J. Cryptol. (2014) 27: 1–22
DOI: 10.1007/s00145-012-9130-9

Journal of
CRYPTOLOGY

An Efficient State Recovery Attack on the X-FCSR Family of Stream Ciphers*

Paul Stankovski, Martin Hell, and Thomas Johansson

Dept. of Electrical and Information Technology, Lund University, P.O. Box 118, 221 00 Lund, Sweden
paul.stankovski@eit.lth.se; martin.hell@eit.lth.se; thomas.johansson@eit.lth.se

Communicated by Mitsuru Matsui

Received 25 August 2010
Online publication 8 September 2012

Abstract. We describe a state recovery attack on the X-FCSR family of stream ciphers. In this attack we analyse each block of output keystream and try to solve for the state. The solver will succeed when a number of state conditions are satisfied. For X-FCSR-256, our best attack has a computational complexity of only $2^{4.7}$ table lookups per block of keystream, with an expected $2^{44.3}$ such blocks before the attack is successful. The precomputational storage requirement is 2^{33}. For X-FCSR-128, the computational complexity of our best attack is $2^{16.3}$ table lookups per block of keystream, where we expect $2^{55.2}$ output blocks before the attack comes through. The precomputational storage requirement for X-FCSR-128 is 2^{67}.

Key words. 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 [11].

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 [1,12]. These constructions were cryptanalyzed in [10], using a weakness in the initialization function. Also a time/memory tradeoff attack was demonstrated in the same paper.

* This is the full version of the paper An Efficient State Recovery Attack on X-FCSR [14], solicited by the Editors-in-Chief as one of the best papers from FSE 2009, based on the recommendation of the program committee.

© International Association for Cryptologic Research 2012

Another similar construction targeting hardware environments is F-FCSR-H, which was submitted to the eSTREAM project [6]. F-FCSR-H was later updated to F-FCSR-H v2 because of a weakness demonstrated in [9]. 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 [4]. 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 [7,8] 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 the X-FCSR family consisting of X-FCSR-128 and X-FCSR-256. We use the observation in [7,8]. 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 X-FCSR-256 state can be recovered using $2^{44.3}$ output keystream blocks and a computational complexity of $2^{4.7}$ table lookups per output block on average. Note that table lookup operations are *much* cheaper than testing a single key. The corresponding figures for X-FCSR-128 are $2^{55.2}$ for the number of output keystream blocks with a computational effort of $2^{16.3}$ per block.

The paper is organized as follows. In Section 2 we give an overview of the FCSR construction in general and the X-FCSR-128 and X-FCSR-256 stream ciphers 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 8.

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

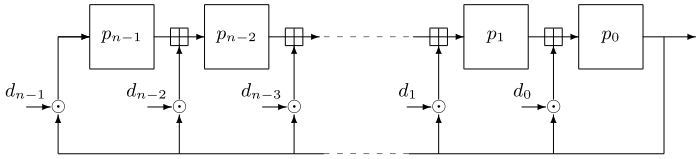

Fig. 1. Automaton computing the 2-adic expansion of p/q.

sufficient detail for understanding the presented attack. For remaining details, the reader is referred to the specification [4].

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 FCSR-based ciphers, p usually depends on the secret key and the initialization vector (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 the 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 Fig. 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 Fig. 1, has a one bit memory, the carry. It operates on three inputs in total, two external 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 Fig. 2 we specifically illustrate (following [2]) the case $q = -347$, which gives us $d = 174 = (10101110)_{\text{binary}}$.

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 Figs. 1 and 2.

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

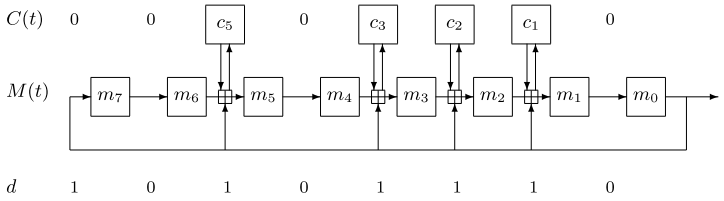

Fig. 2. Example of an FCSR.

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 Prerequisites*

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

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

At each discrete time instance t, Y and Z are used to form a 256-bit block $X(t)$ according to

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

where $\oplus$ denotes bitwise XOR, so that

$$X(0) = (y_{-255} \oplus z_{255}, \ldots, y_{-2} \oplus z_2, y_{-1} \oplus z_1, y_0 \oplus z_0),$$
$$X(1) = (y_{-254} \oplus z_{256}, \ldots, y_{-1} \oplus z_3, y_0 \oplus z_2, y_1 \oplus z_1),$$
$$X(2) = (y_{-253} \oplus z_{257}, \ldots, y_0 \oplus z_4, y_1 \oplus z_3, y_2 \oplus z_2),$$
$$\ldots$$

X-FCSR-128 and X-FCSR-256 are identical up to this point, but they differ in the extraction function. Let us concentrate on X-FCSR-256 for a while. $X(t)$ is used as input to the extraction function. We define

$$W(t) = round_{256}\big(X(t)\big) = mix_{256}\big(sr_{256}\big(sl_{256}\big(X(t)\big)\big)\big), \tag{1}$$

where sl_{256}, sr_{256} and mix_{256} 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 X-FCSR-256 round function operates 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 [4].

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 $t < 0$.

