Breaking the F-FCSR-H Stream Cipher in Real Time

Martin Hell and Thomas Johansson

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

Abstract. The F-FCSR stream cipher family has been presented a few years ago. Apart from some flaws in the initial propositions, corrected in a later stage, there are no known weaknesses of the core of these algorithms. The hardware oriented version, called FCSR-H, is one of the ciphers selected for the eSTREAM portfolio.
In this paper we present a new and severe cryptanalytic attack on the F-FCSR stream cipher family. We give the details of the attack when applied on F-FCSR-H. The attack requires a few Mbytes of received sequence and the complexity is low enough to allow the attack to be performed on a single PC within seconds.

1 Introduction

The cryptographic scene include a variety of efficient and trusted block ciphers. However the same does not seem to hold for stream ciphers. The stream ciphers that have received attention through use in various standards tend to have more or less serious security weaknesses. Examples are A5 algorithms used in GSM, the RC4 algorithm used in for example WLAN applications through the WEP protocol and the E0 stream cipher used in Bluetooth.

Based on a belief that a dedicated stream cipher still has a capability of significantly outperforming a block cipher, the eSTREAM project was launched in 2004. The goal of this project was to solicit and evaluate submitted proposals of stream ciphers for future standardization. The main evaluation criteria set up were long-term security, efficiency in terms of performance, flexibility and market requirements.

The eSTREAM project considered two different profiles, one targeting software implemented stream ciphers; and one for hardware implemented stream ciphers (in particular constrained devices). The hardware category received a total of 25 submitted proposals. After three phases of evaluation, the final eSTREAM portfolio recommended four of them. One of them is a design called F-FCSR-H v2.

F-FCSR-H v2 is one of several algorithms in the F-FCSR family of stream ciphers designed by the French researchers F. Arnault, T.P. Berger, and C. Lauradoux. The family of ciphers is based on feedback with carry shift registers (FCSR) together with a filtering function. The idea of using FCSRs to generate sequences for cryptographic applications was initially proposed by Klapper

and Goresky in [6]. The F-FCSR family was introduced in [1], proposing four concrete constructions. These proposals were cryptanalyzed in [5]. The initial version submitted to eSTREAM, targeting hardware, was called F-FCSR-H. It was shown in [4] that this construction also had security problems. This lead to a change in the initialization procedure and the resulting algorithm was named F-FCSR-H v2. This paper will focus on the specification of F-FCSR-H v2 given in [2].

The eSTREAM class of hardware stream ciphers (and F-FCSR-H v2 in particular) prescribes a key of length 80 bits. Apart from the initial flaws (on the IV-setup procedure, and a TMD tradeoff attack), there are yet no known weaknesses of the core of these algorithms and the best attack on F-FCSR-H v2 is an exhaustive key search.

In this paper we present a new and severe cryptanalytic attack on the F-FCSR stream cipher family. We give the details of the attack when applied on F-FCSR-H v2. The attack is based on observing that the contribution of nonlinearity comes from the carry bits and that sometimes this contribution is too low and the system can be linearized. The whole attack require a few Mbytes of received sequence and the complexity is low enough to allow the attack to be performed on a single PC within seconds. The attack has been fully implemented using the designers' reference implementation.

In Section 2 we give an overview of the FCSR automaton and the F-FCSR construction. In Section 3 we then discuss the underlying weaknesses giving the attack. In Section 4 we give a description of the attack and in Section 5 we give a more detailed analysis of parts of the attack and we also give the estimated and simulated complexities. In Section 6 we give a rough outline of how the key could be reconstructed from a known state.

2 Recalling the FCSR automaton and the F-FCSR construction

Recall that a Feedback with Carry Shift Register (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 can assume that $q < 0 < p < |q|$. Following the notation from [2], the size n of the FCSR is the value such that $n+1$ is the bitlength of $|q|$. In the stream cipher construction, p depends on the secret key (and the IV), and q is a public parameter. The choice of q induces some properties of the FCSR. The most important one is that it completely determines the length of the period T of the keystream. The conditions for an optimal choice as used in the F-FCSR family of stream ciphers are: q is a (negative) prime of bitsize $n+1$; the order of 2 modulo q is $|q|-1$; and $T = (|q|-1)/2$ is also prime. Furthermore, set $d = (1+|q|)/2$. Then the Hamming weight $W(d)$ of the binary expansion of d is checked to be not too small, say $W(d) > n/2$.

