* shifts (i.e., rotates) each row of $\mathbf{A}$ to the left at byte level, shifting the first, second, third and fourth rows 0, 1, 3 and 4 bytes respectively.

The purpose of the third operation, mix, is to mix the columns of $\mathbf{A}$. This is also done at byte level according to

$$mix_{256}\begin{pmatrix} a_{0,j} \\ a_{1,j} \\ a_{2,j} \\ a_{3,j} \end{pmatrix} = \begin{pmatrix} a_{3,j} \oplus a_{0,j} \oplus a_{1,j} \\ a_{0,j} \oplus a_{1,j} \oplus a_{2,j} \\ a_{1,j} \oplus a_{2,j} \oplus a_{3,j} \\ a_{2,j} \oplus a_{3,j} \oplus a_{0,j} \end{pmatrix}$$

for every column j of $\mathbf{A}$.

Note that sl, sr and mix are all both invertible and byte oriented. Finally, the 256 bits of keystream that are output at time t are given by

$$out(t) = X(t) \oplus W(t-16). \tag{2}$$

This last equation introduces a time delay of 16 time units. The first block of keystream is produced at $t = 0$ and the key schedule takes care of defining $W(t)$ for $t < 0$.

3 Describing the Attack

3.1 Idea of Attack

A conceptual basis for understanding the attack is obtained by dividing it into the four parts listed below. Each part has been attributed its own section.

- LFSRization of FCSRs
- Combining Output Blocks
- Analytical Unwinding
- Brute-forcing the State

In Section 3.2 we describe a trick we call “LFSRization of FCSRs”. We explain how an observation in [5] can be used to allow treating FCSRs as LFSRs. There is a price to pay for introducing this simplification, of course, but the penalty is not as severe as one may expect.

We observe that we can combine a number of consecutive output blocks to effectively remove most of the dependency on $X(t)$ introduced in (2). The LFSRization process works in our favor here as it provides a linear relationship between FCSR variables. Output block combination is explored in Section 3.3.

Once a suitable combination Q of output blocks is defined, state recovery is the next step. This is done in two parts. In Section 3.4 we explain how to work with Q analytically to transform its constituent parts into something that will get us closer to the state representation. As it turns out, we can do quite a bit here. Finally, having exhausted the analytical options available to us, we bring in the computational artillery and do the remaining job by brute-force. We find that the state can be divided into several almost independent parts and perform exhaustive search on each part separately. This is described in Section 3.5.

3.2 LFSRization of FCSRs

As mentioned above, an observation in [5] provides a way of justifying the validity in treating FCSRs as LFSRs, and does so at a very reasonable cost. We call this process LFSRization of FCSRs, or simply LFSRization when there is no confusion as to what is being treated as an LFSR. There are two parts to the process, a flush phase and a linearity phase.

The observation is simply that a zero feedback bit causes the contents of the carry registers to change in a very predictable way. Adopting a statistical view and assuming independent events is helpful here. Assuming a zero feedback bit, carry registers containing zeros will not change, they will remain zero. The carry registers containing ones are a different matter, though. A 'one' bit will change to a zero bit with probability $\frac{1}{2}$. In essence this means that one single zero feedback bit will cut the number of ones in the carry registers roughly in half.

The natural continuation of this observation is that a sufficient amount of consecutive zero feedback bits will eventually flush the carry registers so that they contain only zeros. On average, roughly half of the carry registers contain ones to start with, so an FCSR with N active carry registers requires roughly $\lg \frac{N}{2} + 1$ zero feedback bits to flush the 'ones' away with probability $\frac{1}{2}$. By expected value we therefore require roughly $\lg \frac{N}{2} + 2$ zero feedback bits to flush a register completely. For X-FCSR-256 we have $N = 210$, indicating that we need no more than nine zero feedback bits to flush a register.

After the flush phase, a register is ready to act as an LFSR. In order to take advantage of this state we need to maintain a linearity phase in which we keep having zero feedback bits fed for a sufficiently long duration of time. As we will see from upcoming arguments, we will in principle require the linearity property for two separate sets of six consecutive zero feedback bits, with the two sets being sixteen time units apart. We will need the FCSRs to act as LFSRs during this time, so our base requirement consists of two smaller LFSRizations, each requiring roughly $9 + 6$ bits for flush and linearity phase respectively. The probability of the two smaller LFSRizations occurring in *both* registers Y and Z simultaneously is $2^{-4(9+6)} = 2^{-60}$. In other words, our particular LFSRization condition appears once in about 2^{60} output blocks.