X-FCSR-128 has a very similar extraction function, but it operates on a 128-bit block. If we by $X_L(t)$ and $X_R(t)$ denote the left and right parts of $X(t)$ according to $X(t) = X_L(t)||X_R(t)$ where $||$ denotes concatenation, we form $\widetilde{X}(t) = X_L(t) \oplus X_R(t)$ and similarly define

$$W(t) = round_{128}\big(\widetilde{X}(t)\big) = mix_{128}\big(sr_{128}\big(sl_{128}\big(\widetilde{X}(t)\big)\big)\big), \tag{3}$$

and

$$out(t) = \widetilde{X}(t) \oplus W(t-16) \tag{4}$$

for X-FCSR-128. Now view the 128-bit block as a 4×4 matrix at byte level. sr_{128} shifts the first, second, third and fourth rows by 0, 1, 2 and 3 bytes respectively, and the corresponding mix function uses the same matrix as above, but now with only four columns.

Throughout this paper we will use X-FCSR-256 as our basic case to show how the attack works in full detail. In Section 6 we show how to adapt the attack to X-FCSR-128.

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
- Solving for the State

In Section 3.2 we describe a trick we call "LFSRization of FCSRs". We explain how an observation in [7,8] 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, we find that the state can be divided into several almost independent parts that can be treated separately. This is described in Section 3.5.

3.2. *LFSRization of FCSRs*

As mentioned above, an observation in [7,8] 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 in the Galois implementation of an FCSR, see Fig. 2, 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 the X-FCSR family we have $N = 210$, indicating that we need no more than 9 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 5

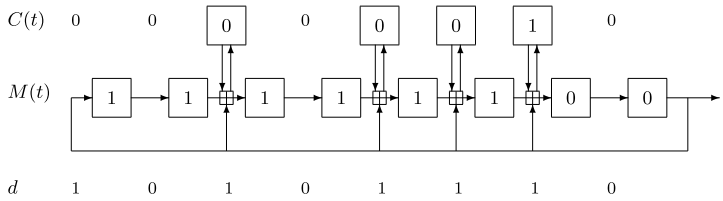

Fig. 3. Maximally linearized FCSR outputting zero feedback bits.

R1 At time $t-16$, the carry registers of Y are completely flushed except for the last bit.
R2 At least 5 consecutive zero feedback bits are output starting from time $t-16$.
R3 At time t, the carry registers of Y are completely flushed except for the last bit.
R4 At least 5 consecutive zero feedback bits are output starting from time t.

Fig. 4. Requirements for the Y register.

consecutive zero feedback bits, with the two sets being 16 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 $9+5$ bits for flush and linearity phase, respectively. The probability of the two smaller LFSRizations occurring in *both* registers Y and Z simultaneously is $2^{-4(9+5)} = 2^{-56}$. In other words, our particular LFSRization condition appears once in about 2^{56} output blocks.

A real-life deviation from the theoretical flush reasoning was noted in [7,8]. We cannot flush the carry register entirely as the last active carry bit will tend to one instead of zero. As further noted in [7,8], flushing all but the last carry bit does not cause a problem in practice. Consider the linearized FCSR in Fig. 3, it produces a maximal number of zero feedback bits for an FCSR of its size.

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 [7,8] 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 about $2^{28.0}$ for the Y register and $2^{26.0}$ for Z for a total of at most $2^{54.0}$ expected output blocks for LFSRization in X-FCSR in the basic setting made explicit below.

Our requirements for the basic attack are summarized as follows. At some specific time instance we require the carry registers of Y and Z to be completely flushed except for the last bit. Here we also require the tails of the main registers to contain the bit sequence $\ldots 11100$ as in Fig. 3 to guarantee at least 5 consecutive zero feedback bits for the five upcoming time instances. Sixteen time instances later we require this set-up to appear once again. In each flush-set, the five upcoming zero feedback bits ensure that the main registers remain linear.

In Fig. 4 we explicitly list the requirements for the Y register, with the requirements for the Z register defined correspondingly.

The X-FCSR family members output a block of keystream at each clocking, 128 and 256 bits for X-FCSR-128 and X-FCSR-256, respectively. Let $COST_{\text{keystream}}$ denote the required number of such output blocks (or clockings) for an attack to come through. To be fair and accurate we will use the simulation values, which puts us at

$$COST_{\text{keystream}} < 2^{54.0}$$

for the basic attack scenario with requirements **R1**–**R4**. Later, in Sections 5.1 and 5.2, we will minimize keystream by relaxing requirements **R2** and **R4** to only 3 consecutive zero feedback bits, and in Section 5.3 we use a symmetry observation for a reduced keystream complexity of

$$COST_{\text{keystream}} < 2^{44.3}.$$

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 the following equality:

$$X(t) \oplus \big[X(t+1) \ll 1\big] \oplus \big[X(t+1) \gg 1\big] \oplus X(t+2) = (\star, 0, 0, \ldots, 0, \star). \quad (5)$$

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

$$OUT(t) = out(t) \oplus \big[out(t+1) \ll 1\big] \oplus \big[out(t+1) \gg 1\big] \oplus out(t+2) \quad (6)$$

in the corresponding way. We have

$$\begin{aligned} OUT(t) &= X(t) \oplus \big[X(t+1) \ll 1\big] \oplus \big[X(t+1) \gg 1\big] \oplus X(t+2) \\ &\quad \oplus W(t-16) \oplus \big[W(t-15) \ll 1\big] \oplus \big[W(t-15) \gg 1\big] \oplus W(t-14) \\ &= (\star, 0, 0, \ldots, 0, \star) \\ &\quad \oplus W(t-16) \oplus \big[W(t-15) \ll 1\big] \oplus \big[W(t-15) \gg 1\big] \oplus W(t-14) \\ &\approx W(t-16) \oplus \big[W(t-15) \ll 1\big] \oplus \big[W(t-15) \gg 1\big] \oplus W(t-14), \end{aligned} \quad (7)$$

where $\approx$ denotes bitwise equality except for the peripheral bits. This expression allows us to relate keystream bits to bits inside the generator that are just a few time instances