The FCSR automaton as described in [2] is one way to efficiently implement the generation of the 2-adic expansion sequence. It contains two registers: the main register $\mathbf{M}$ and the carries register $\mathbf{C}$. The main register $\mathbf{M}$ contains n

cells. Let $\mathbf{M} = (m_{n-1}, m_{n-2}, \ldots, m_1, m_0)$ and associate $\mathbf{M}$ to the integer $M = \sum_{i=0}^{n-1} m_i \cdot 2^i$.

Recall the positive integer $d = (1 + |q|)/2$ and its binary representation $d = \sum_{i=0}^{n-1} d_i \cdot 2^i$. The carries register contains l active cells where $l + 1$ is the number of nonzero d_i binary digits in d. The active cells are the ones in the interval $0 \leq i \leq n - 2$ and $d_{n-1} = 1$ always hold. For this purpose we write the carries register $\mathbf{C}$ as $\mathbf{C} = (c_{n-2}, c_{n-3}, \ldots, c_1, c_0)$ and associate $\mathbf{C}$ to the integer $C = \sum_{i=0}^{n-2} c_i \cdot 2^i$. Note that only l of the bits in $\mathbf{C}$ are active and the remaining ones are set to zero. Let the integer p be written as $p = \sum_{i=0}^{n-1} p_i \cdot 2^i$, where $p_i \in \{0, 1\}$. Then the 2-adic expansion of the number p/q is computed by the automaton given in Figure 1.

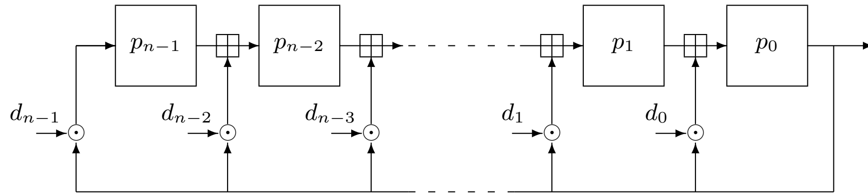

Fig. 1. Automaton to compute the 2-adic expansion of p/q.

The automaton is referred to as the Galois representation and it is very similar to the Galois representation of a usual LFSR. Other representations in connection with F-FCSR were considered in [7]. For all defined variables we also introduce a time index t, and let $\mathbf{M}(t)$ denote the content of $\mathbf{M}$ at time t. Similarly, $\mathbf{C}(t)$ denotes the content of $\mathbf{C}$ at time t.

The addition with carry, denoted $\boxplus$ in Figure 1, has a one bit memory (the carry). It takes three inputs in total, two external inputs and the carry bit. It outputs the XOR of the inputs and it sets the new carry value to one if the integer sum of the three inputs is two or three.

In Figure 2 we give an illustrating example (following [2]). Here $q = -347$ giving $d = 174$ and its binary expansion (10101110). The F-FCSR family of stream ciphers uses this particular automaton as the central part of their construction. So for future considerations in this paper we only need to recall the FCSR automaton as implemented in Figure 1 and Figure 2. Important facts are that the FCSR automaton has n bits of memory in the main register and l bits in the carry register, in total $n + l$ bits. If $(\mathbf{M}, \mathbf{C})$ 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.

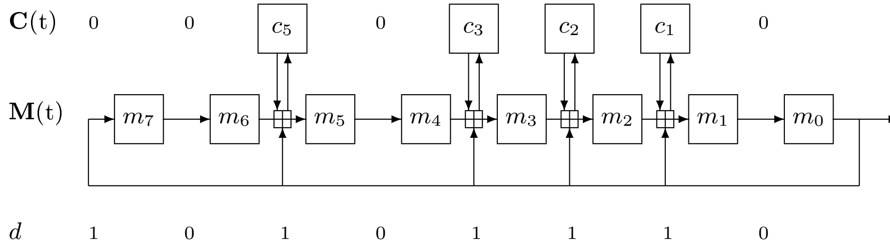

Fig. 2. Example of an FCSR.

2.1 Describing the F-FCSR-H Construction

The F-FCSR family of stream ciphers combines the FCSR automaton with a filtering function. The filtering function extracts keystream bits from the state of the main register in the FCSR automaton. The filter is a simple linear function of bits from the state. In order to increase the throughput, the constructions extract not only one but many bits each clock cycle. The number of extracted bits is eight for F-FCSR-H. Thus there are 8 different filters, now called subfilters, used to extract an 8 bits keystream byte after each transition of the automaton.