A real life deviation from the theoretical flush reasoning was noted in [5]. We cannot flush the carry register entirely as the last active carry bit will tend to one instead of zero. As further noted in [5], flushing all but the last carry bit does not cause a problem in practice. Consider the linearized FCSR in Figure 3, it produces a maximal number of zero feedback bits for an FCSR of its size.

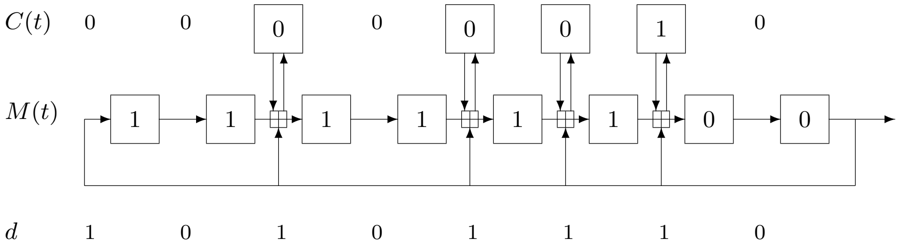

Fig. 3. Maximally linearized FCSR.

In simulations and analytical work we must compensate for this effect, of course, but the theoretical reasoning to follow remains valid as we allow ourselves to treat FCSRs as simple LFSRs. The interested reader is referred to [5] for details on this part.

Furthermore, assumptions of independence are not entirely realistic. Although the theoretical reasoning above is included mainly for reasons of completeness, simulations show that we are not far from the truth, effectively providing some degree of validation for the theory. Our simulations show that we have $2^{28.7}$ for the Y register and $2^{27.5}$ for Z for a total of at most $2^{56.2}$ expected output blocks for LFSRization in X-FCSR as we require it.

Our requirements for the basic attack are as follows. At some specific time instance t we require the carry registers of X and Y to be completely flushed except for the last bit. Here we also require the tails of the main registers to contain the bit sequence 111100 as in Figure 3 to guarantee at least six consecutive zero feedback bits. At $t + 16$ we require this precise set-up to appear once again. In each set, the first five zero feedback bits are needed to ensure that the main registers are linear. The last remaining zero feedback bit is used only to facilitate equation solving in the state recovery part, as it guarantees that the last carry bit remains set.

To be fair and accurate we will use the simulation values, which puts us at

$$COST_{LFSRization} \leq 2^{56.2}$$

for the basic attack. Later, in Section 4.2, we will see how we can reduce the requirements to only four consecutive zero feedback bits per set for a complexity of

$$COST_{LFSRization} \leq 2^{49.3}.$$

3.3 Combining Output Blocks

The principal reason for combining consecutive output blocks is to obtain a set of data that is easier to analyze and work with, ultimately leading to a less complicated way to reconstruct the cipher state. Remember that we now treat the two FCSRs as LFSRs with the properties given in Section 3.2.

The main observation is that the modest and regular clocking of the two main registers provides us with the following equality:

$$X(t) \oplus [X(t+1) \ll 1] \oplus [X(t+1) \gg 1] \oplus X(t+2) = (\star, 0, 0, \ldots, 0, \star) \tag{3}$$

The shifting operations $\ll$ and $\gg$ on the left hand side denote shifting of the corresponding 256-bit block left and right, respectively. From this point onward we discard bits that fall over the edge of the 256 bit blocks, and we do so without loss of generality or other such severe penalties. The right hand side is then the zero vector[1], with the possible exception of the first and last bits which are undetermined (and denoted $\star$). Define

$$OUT(t) = out(t) \oplus [out(t+1) \ll 1] \oplus [out(t+1) \gg 1] \oplus out(t+2) \tag{4}$$

[1] Recall that we ignore the effects of the last carry bit being one instead of zero, as explained in Section 3.2. The arguments below are valid as long as adjustments are made accordingly.