[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.

apart. This will turn out to be very useful when recovering the state of the FCSRs. In order to further unwind Eq. (7) 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 also see that the operations *mix*, *sr* and their inverses are linear over $\oplus$, such that

$$mix(A \oplus B) = mix(A) \oplus mix(B),$$
$$sr(A \oplus B) = sr(A) \oplus sr(B).$$

Obviously, *sl* does not harbor the linear property. So, in order to unwind (7) 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, however, cause us some problems since

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

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 6 middlemost bits as mix^{-1} is applied. One single bit shift does not affect the destination byte of these bits. Furthermore, a peripheral bit that is shifted out of its byte position is mapped to another peripheral bit position. We therefore have

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

where $\cong$ denotes equality with respect to the 6 middlemost bits of each byte. The same arguments apply to sr^{-1}, so we define

$$Q(t) = sr^{-1}\big(mix^{-1}\big(OUT(t)\big)\big) \tag{8}$$

to obtain

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

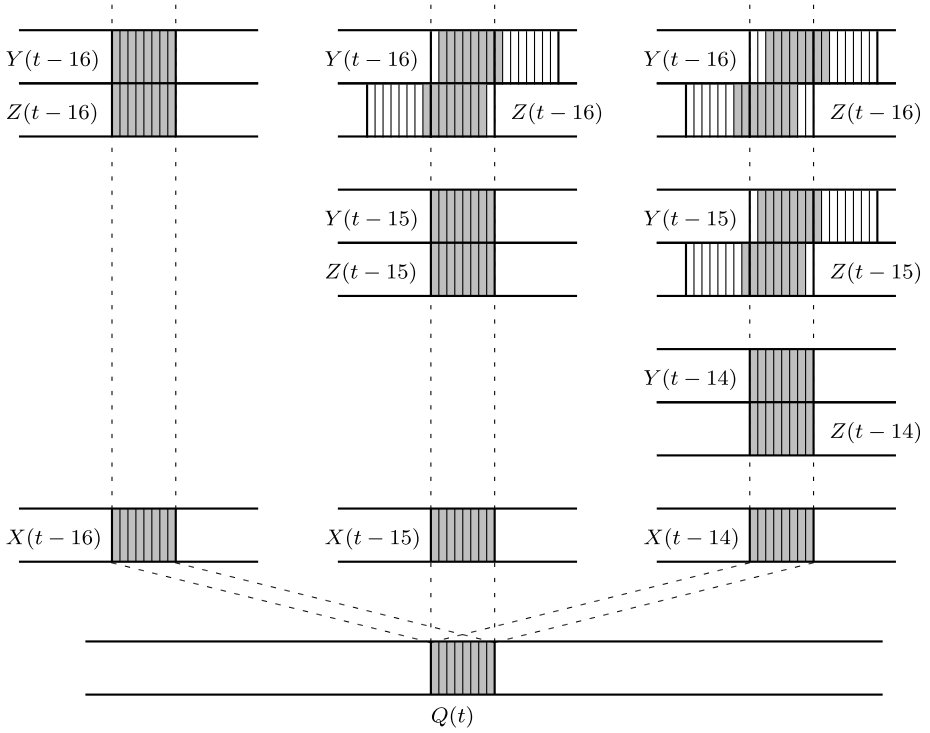

Fig. 5. Bit usage for one byte in $Q(t)$.

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 $out(t)$. 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. *Solving for the State*

In this section we outline the state recovery step. We proceed in a divide-and-conquer fashion by dividing the state into several almost independent parts and treat each part separately by solving a related equation system.

State solving 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 corresponding 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.

Consider the first part of expression (8): $sl(X(t-16))$. Since sl is an S-box function that operates on bytes, we need to know the full corresponding byte from $X(t-16)$. Those 8 bits are derived from 8 bits in each of Y and Z, totaling 16 bits, as shown in the left column of Fig. 5.

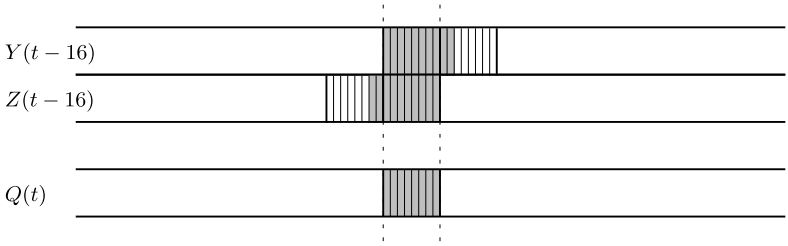

Fig. 6. Bit usage in $Q(t)$.

The next parts of (8) 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 2 more bits are needed, one from Y and one from Z. This is illustrated by the middle column of Fig. 5. We count 18 bits in Y and Z so far.

In the same vein, 2 more bits are needed from Y and Z to calculate $sl(X(t-14))$, illustrated in the remaining part of Fig. 5. 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 Fig. 6.

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 6 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 6 middlemost bits, and 20 unsolved variables, one for each bit position in Y and Z. This amounts to an underdetermined system, but we can easily add more equations by having a look at the same byte position in $Q(t+1)$. The 6 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 Fig. 5. 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 Fig. 7.

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 $Q(t+1)$, and 18 equations and 24 variables after including $Q(t+2)$. The corresponding bit usage for our three consecutive Q's in terms of bits in $Y(t-16)$ and $Z(t-16)$ is illustrated in Fig. 8.

When solving one byte position in Q we essentially recover 24 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 24, we need only solve for 16 as the remaining 8 have already been determined. Thus, we actually have an overdetermined system with 18 equations and 16 variables. This is illustrated in Fig. 9.