A one bit filter F is a bitstring $(f_0, \ldots, f_{n-1})$ of length n. The output bit of the filter is defined to be,

$$F(\mathbf{M}) = \bigoplus_{i=0}^{n-1} f_i m_i,$$

i.e., the scalar product. As F is a known string the output is a linear function (in F_2).

For the 8 bit filter, it consists of 8 such binary functions $F_0, F_1, \ldots, F_7$. However, filter F_j uses only cells m_i in the main register that satisfies $i = j \pmod 8$.

The parameters for F-FCSR-H are now given. The proposal uses key length 80 and IV of bitsize v with $32 \leq v \leq 80$. The core of the F-FCSR-H algorithm has remained identical to the one originally proposed in [1]. Only the key and IV initialization procedure was updated in [2].

The FCSR length (size of the main register) is $n = 160$. The carries register contains $l = 82$ cells. The feedback is determined by the prime

$$q = 1993524591318275015328041611344215036460140087963.$$

This gives

$$d = (1 + |q|)/2 = (\text{AE985DFF 26619FC5 8623DC8A AF46D590 3DD4254E})$$

(hexadecimal notation). So addition boxes and carries cells are present at the positions matching the binary ones in the binary expansion of d. To extract one keystream byte, FCSR-H uses the static filter

$$F = d = (AE985DFF26619FC58623DC8AAF46D5903DD4254E).$$

Using the designers notation, this means that the 8 subfilters (subfilter j is obtained by selecting the bit j in each byte of F) are given by

$$F_0 = (00110111010010101010),\ F_4 = (01110010001000111100),$$
$$F_1 = (10011010110111000001),\ F_5 = (10011100010010001010),$$
$$F_2 = (10111011101011101111),\ F_6 = (00110101001001100101),$$
$$F_3 = (11110010001110001001),\ F_7 = (11010011101110110100).$$

So the F-FCSR-H generator outputs one byte every time instance and it is simply given as

$$\mathbf{z} = (m_8 + m_{24} + m_{40} + m_{56} + \ldots + m_{136}, m_1 + m_{49} + \ldots, \ldots, m_{23} + \ldots).$$

The key and IV initialization consists of loading key and IV into the main register, clocking 20 times and extracting 20 bytes of output. These 160 bits are used as initial state in the main register of the FCSR automaton and it is clocked 162 times without producing output. More details are given in Section 6.

The second relevant construction in the F-FCSR family, called F-FCSR-16, is constructed in a similar manner. However it has a larger state and extracts 16 bits every clock cycle.

3 Weaknesses of the FCSR Automaton and the F-FCSR Family of Stream Ciphers

As the filtering function is F_2 linear, essentially all the security of the FCSR constructions rely on the FCSR automaton ability to create nonlinearity. It might at first glance look like this is achieved. The nonlinearity lies in the carry bit calculation, and carry bits are quickly spread over the entire main register. They enter new carry bit calculations, thus increasing the degree of nonlinear expressions rapidly. This is probably the first way one tries to analyze the construction, looking at the algebraic expressions created when the automaton is clocked a few times. It looks difficult to find some useful algebraic expression or some correlation between different variables that can be tracked all the way to the keystream symbols.

Instead, we look at the nonlinearity from a different perspective. The main observation we use is the fact that the carry bits in the carries register behave very far from random. The key point is that they all have one common input variable, the feedback bit. Let us look at what happens for a carry bit when the feedback bit is set to zero. We can see that when the feedback bit is zero then a carry bit that is zero must remain zero whereas if the carry bit is one then by probability 1/2 it will turn to zero (assuming random input on the active input). If we now assume that the feedback bit is zero a few consecutive time instances, then it is very likely that the carry bit is pushed to zero.

Actually, the same arguments can be repeated when the feedback bit is one. Then the carry is more likely to be one and by repeatedly having ones on the

feedback bit we push the carry value to one. However, for the moment we ignore this case.

Since the feedback bit is a common input to all carries, this has a dramatic effect on the carries vector **C**. We know that **C** has $l = 82$ active cells (carry bits) and we can expect that on average **C** will have a weight of 41. However, the weight is strongly correlated to the values of the feedback bit. Every time the feedback bit is zero all cells in **C** that are zero must remain zero, whereas those with value one has a 50% chance of becoming zero. So a zero feedback bit at time t gives a carries vector at time $t+1$ of roughly half the weight compared to time t. This behavior is easily checked by just running the generator and observing the contents of **C**.