in the corresponding way. We have

$$\begin{aligned} OUT(t) = & \\ & X(t) \oplus [X(t+1) \ll 1] \oplus [X(t+1) \gg 1] \oplus X(t+2) \oplus \\ & W(t-16) \oplus [W(t-15) \ll 1] \oplus [W(t-15) \gg 1] \oplus W(t-14) \\ & = \\ & (\star, 0, 0, \ldots, 0, \star) \oplus \\ & W(t-16) \oplus [W(t-15) \ll 1] \oplus [W(t-15) \gg 1] \oplus W(t-14) \\ & \approx \\ & W(t-16) \oplus [W(t-15) \ll 1] \oplus [W(t-15) \gg 1] \oplus W(t-14), \end{aligned} \tag{5}$$

where $\approx$ denotes bitwise equality except for the first and last bit. This expression allows us to relate keystream bits to bits inside the generator that are just a few time instances apart. This will turn out to be very useful when recovering the state of the FCSRs. In order to further unwind equation (5) we need to take a closer look at the constituent parts of W, namely the round function operations sl, sr and mix.

3.4 Analytical Unwinding

Reviewing the round function operations from Section 2.2, recall that all of the operations are invertible and byte oriented. We can also see that the operations mix, sr and their inverses are linear over $\oplus$, such that

$$\begin{aligned} mix(A \oplus B) &= mix(A) \oplus mix(B), \\ sr(A \oplus B) &= sr(A) \oplus sr(B). \end{aligned}$$

Obviously, sl does not harbor the linear property. So, in order to unwind (5) as much as possible, we would now ideally like to apply mix^{-1} and sr^{-1} in that order. Let us begin with focusing on the mix operation.

The linearity of mix over $\oplus$ is the first ingredient we need as it allows us to apply mix^{-1} to each of the W terms separately. The shifting does cause us some problems, however, since

$$mix^{-1}\left(W(t) \ll 1\right) \neq mix^{-1}\left(W(t)\right) \ll 1.$$

Therefore mix^{-1} cannot be applied directly in this way, but realizing that mix^{-1} is a byte-oriented operation, it is clear that the equality holds if one restricts comparison to every bit position except the first and last bit of every byte. This is easy to realize if one considers the origin and destination byte of the six middlemost bits as mix^{-1} is applied. One single bit shift does not affect the destination byte of these bits. Furthermore, the peripheral bits that are shifted out of their byte position are mapped to another peripheral bit position. We therefore have

$$\begin{aligned} mix^{-1}\left(OUT(t)\right) \cong\ & sr\left(sl\left(X(t-16)\right)\right) \oplus \\ & [\ sr\left(sl\left(X(t-15)\right)\right) \ll 1\ ] \oplus \\ & [\ sr\left(sl\left(X(t-15)\right)\right) \gg 1\ ] \oplus \\ & sr\left(sl\left(X(t-14)\right)\right), \end{aligned}$$

where $\cong$ denotes equality with respect to the six middlemost bits of each byte. The same arguments apply to sr^{-1}, so we define $Q(t) = sr^{-1}\left(mix^{-1}\left(OUT(t)\right)\right)$ to obtain

$$\begin{aligned} Q(t) \cong\ & sl\left(X(t-16)\right) \oplus \\ & [\ sl\left(X(t-15)\right) \ll 1\ ] \oplus \\ & [\ sl\left(X(t-15)\right) \gg 1\ ] \oplus \\ & sl\left(X(t-14)\right). \end{aligned} \tag{6}$$

Loosely put, we can essentially bypass the effects of the *mix* and *sr* operations by ignoring the peripheral bits of each byte.

We have combined consecutive keystream blocks $out(t)$ into Q in hope of Q being easier to analyze than $out(t)$. Since the ultimate goal is to map $out(t)$ to Y and Z, we don't have very far to go now. As our expression for Q involves only X and sl, let's see how and at what cost we can brute-force Q and solve for Y and Z.

3.5 Brute-forcing the State

The brute-forcing part can most easily be understood by focusing on one specific byte position in $Q(t)$. Given the, say, seventh byte in $Q(t)$, how can we *uniquely* reconstruct the relevant parts of Y and Z? Let us first figure out which bits one needs from $Y(t-16)$ and $Z(t-16)$ in order to be able to calculate the given byte in $Q(t)$. Note that this step is possible only because of the LFSRization described in Section 3.2.