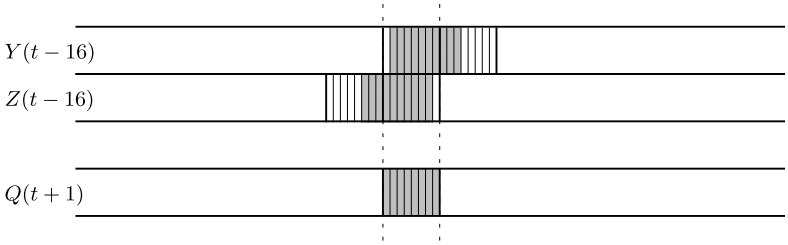

Fig. 7. Bit usage in $Q(t+1)$.

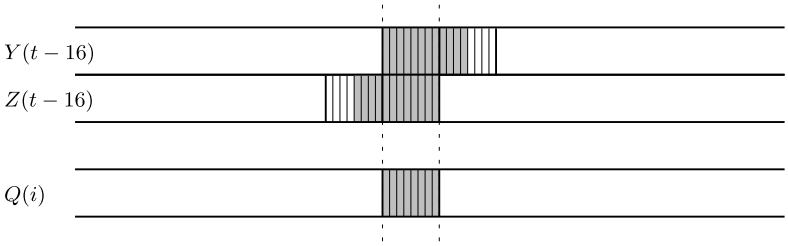

Fig. 8. Total bit usage for $Q(i)$, $t \leq i \leq t+2$.

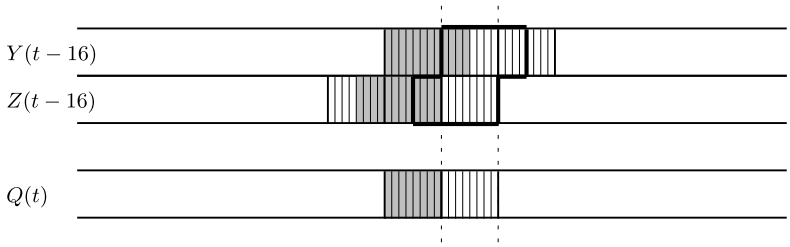

Fig. 9. Reusing bits when solving for $Q(t)$.

Reusing bits in this way works fine for all byte positions except the first one. For the first byte position we do not have any prior solution to lean back on, so at first glance it seems that this system is larger and thus more expensive to solve. In Section 4 we will explain what the first and last byte position systems look like in more detail, and we will see how to use the LFSRization assumption to reduce the system complexity in these cases.

As it turns out, the middle byte systems are largest in terms of unsolved bits, which will dominate the worst-case cost of the equation solving part. Let $COST_{\text{solver}}$ denote the required number of variable assignments that must be tested for an attack to come through. Employing bit reuse, the *worst-case* cost for the solving part becomes

$$COST_{\text{solver}} < 32 \times 2^{16} = 2^{21}.$$

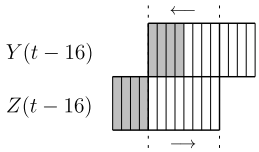

Fig. 10. Equation system at middle byte position.

This concludes the principles of the basic attack, in which we have assumed availability of four separate sets of 5 consecutive zero feedback bits as described in Section 3.2. The only thing that remains is to calculate the solving complexity more rigorously. Using precomputed lookup tables and considering the expected-case complexity, we can significantly lower the cost for equation solving. This is what we will do in the following sections.

4. The Anatomy of Equation Solving

In our attack scenario we wait for the first opportunity in which our keystream fulfills the requirements given in Fig. 4. For every block of keystream that is output, we try to solve for the state. Most times we fail, but our solver will find a solution when the requirements **R1**–**R4** have been met for registers Y and Z. Therefore, the average cost is more interesting from a practical perspective, so this is what we will compute next.

In Section 4.1 we warm up by finding the cost when precomputation is disallowed. In Section 4.2 we analyze the precomputation case, which concludes the basic attack on X-FCSR-256. We start by taking a closer look at the equation systems at different byte positions.

4.1. *Equation Solving*

Restating Eq. (9), one may view the equation solving game as solving for the state at time $t-16$ given output at time t.

$$\begin{aligned} OUT(t) &= X(t) \oplus \left[X(t+1) \ll 1\right] \oplus \left[X(t+1) \gg 1\right] \oplus X(t+2) \\ &\quad \oplus W(t-16) \oplus \left[W(t-15) \ll 1\right] \oplus \left[W(t-15) \gg 1\right] \oplus W(t-14) \\ &= (\star, 0, 0, \ldots, 0, \star) \\ &\quad \oplus W(t-16) \oplus \left[W(t-15) \ll 1\right] \oplus \left[W(t-15) \gg 1\right] \oplus W(t-14). \end{aligned} \tag{9}$$

When solving for the state without precomputation, what we do in practice is to run through all unknown bits in $Y(t-16)$ and $Z(t-16)$ to see if we can find a configuration that produces the expected output. We do this byte by byte from left to right in the Y and Z registers for efficiency. Three byte position cases need to be considered; first, middle and last. The simplest case, the middle byte position case, is depicted in Fig. 10.

We saw this system in Fig. 9 before. The grayed bits are the ones we can reuse from solving the equation system from the preceding byte position, leaving 16 bits unsolved. Feedback bits do not affect the equation systems at middle byte position.

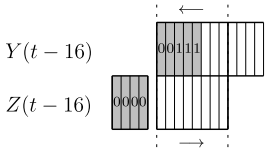

Fig. 11. Equation system at first byte position.