Having found this crucial observation, the attack looks almost trivial. We assume that we have a number of consecutive feedback bits all zero. This would push the carries register to the all zero content. Then 19 more zero feedback bits to keep **C** zero all the time. During this time the generator outputs 20 bytes, or 160 bits. We can thus reconstruct the main register from knowing these values and the fact that **C** is zero. The only problem is that this does not work.

4 Describing the Attack

The underlying ideas of the attack were given in the previous section. However, the assumption that a large number of consecutive zero feedback bits would push the weight of **C** to zero is wrong. By simply running the generator we could see that this never happened. Once you look at the details, there is a simple explanation for this. Look at the FCSR automaton as illustrated in Figure 2, especially the last (least significant) active cell c_1 among the carries. Assume that the feedback bits are zero from time t to $t+t_0$ and the feedback bit at time $t-1$ was one. Now since the feedback bit at time $t-1$ was one and the feedback bits are zero from time t to $t+t_0$ the last carry addition must return zero to the next main register cell. Thus it must set the carry to one. Now, when the carry is one the only way we can have zero output and thus zero feedback is if the main register input to the last carry addition is one. Thus the last carry cell will never be pushed to zero, as we initially hoped. The fact that the carry vector and the feedback will not be zero for several consecutive clock cycles was actually observed in [3]. It was shown that this situation can not occur if the FCSR automata has reached a state of the main cycle, which is the case for all proposed F-FCSR stream ciphers.

However, this is not a problem. We slightly modify our approach and then it will work. As we described above, the all zero feedback sequence can appear if the main register input to the last carry addition is the all one sequence and we start with setting the carry bit to one. Then the all zero feedback will push the weight of **C** to one (the last active carry cell is always one). So it is natural to define the following event.

$$\text{Event } E_{\text{zero}} : \mathbf{C}(t) = \mathbf{C}(t+1) = \ldots = \mathbf{C}(t+19) = (0, 0, \ldots, 0, 1, 0).$$

When this happens we know that we have had 20 consecutive zeros in the feedback and that the carry has remained constant for 20 time instances. Using our previous arguments we would think that we need about $\log_2 82 \approx 7$ zeros in the feedback to push the weight of $\mathbf{C}$ to 1 and then an additional 19 zeros in the feedback to keep $\mathbf{C}$ constant for 20 time instances. Assuming a uniform distribution on the feedback bits this would lead to a probability of very roughly 2^{-26} for the event E_{zero} to happen. As we will see in the next section is it possible to use more information about the state in order to increase the efficiency of the attack. For now, let us just assume that we know how the main register $\mathbf{M}$ at time $t+1, t+2, \ldots, t+19$ depends on $\mathbf{M}(t)$ and that this dependency is linear.

Assuming that event E_{zero} occurs, the remaining part is to recover the main register from the given keystream bytes $\mathbf{z}(t), \mathbf{z}(t+1), \ldots, \mathbf{z}(t+19)$. This will lead to a linear system of equations with 160 equations in 160 unknowns. This could basically be solved through Gaussian elimination, costing something like 160^3 operations. However, we observe that the equations have the special byte structure explained before. There are 20 equations that only include the main register variables $m_0, m_8, m_{16}, \ldots, m_{152}$, there are 20 equations that only include $m_1, m_9, m_{17}, \ldots, m_{153}$, etc. Note that we are only shifting in zeros in $\mathbf{M}$ due to the assumption.

So it is much more efficient to treat each 20 by 20 system of equations independently. Let us describe the received systems of linear equations in more detail. We denote the least significant bit of $\mathbf{z}(t)$ by $\mathbf{z}(t)_0$, the next bit by $\mathbf{z}(t)_1$ etc, i.e., the output byte $\mathbf{z}(t)$ at time t is given by

$$\mathbf{z}(t) = (\underbrace{\mathbf{z}(t)_7}_{MSB}, \mathbf{z}(t)_6, \mathbf{z}(t)_5, \mathbf{z}(t)_4, \mathbf{z}(t)_3, \mathbf{z}(t)_2, \mathbf{z}(t)_1, \underbrace{\mathbf{z}(t)_0}_{LSB}). \tag{1}$$