Have another look at the first part of expression (6): $sl\left(X(t-16)\right)$. Since sl is an S-box function that operates on bytes, we need to know the full corresponding byte from $X(t-16)$. Those eight bits are derived from

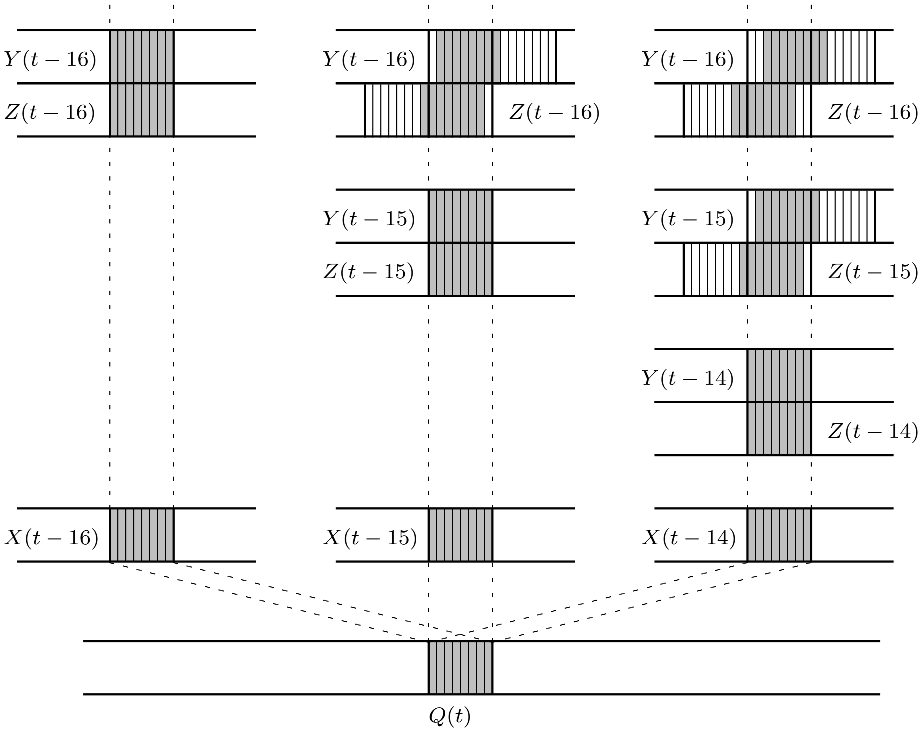

Fig. 4. Bit usage for one byte in $Q(t)$.

eight bits in each of Y and Z, totaling 16 bits, as shown in the left column of Figure 4 below.

The next parts of (6) involves $sl(X(t-15))$. The same reasoning applies here, we need to know the full corresponding byte of $X(t-15)$ in order to be able to calculate this S-box value. But, since the main registers act like LFSRs, most of the bits we need from Y and Z for $X(t-15)$ have already been employed for $X(t-16)$ previously. Since the two main registers are clocked only one step at each time instance, only two more bits are needed, one from Y and one from Z. This is illustrated by the middle column of Figure 4 below. We count 18 bits in Y and Z so far.

In the same vein, two more bits are needed from Y and Z to calculate $sl(X(t-14))$, illustrated in the remaining part of Figure 4. This brings the total up to 20 bits. All in all, for one byte position in $Q(t)$ we have total bit usage as shown in Figure 5.

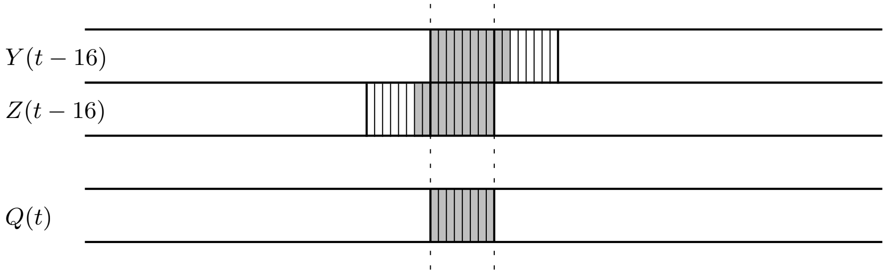

Fig. 5. Bit usage in $Q(t)$.