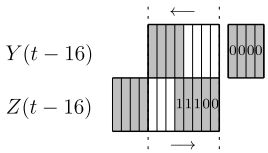

Fig. 12. Equation system at last byte position.

The equation system for the first byte position is shown in Fig. 11.

As before, we have 24 variables and 18 equations. One difference is that four of the variables are new, having just entered the Z register. Another difference is that we cannot reuse variables from a prior solution. On the other hand we can use assumption **R2**. The last 5 bits of Y are known (00111), and the 4 bits entering Z are all zero.

Thus, for the first byte position system, 9 of the 24 bits are predetermined, leaving 15 bits unsolved.

The equation system at the last byte position mirrors that of the first, except that the bits from the previous byte position system are also given.

Thus, for the last byte position system, 17 of the 24 bits are predetermined, leaving only 7 bits unsolved (Fig. 12).

The amortized cost for attempting to solve for the entire state is then given by considering the relative frequencies of solving attempts per byte position. We process the byte positions from left to right in the natural way.

Using verification of Eq. (9) as unit, the *expected*[2] cost for recovering the state is given by

$$COST_{\text{solver}} = 2^{15} + \frac{1}{8}\left(2^{16} + \frac{2^{16}}{4} + \frac{2^{16}}{4^2} + \cdots + \frac{2^{16}}{4^{29}} + \frac{2^7}{4^{30}}\right) < 2^{15.5}. \qquad (10)$$

The factor $\frac{1}{8}$ is derived from the fact that we have 15 variables and 18 equations for the first byte position system. For the middle byte position systems we have 16 variables and 18 equations, producing the factor $\frac{1}{4}$ above.

[2] It is also possible to reduce the size of the first equation system even further by using the zero feedback bits from the flush phase. That approach does produce a significant saving to, for example, $COST_{\text{solver}} < 2^8$ for eight flush-bits. As the number of bits needed to flush the carry register is unknown, this assumption may be false, leading to more keystream before the state can be recovered.

4.2. *Equation Solving with Precomputation*

The amortized cost for attempting to solve for the entire state using precomputation is, similarly, given by considering the relative frequencies of lookups per byte position. Instead of solving an equation system at each step, we look the answer up in a table. Letting $COST_{\text{solver}}$ denote the average number of required table lookups for a keystream block to be fully analyzed, we have

$$COST_{\text{solver}} = 1 + \frac{1}{8}\left(1 + \frac{1}{4} + \frac{1}{4^2} + \cdots + \frac{1}{4^{30}}\right) < 2^{0.3}.$$

Total computational complexity, i.e., the total number of table lookups, is given by

$$COST = COST_{\text{keystream}} \times COST_{\text{solver}}.$$

To see how the corresponding tables are constructed, consider Eq. (8) once more. We have 24 Y and Z variables that are combined into 18 Q values. As a conceptual starting point, make an auxiliary table A containing the corresponding Q values for all 2^{24} variable configurations. That is, table A has 2^{24} entries, each containing an 18-bit value.

The equation system for the first byte position has only 15 of the 24 Y and Z variables undetermined. Filtering out the corresponding 2^{15} entries from table A and making a reverse lookup hash table will do the trick. The hash table will be indexed on, at most, 2^{15} 18-bit Q values, and the entries will be the corresponding variable assignment (15 bits) for Y and Z.

For the middle byte position systems we correspondingly populate a hash table indexed on the 18-bit Q values *and* the eight known and reused variables. This table will contain 2^{24} entries, as we will use all of table A. Each entry will state the corresponding variable assignment (16 bits) for Y and Z.

Although it seems to be possible to use the table for the first byte position system for the last byte position by mirroring, this opportunity is destroyed by our upcoming minimizations of keystream requirements. Therefore, for the last byte position systems we construct a hash table indexed on the 18-bit Q values *and* the eight known and reused variables. This table will contain 2^{15} entries, where each entry represents the corresponding variable assignment (7 bits) for Y and Z.

In total, no more than 2^{25} table entries are needed, and each table entry fits well within a 32-bit word. The above numbers are possible to obtain in practice by employing, for example, cuckoo hashing (see [13]), which offers practical $O(1)$ lookups and amortized $O(1)$ insertions with $O(n)$ storage (all constants small).

This concludes the basic attack on X-FCSR-256 for which we have

$$COST_{\text{keystream}} < 2^{54.0}$$

and

$$COST_{\text{solver}} < 2^{0.3}$$

with 2^{25} precomputational storage. These numbers assume 5 feedback bits as described by the requirements stated in Fig. 4.

In the next section, our aim is to reduce the amount of required keystream.

5. Reducing Keystream

We go on to reduce the keystream requirements by increasing the amount of equation solving. This is done in two steps. In Section 5.1 we see how the zero vector compensation from Eq. (7) can be modified to allow for a faster state recovery. The corresponding effort is then applied to the equation solving part, which will reduce the required keystream even further. This is examined in Section 5.2.

5.1. *Zero Vector Compensation*

We will now take a closer look at requirements **R3** and **R4** from Fig. 4. Referring to Eq. (9) once more, one can see that the purpose of **R3** and **R4** is to make way for the X's to cancel out properly according to Eq. (5). Requirement **R4** for the Y register dictates the behavior at one end of the vector, and that of the Z register controls the other.

If we relax **R4** from at least 5 consecutive zero feedback bits to precisely four, that fifth one feedback bit 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 Eq. (9). The important issue is that we are in control of the resulting changes. As noted in Section 3.2, at least 5 consecutive zero feedback bits forces the tails of the main registers to contain the bit sequence ...11100 as in Fig. 3. To handle the case with *precisely* 4 consecutive zero feedback bits, one must compute the corresponding zero vector[3] for the five-bit tail ...01100 and compensate accordingly. Solving for the state in the case with precisely 4 consecutive zero feedback bits amounts to solving a very similar equation system for the first and last byte position.