Then the linear equations involving the main register bits m_i when $i \equiv 0 \mod 8$ at time t can be written as

$$\begin{aligned} \mathbf{z}(t)_0 &= m_8 \oplus m_{24} \oplus \ldots \oplus m_{136}, \\ \mathbf{z}(t+1)_7 &= m_{24} \oplus m_{40} \oplus \ldots \oplus m_{152}, \\ &\vdots \\ \mathbf{z}(t+19)_5 &= m_{32} \oplus m_{48} \oplus \ldots \oplus m_{152}. \end{aligned}$$

Similar equations containing only the main register bits m_i such that $i \equiv 1 \mod 8$ can also be listed. The same then goes for equations using only m_i bits when $i \equiv 2 \mod 8$, etc. Altogether, we can for simplicity write

$$\begin{aligned} \mathbf{W_0} &= (\mathbf{z}(t)_0, \mathbf{z}(t+1)_7, \ldots, \mathbf{z}(t+19)_5), \\ \mathbf{W_1} &= (\mathbf{z}(t)_1, \mathbf{z}(t+1)_0, \ldots, \mathbf{z}(t+19)_6), \\ &\vdots \\ \mathbf{W_7} &= (\mathbf{z}(t)_7, \mathbf{z}(t+1)_6, \ldots, \mathbf{z}(t+19)_4). \end{aligned}$$

The vector of main register values $m_0, m_8, m_{16}, \ldots, m_{152}$ is denoted $\hat{\mathbf{M}}_\mathbf{0}$. Then we get

$$\mathbf{W_0} = \hat{\mathbf{M}}_\mathbf{0} P_0, \quad (2)$$

where P_0 is a known 20 by 20 matrix (determined from the filter F). Similarly, $\hat{\mathbf{M}}_\mathbf{i}$, $1 \leq i \leq 7$ will denote the main register variables $(m_i, m_{i+8}, m_{i+16}, \ldots, m_{i+152})$. With this notation we can write the eight 20 by 20 linear systems of equations as

$$\mathbf{W_0} = \hat{\mathbf{M}}_\mathbf{0} P_0, \mathbf{W_1} = \hat{\mathbf{M}}_\mathbf{1} P_1, \ldots, \mathbf{W_7} = \hat{\mathbf{M}}_\mathbf{7} P_7. \quad (3)$$

Of course, some equations need to have 1 added to them since we have to compensate for the fact that the carry vector is given as $\mathbf{C} = (0, 0, \ldots, 0, 1, 0)$.

The idea is now to precompute, for each linear system, the solution $\hat{\mathbf{M}}_\mathbf{i}$ for each possible value of the vector of keystream bits $\mathbf{W_i}$. This would require 8 tables of size 2^{20} entries, each entry being a 20 bit vector. Though, the real time phase will be more efficient if 20 bytes are stored in each entry, having values only in the bit positions corresponding to the bits in $\hat{\mathbf{M}}_\mathbf{i}$. Then a full candidate state can be found by just ORing together the 8 saved contributions.

Finding the main register content would then require only to compute the vectors $\mathbf{W_i}$, $0 \leq i \leq 7$ from the keystream and then 8 table lookups to get the candidate main register state. The part of a candidate main register state given by $\mathbf{W_i}$ is denoted $\text{TABLE}_i[\mathbf{W_i}]$.

We can note that the P_i matrices are not all of full rank. This means that for our table of solutions, some $\mathbf{W_i}$ values will have no solutions whereas other values will have multiple (a power of two) solutions. This fact will then be combined over all 8 systems of equations, leading to a total number of $S = \prod_{i=0}^{7} s_i$ solutions, where s_i is the number of solutions to the ith system. Thus $\text{TABLE}_i[\mathbf{W_i}]$ returns a set of zero or more solutions.

In our case this property will increase the efficiency of the attack because if we get a value $\mathbf{W_0}$ for which $\text{TABLE}_0[\mathbf{W_0}]$ returns no solutions we can immediately stop and conclude that our assumption of event E_{zero} was wrong.