So, 10 bits in $Y(t-16)$ and 10 bits in $Z(t-16)$ is what we require to be able to calculate one specific byte position in $Q(t)$. By restricting our attention to the six middlemost bits of each byte in Q we accomplish two objectives; we effectively reduce the number of unknown bits we are dealing with in Y and Z, and we simplify the expression for calculating the byte in Q by safely reducing the effects of the shifting operation. Specifically, shifting one bit left or right does not bring neighboring bytes into play.

Focusing on one single byte position gives us six equations, one for each of the six middlemost bits, and 20 unsolved variables, one for each bit position in Y and Z. This amounts to an underdetermined system, of course, but we can easily add more equations by having a look at the same byte position in $Q(t+1)$. The six middle bits of that byte give us six new equations at the cost of introducing a few new variables. To see how many, we must perform the analysis for $Q(t+1)$ corresponding to Figure 4. The total bit usage for one byte position in $Q(t+1)$ in terms of bits in $Y(t-16)$ and $Z(t-16)$ is given in Figure 6.

From this we see that the six new equations have the downside of introducing two new variables. In total we therefore have 12 equations and 22 variables after including $Q(t+1)$. The system is still underdetermined, so we can add $Q(t+2)$ as well. This brings us to 18 equations and 24 variables, and so on. Adding $Q(t+3)$ provides 24 equations for 26 variables, but at this level we will obtain a resulting system that provides hope of being fully determined as we may also reduce the number of variables by reusing already determined values as we scan Q byte by byte from one

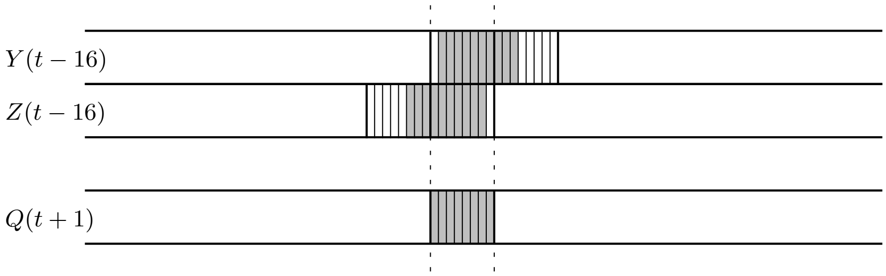

Fig. 6. Bit usage in $Q(t+1)$.

end to the other to solve for bits in Y and Z. The corresponding bit usage for our four consecutive Q's in terms of bits in $Y(t-16)$ and $Z(t-16)$ is illustrated in Figure 7 below.

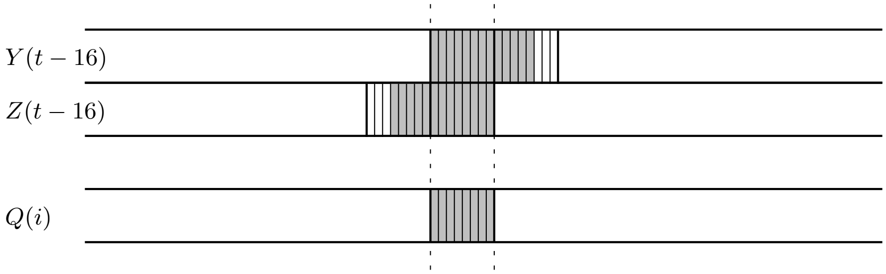

Fig. 7. Total bit usage for $Q(i)$, $t \leq i \leq t+3$.

When brute-forcing one byte position in Q we essentially solve for 26 bits. If we scan Q from left to right, solving the corresponding system for each byte, we can reuse quite many of these bits. Instead of solving for 26, we need only solve for 16 as the remaining 10 have already been determined. This is illustrated in Figure 8. Reusing bits in this way works fine for all byte positions except the first one. For the first byte position we don't have any prior solution to lean back on, but we can use the LFSRization assumption. We have already assumed that we have 'zero' feedback bits coming in and these are valid to use when solving the system. The system for the first byte contains 21 unsolved variables, so the 24 equations do indeed provide a fully determined system.