It is the tail of the Y register that determines the left end of the zero vector. The tail of the Z register determines the right end. It seems at first that we need to quadruple the computation to solve for all four variants. Taking the relative frequencies into account, the last byte position system is very cheap to solve. In fact, it comes almost for free. The modified cost for the case when we relax **R4** to at least 4 consecutive zero feedback bits is

$$COST_{\text{solver}} = 2\left(1 + \frac{1}{8}\left(1 + \frac{1}{4} + \frac{1}{4^2} + \cdots + \frac{1}{4^{29}} + \frac{2}{4^{30}}\right)\right) < 2^{1.3}.$$

To support our new solver we also need two new tables. These are the same size as the previous ones for the first and last byte position. The total space requirement therefore remains at most 2^{25} table entries, since the storage requirement for the middle byte position systems dominates.

We take the procedure one step further and relax **R4** to only 3 consecutive zero feedback bits. This time we run into a complication. The bad news is that we get a one feedback bit for the last of the five output blocks. This triggers an additional summation at all carry cell positions, effectively pushing several ones into the carry vector. The problem with this is that the LFSRization effect is ruined, so we cannot hope to push the process even further to relax **R4** to only two consecutive zero feedback bits. For the

[3] The term zero vector may seem a little out of place as the vector is not all-zeros, but we appeal to the readers' idealizingly Plaotesque nature.

three-case, however, we can still calculate a zero vector compensation and proceed as above.

Another positive note is that we do not need to consider both tail cases when we consider 3 consecutive feedback bits. We have covered the four-or-more case above with five-bit tails, and it remains to treat the precisely-three-case. The tails of the registers must contain the bit sequence ...00100 or ...10100, when compared to Fig. 3. Both tail sequences lead to the exact same zero vector compensation, so we only need to consider the four-bit tail ...0100. In terms of equation solving, this means that we have one less known variable for the first and last byte position systems. But the storage requirements are, as before, dominated by the middle byte position systems, so we may disregard the sizes of the first and last byte position systems. We cannot, however, disregard the equation system differences at the different middle byte positions, the differences imposed by the last feedback bit setting the many carries. The sizes of the systems remain the same, but we now have 30 different middle byte position systems, which increases memory usage for precomputation by a factor 2^5.

To summarize, we can recover the state also when **R4** is relaxed to 3 or more consecutive zero feedback bits. The three different tails and the possibility of the one feedback bit setting the many carries together generate six different equation systems for the first and last byte position. For the middle bytes there are four different systems. It is possible to recover the entire state with an expected

$$COST_{\text{solver}} < 6\left(1+\frac{1}{8}\left(1+\frac{1}{4}+\frac{1}{4^2}+\cdots+\frac{1}{4^{29}}+\frac{6}{4^{30}}\right)\right) < 2^{2.9}$$

table lookups into a precomputational storage of at most 2^{30} table entries. The interested reader is referred to [7,8], in which a similar situation is discussed.

Table lookups for the first byte position are most expensive as they occur most frequently. We may optimize the solver by merging all first byte position system tables. The cost of recovering the entire state is then reduced to

$$COST_{\text{solver}} < 1+\frac{6}{8}\left(1+\frac{1}{4}+\frac{1}{4^2}+\cdots+\frac{1}{4^{29}}+\frac{6}{4^{30}}\right) < 2^{1.1}$$

without increasing the storage requirements.

5.2. *A Second Requirement Relaxation*

Having relaxed requirement **R4** to 3 consecutive zero feedback bits, we now turn the attention to requirement **R2**. Can we use the corresponding technique to relax **R2** to at least 3 consecutive zero feedback bits? This question is answered in the affirmative, and the corresponding cost of recovering the entire state is then

$$COST_{\text{solver}} < 1+\frac{6^2}{8}\left(1+\frac{1}{4}+\frac{1}{4^2}+\cdots+\frac{1}{4^{29}}+\frac{6^2}{4^{30}}\right) < 2^{2.9},$$

when both **R2** and **R4** are simultaneously relaxed. As before, the possible Z register tails at last byte position are solved for at virtually no cost. The formula above indicates that we can treat the possible endings in each register as a separate system and

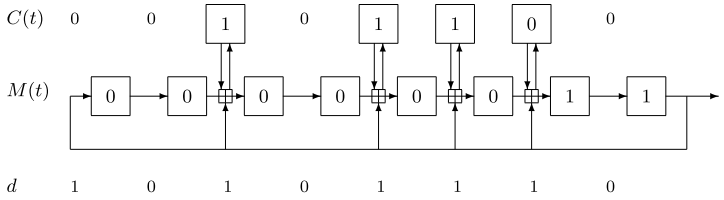

Fig. 13. Maximally linearized FCSR outputting 'one' feedback bits.

create a separate table for each. For the middle byte position there are, as before, four different systems. The size of the systems at middle byte position dominates the storage requirements. We double our previous storage estimate to at most 2^{31} table entries.

5.3. *Feedback Ones—A Symmetry Case*

It is also possible to shorten the keystream requirement further by considering the symmetry case of several consecutive one feedback bits. Analogously to Fig. 3, a maximally linear FCSR outputting one feedback bits is given in Fig. 13.

In the original case with zero feedbacks, we wait for the carries to be flushed in order for the FCSR to act linearly. In the conjugate case with one feedbacks, the same linear behavior appears when we have accumulated ones in the carries. Reviewing the entire methodology for the zero feedback case, one can see that the corresponding arguments and techniques hold when we are facing one feedback bits as well. The only practical difference is that we alter the constants in the equation systems we are solving.

Instead of requiring simultaneous LFSRization with zero feedbacks in both Y and Z registers, we can relax our requirement to simultaneous LFSRization with zero *or* one feedbacks in each of Y and Z. Thus, by quadrupling the precomputational storage requirements and increasing the computational effort, we may reduce the amount of required keystream to one quarter using this additional observation.

To summarize again, with requirements **R2** and **R4** relaxed to at least 3 zero feedback bits and exploiting the symmetry ones-case, we obtain

$$COST_{\text{keystream}} < 2^{44.3}$$

with

$$COST_{\text{solver}} < 1 + \frac{4 \cdot 6^2}{8}\left(1 + \frac{1}{4} + \frac{1}{4^2} + \cdots + \frac{1}{4^{29}} + \frac{4 \cdot 6^2}{4^{30}}\right) < 2^{4.7}$$

using precomputational storage of size 2^{33}.

This is our best result, both for minimizing the keystream requirement of the attack and for minimizing the total number of table lookups for recovering the state. The various costs are shown in Table 1.

6. X-FCSR-128

The LFSRization process is identical for both variants of X-FCSR, as is the analytical unwinding, leaving only the equation solving parts to be considered. In Eq. (3) we can

Table 1. Costs for the X-FCSR-256 attack.

	Keystream	Solver	Storage
Basic attack w/o tables	$2^{54.0}$	$2^{15.5}$	–
Basic attack w/ tables	$2^{54.0}$	$2^{0.3}$	2^{25}
Reduced keystream attack Sections 5.1–5.3	$2^{44.3}$	$2^{4.7}$	2^{33}

see that the 256-bit entity $X(t)$ is "folded" to produce a 128-bit result for X-FCSR-128. In effect, more state 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 are forced to solve larger equation systems to recover the state, so we therefore need more Qs to increase the number of equations. The equation system for the first byte position is illustrated in Fig. 14 for the case when six Qs are used.

This system is the largest with its 45 unknown variables. As before, the time complexity of state recovery is largely determined by the size of the middle byte position system. Regardless of how many Qs we use, this system has 32 unknown variables as depicted in Fig. 15.

Note that the six Qs induce 36 equations, leaving the first byte position system underdetermined by a factor 2^9, and the middle byte position systems overdetermined by a factor 16. The corresponding third byte position system is not illustrated, but it has 17 unsolved variables. Solving for the state without precomputation (compare to Eq. (10)) therefore costs

$$COST_{\text{solver}} = 2^{45} + 2^9\left(2^{32} + \frac{2^{32}}{16} + \frac{2^{32}}{16^2} + \cdots + \frac{2^{32}}{16^{13}} + \frac{2^{17}}{16^{14}}\right) < 2^{45.1},$$

where the factors 2^9 and $\frac{1}{16}$ are derived from the over- and underdeterminedness of the respective systems.

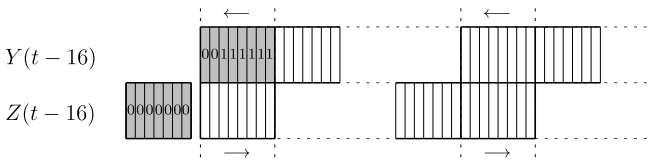

Fig. 14. Equation system at first byte position (6 Qs).

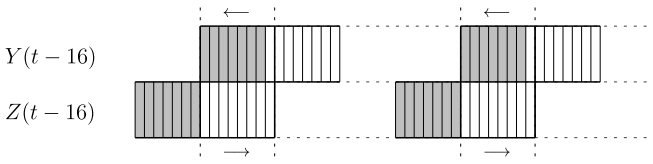

Fig. 15. Equation system at middle byte position (6 Qs).

Table 2. Costs for the X-FCSR-128 attack.

	Keystream	Solver	Storage
Basic attack w/o tables	$2^{64.0}$	$2^{45.1}$	–
Basic attack w/ tables	$2^{64.0}$	$2^{9.1}$	2^{60}
Reduced keystream attack Sections 5.1–5.3	$2^{55.2}$	$2^{16.3}$	2^{67}

The time complexity of recovering the state using six Qs *with* precomputation is given by

$$COST_{\text{solver}} = 1 + 2^9\left(1 + \frac{1}{16} + \frac{1}{16^2} + \cdots + \frac{1}{16^{13}} + \frac{1}{16^{14}}\right) < 2^{9.1}$$

in the basic setting with no relaxation of requirements **R2** and **R4**. The precomputational storage is now 2^{60}, again dominated by the middle byte position system.

We minimize $COST_{\text{keystream}}$ by using six Qs. With requirements **R2** and **R4** relaxed to at least 3 zero feedback bits and exploiting the symmetry ones-case, we obtain

$$COST_{\text{keystream}} < 2^{55.2}$$

with

$$COST_{\text{solver}} = 1 + 4 \cdot 6^2 \cdot 2^9\left(1 + \frac{1}{16} + \frac{1}{16^2} + \cdots + \frac{1}{16^{13}} + \frac{4 \cdot 6^2}{16^{14}}\right) < 2^{16.3}$$

using precomputational storage of size 2^{67}. The corresponding cost table for X-FCSR-128 is given in Table 2.

7. Summing Up the Attack

The results have been verified with simulations. Specifically, for X-FCSR-256 we have successfully recovered the entire state for all variations on the requirement set {**R1**, **R2**, **R3**, **R4**} discussed above.