Employing bit reuse, the total cost for the brute-forcing part becomes

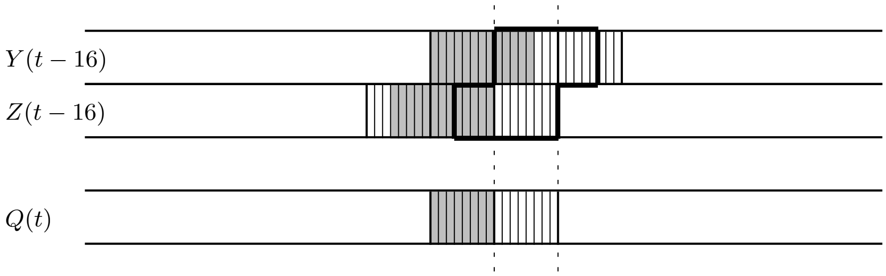

Fig. 8. Reusing bits when solving for $Q(t)$.

$$COST_{brute-force} \leq 2^{21} + 31 \times 2^{16} < 2^{22}.$$

This calculation is a little bit idealized, however, since we in practice do obtain multiple solutions in some cases. These occur sometimes because the peripheral bits in the system appear in only one or two of the equations, causing false solutions. These are easy to spot, though, as the succeeding equation system will generally be unsolvable as we attempt to reuse 10 of the bits from the false solution. And since the false solutions do not appear in abundance, we do not compensate for this complexity wise.

This concludes the basic attack, in which we have assumed availability of four separate sets of six consecutive zero feedback bits as described in Section 3.2.

4 Improving the Attack

4.1 Precomputation

We can reduce the workload of the brute-force part almost entirely using precomputation. A precomputation table for solving the first byte system would require 2^{24} entries[2] as we have the 24 bits from the four Q's as input to resolve 21 bits. For succeeding byte positions we may limit the number of Q's to three, which provides 18 equations for the 16 unsolved variables. Adding the already determined 8 bits to the formula, we can see that a table with $2^{18+8} = 2^{26}$ entries will suffice. In this context we consider these tables to be of reasonable size.

[2] The storage is trivially realized using on average at most two 4-byte words per entry.

The total amortized cost for attempting to solve for the entire state is then given by considering the relative frequencies of table lookups per byte position. Using table lookups as unit, we have

$$COST_{brute-force} < 1 + \frac{1}{8}\left(1 + \frac{1}{4} + \frac{1}{4^2} + \ldots\right) = \frac{7}{6}$$

using no more than 2^{27} table entries worth of storage.

4.2 Lowering the required keystream

In the basic attack we assumed existence of four separate sets of six consecutive zero feedback bits, as explained in Section 3.2. Our next improvement is to reduce the required keystream by loosening the above requirement to only four consecutive zero feedback bits in each set and increasing the calculation effort correspondingly.

To shine some light upon some of the details involved in this process, consider equation (5) once more. The purpose of the second of the two sets of zero feedback bits is to make way for the X's to cancel out properly according to equation (3). A 'one' feedback bit in the second set prohibits the X's from canceling out entirely. We can cope with this anomaly by compensating for such a non-null aggregate of the X's in equation (5). The important issue is that we are in control of the resulting changes.

The first set of zero feedback bits govern the composition of the W's. With zero feedback bits all the way we obtain a well defined system when solving for the first byte position in Q. If the fifth feedback bit is a 'one' the system changes somewhat, but it is still as well defined as before. Here, too, we are in control of the resulting changes. Our increase in computational effort consists of constructing and using the corresponding tables for the resulting systems, so that we can solve the resulting system regardless of these last bit values.

Without the sixth and last zero feedback bit in each set we would not know if the last remaining carry bit has ultimately been nullified or not. Our basic attack assumptions allow us to easily figure out the value of the last carry bit. We may remove the requirement of the sixth zero feedback bit in each set if we instead solve all the 16 similar but essentially different resulting variants of the system. In principle, we can allow creation of 16 new tables, one for each system, for a total workload increase factor of 16. Therefore, storage requirements increase to 2^{28} table entries for the first byte position systems but remain at at most 2^{26} for the succeeding byte position systems for a total of 2^{29} table entries. Note that no specialized tables for the last byte position system are needed because of the symmetry in the systems for the first and last byte positions.