The total cost for state recovery in terms of table lookups is given by

$$COST = COST_{\text{keystream}} \times COST_{\text{solver}}.$$

To summarize, we have $COST < 2^{44.3+4.7} = 2^{49.0}$ for X-FCSR-256 using at most 2^{33} table entries of precomputational storage. This attack variant minimizes both keystream and total complexity. The corresponding cost for X-FCSR-128 is $COST < 2^{55.2+16.3} = 2^{71.5}$ using at most 2^{67} storage.

A high-level description of the algorithm may be specified as follows. Recall $Q(t)$ from to Eq. (8). The on-line part of the attack begins by calculating k consecutive such Q-values, $Q(i), t - k + 1 \leq i \leq t$, collectively denoted $Q(\cdot)$ below. For X-FCSR-256 and X-FCSR-128 we have $k = 3$ and 6, respectively. $Q(\cdot)$ is then analyzed byte by byte from left to right. A lookup table set T_1 is queried for plausible state configurations corresponding to the first byte position of $Q(\cdot)$. If solutions exist, we go on and query

table set T_2 for matching state configurations corresponding to the second byte position of $Q(\cdot)$, and so on. Two neighboring state configurations are said to be matching if they have identical variable assignments for their common variables.

Precomputation. Create lookup table sets T_i, one for each byte position $i = 1, 2, \ldots, n$ of $Q(\cdot)$. Each table set T_i contains lookup tables for all the different requirement variations for registers Y and Z discussed in Sections 4.2 and 5. The tables for the first byte position, $i = 1$, may be merged for efficiency.

The algorithm is easily described in terms of depth-first search, if one views the plausible state configurations as vertices in a tree in which two vertices are adjacent if and only if they represent matching solutions at neighboring byte positions. $Q(\cdot)$ corresponds to a forest, the solution space, in which each solution to the first byte position system generates a separate tree. A path of length $n - 1$ in this tree represents a permissible configuration for the entire state.

State recovery at time t.

1. Compute $Q(\cdot)$ according to Eq. (8).
2. Using the precomputed lookup table sets T_i above, perform a Depth-First Search into the solution space of $Q(\cdot)$.
   The state can be recovered if and only if a vertex at depth n is reached.

8. 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.

The attack presented here is not directly applicable to the newer ring-FCSRs presented in [3]. The desired LFSRization effect is much less likely to appear in ring-FCSRs since these allow multiple simultaneous feedbacks. After the publication of [14], new ring-FCSR versions of the F-FCSR family and X-FCSR-128 were presented in [3] and [5], respectively. All of these new proposals are unbroken at the time of writing.

References

[1] F. Arnault, T. Berger, F-FCSR: Design of a new class of stream ciphers, in *Fast Software Encryption 2005*, ed. by H. Gilbert, H. Handschuh. Lecture Notes in Computer Science, vol. 3557 (Springer, Berlin, 2005), pp. 83–97.

[2] F. Arnault, T. Berger, 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. Berger, C. Lauradoux, M. Minier, B. Pousse, A New Approach for F-FCSRs, in *Selected Areas in Cryptography—SAC 2009*, ed. by V. Rijmen, M.J. Jacobson Jr., R. Safavi-Naini. Lecture Notes in Computer Science, vol. 5867 (Springer, Berlin, 2009), pp. 433–448

[4] F. Arnault, T.P. Berger, C. Lauradoux, M. Minier. X-FCSR—a new software oriented stream cipher based upon FCSRs, in *Progress in Cryptology—INDOCRYPT 2007*, ed. by K. Srinathan, C. Pandu Rangan, M. Yung. Lecture Notes in Computer Science, vol. 4859/2007 (Springer, Berlin, 2007), pp. 341–350

[5] T. Berger, M. Minier, B. Pousse, Software oriented stream ciphers based upon FCSRs in diversified mode, in *Progress in Cryptology—INDOCRYPT 2009*, ed. by B. Roy, N. Sendrier. Lecture Notes in Computer Science, vol. 5922 (Springer, Berlin, 2009), pp. 119–135. doi:10.1007/978-3-642-10628-6_8

[6] ECRYPT. eSTREAM: ECRYPT Stream Cipher Project, IST-2002-507932. Available at http://www.ecrypt.eu.org/stream/. Last accessed on January 14, 2011

[7] M. Hell, T. Johansson, Breaking the F-FCSR-H stream cipher in real time, in *Advances in Cryptology—ASIACRYPT 2008*. Lecture Notes in Computer Science, vol. 5350/2008 (Springer, Berlin, 2008), pp. 557–569

[8] M. Hell, T. Johansson, Breaking the Stream Ciphers F-FCSR-H and F-FCSR-16 in Real Time. *J. Cryptol.* (2009)

[9] E. Jaulmes, 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

[10] E. Jaulmes, F. Muller, Cryptanalysis of the F-FCSR stream cipher family, in *Selected Areas in Cryptography—SAC 2005*, ed. by B. Preneel, S. Tavares. Lecture Notes in Computer Science, vol. 3897 (Springer, Berlin, 2005), pp. 36–50

[11] A. Klapper, M. Goresky, 2-adic shift registers, in *Fast Software Encryption'93*, ed. by R.J. Anderson. Lecture Notes in Computer Science, vol. 809 (Springer, Berlin, 1994), pp. 174–178

[12] A. Klapper, M. Goresky, Feedback shift registers, 2-adic span, and combiners with memory. *J. Cryptol.* **10**(2), 111–147 (1997)

[13] R. Pagh, F. F. Rodler. Cuckoo hashing. *J. Algorithms* **51**, 122–144 (2004)

[14] P. Stankovski, M. Hell, T. Johansson, An Efficient State Recovery Attack on X-FCSR-256, in *Fast Software Encryption 2009*. Lecture Notes in Computer Science, vol. 5665 (Springer, Berlin, 2009), pp. 23–37