The corresponding arguments are valid when removing the requirement of the fifth zero feedback bit. The fifth feedback bit from two of the sets affect the system of the first byte position for an increase in storage and computation of a factor of at most 16, again. Storage requirements increase to 2^{32} table entries for the first byte position systems and remain at at most 2^{26} for succeeding byte position systems. All in all, we can solve the entire system for all cases using only

$$COST_{brute-force} < 2^{4+4} \times \frac{7}{6} < 2^{8.3}$$

table lookups into at most 2^{33} table entries of storage. The interested reader is referred to [5], in which a similar situation is discussed.

In practice, the $COST_{LFSRization}$ part tells us how many keystream blocks we need to analyze before we can find a favorable situation that allows the brute-force method to go all the way to recovering the state. The $COST_{brute-force}$ part is payed by performing that many calculations for each analyzed keystream block. To summarize, we have

$$COST = COST_{LFSRization} \times COST_{brute-force} < 2^{49.3+8.3} = 2^{57.6}$$

using no more than 2^{33} table entries worth of precomputational storage.

5 Concluding remarks

It is clear that the design of the X-FCSR stream cipher family is not sufficiently secure. Depending on one's inclination, it is possible to attribute this insufficiency to the modest clocking of the two FCSRs, the size or number of FCSRs, how they are combined, the complexity of the round function or some other issue. All of these factors are parts of the whole, but the key insight, however, is that it is important not to rely on the non-linear property of FCSRs too heavily. The LFSRization process shows that it is relatively cheap to linearize FCSRs, the cost being roughly logarithmic in the size of active carry registers.

More details on the last improvements and a more in-depth exposé of the effects of the last carry bit on system solving are available in the full version of this paper. There we also exploit the symmetry situation of requiring several consecutive 'one' feedback bits for an additional reduction in required keystream.

Let us end with a note on applicability to X-FCSR-128. The basic attack presented here works for X-FCSR-128 as well, but the resulting complexity is much less impressive. The LFSRization process is identical for both variants of X-FCSR, as is the analytical unwinding. Enter round functions. The two registers are 256 bits in size in both cases, but X-FCSR-128 "folds" the contents of the registers to produce a 128-bit result, implying that more bits are condensed into one byte position of Q as analyzed in Section 3.5. This affects cost in a negative way, actually making the attack more expensive for X-FCSR-128. We estimate that at least twelve consecutive Q's are needed for a fully determined first byte system. This leads to a guesstimated expected value of about 2^{74} output blocks for the attack to come through in the basic setting, each output block requiring roughly one table lookup into a storage of at most 2^{72} table cells.

References

1. F. Arnault and T. Berger. F-FCSR: Design of a new class of stream ciphers. In H. Gilbert and H. Handschuh, editors, *Fast Software Encryption 2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 83–97. Springer-Verlag, 2005.
2. F. Arnault, T. Berger, and C. Lauradoux. Update on F-FCSR stream cipher. eSTREAM, ECRYPT Stream Cipher Project, Report 2006/025, 2006. *http://www.ecrypt.eu.org/stream*.
3. F. Arnault, T. P. Berger, C. Lauradoux, and M. Minier. X-FCSR - a new software oriented stream cipher based upon FCSRs. In K. Srinathan, C. Pandu Rangan, and M. Yung, editors, *Progress in Cryptology—INDOCRYPT 2007*, volume 4859 of *Lecture Notes in Computer Science*, pages 341–350. Springer, 2007.
4. ECRYPT. eSTREAM: ECRYPT Stream Cipher Project, IST-2002-507932. Available at *http://www.ecrypt.eu.org/stream/*.
5. M. Hell and T. Johansson. Breaking the F-FCSR-H stream cipher in real time. ASIACRYPT 2008, To Appear, 2008.
6. E. Jaulmes and F. Muller. Cryptanalysis of ECRYPT candidates F-FCSR-8 and F-FCSR-H. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/046, 2005. *http://www.ecrypt.eu.org/stream*.
7. E. Jaulmes and F. Muller. Cryptanalysis of the F-FCSR stream cipher family. In B. Preneel and S. Tavares, editors, *Selected Areas in Cryptography—SAC 2005*, volume 3897 of *Lecture Notes in Computer Science*, pages 36–50. Springer-Verlag, 2005.
8. A. Klapper and M. Goresky. 2-adic shift registers. In R.J. Anderson, editor, *Fast Software Encryption'93*, volume 809 of *Lecture Notes in Computer Science*, pages 174–178. Springer-Verlag, 1994.