We now summarize our attack as follows.

```
0. for t = 1 to T_max do
1.   Select the 20 consecutive output bytes z(t), z(t+1), ..., z(t+19).
     for i = 0 to 7
       Compute W_i
       if TABLE_i[W_i] has no solutions
         go to 0.
       else
         store all possible values for M̂_i.
     end for
3.   "Check candidate states": Test all possible values of (M̂_0, M̂_1, ..., M̂_7),
     by checking if a candidate value generates z(t+20), z(t+21), ....
4. go to 0.
```

5 Improving the Attack Complexity

In the previous section we assumed that the carry vector was fixed to $\mathbf{C}(t) = \mathbf{C}(t+1) = \ldots = \mathbf{C}(t+19) = (0,0,\ldots,0,1,0)$ for all considered time instances. However we note that this is not necessary. As long as we can express the output bits in $\mathbf{z}(t), \mathbf{z}(t+1), \ldots, \mathbf{z}(t+19)$ as linear equations in the main register variables at time t, the attack will work.

Denote the state at time t as $(\mathbf{M},\mathbf{C})(t)$ and let x represent bits in the state that the output can be expressed as linear combinations of. Let ? represent bits that we do not need to know the value of. Assume that the state $(\mathbf{M},\mathbf{C})(t)$ is given by

$$(\mathbf{M},\mathbf{C})(t) = (xx\ldots xx0\underbrace{11\ldots11}_{16}00, 000\ldots0010).$$

Then, the state will be updated as

$$\begin{aligned}
(\mathbf{M},\mathbf{C})(t+1) &= (xx\ldots xx0\underbrace{11\ldots11}_{15}00, 000\ldots0010),\\
(\mathbf{M},\mathbf{C})(t+2) &= (xx\ldots xx0\underbrace{11\ldots11}_{14}00, 000\ldots0010),\\
&\vdots\\
(\mathbf{M},\mathbf{C})(t+15) &= (xxxxxxxx\ldots xx0100, 000\ldots0010),\\
(\mathbf{M},\mathbf{C})(t+16) &= (xxxxxxxx\ldots xxx000, 000\ldots0010),\\
(\mathbf{M},\mathbf{C})(t+17) &= (xxxxxxxx\ldots xxxx10, 000\ldots0000),\\
(\mathbf{M},\mathbf{C})(t+18) &= (xxxxxxxx\ldots xxxxx1, 000\ldots0000),\\
(\mathbf{M},\mathbf{C})(t+19) &= (xxxxxxxx\ldots xxxxxx, ??????????).
\end{aligned}$$

The only difference from the case presented in the previous section is that we should not compensate for the carry bit when computing the state $(\mathbf{M},\mathbf{C})(t+18)$ and we need to compensate for the 1 in the feedback when computing the state $(\mathbf{M},\mathbf{C})(t+19)$. Note that the feedback used when calculating $(\mathbf{M},\mathbf{C})(t+19)$ will cause the carry vector to be unpredictable. However, only $\mathbf{M}(t+19)$ is used to extract $\mathbf{z}(t+19)$ and knowledge of the carry vector here is not necessary. Using these observations, we can conclude that we only require the carry vector to take the value $(0,0,\ldots,0,1,0)$ at least 17 consecutive time instances. Thus, we update the definition of E_{zero} to

$$\text{Event } E_{\text{zero}} : \mathbf{C}(t) = \mathbf{C}(t+1) = \ldots = \mathbf{C}(t+16) = (0,0,\ldots,0,1,0).$$

The probability of E_{zero} has been simulated using in total 2 TB data and 2000 different keys and is estimated to be

$$P(E_{\text{zero}}) = 2^{-25.3}. \tag{4}$$

Thus, we would expect that we need on average $2^{25.3}$ bytes of keystream to recover the state.

The attack using the observations from this section has been fully implemented. The low complexity of the attack allows it to be simulated targeting the full version of F-FCSR-H v2. Using 5000 random keys, the state was recovered using on average $2^{24.7}$ bytes of keystream. The success rate was 100%. The slightly lower amount of keystream which was observed compared to the expected amount can easily be accounted for. For each state there are many equivalent states and sometimes one of these equivalent states is recovered. As an example, if $\mathbf{C}(t) = \mathbf{C}(t+1) = \mathbf{C}(t+15) = (0, 0, \ldots, 0, 1, 0)$ but $\mathbf{C}(t-1) \neq (0, 0, \ldots, 0, 1, 0)$, then $(\mathbf{M}, \mathbf{C})_{t-1}$ can be recovered if it is equivalent to another state $(\mathbf{M}', \mathbf{C}')$ with $\mathbf{C}' = (0, 0, \ldots, 0, 1, 0)$. Since the two states will merge after a few clocks, the attack will also recover the real state.

A slight improvement of the attack is achieved by noting that we can also look at the situation when the carry vector is one in all active positions except the last. The required keystream length will be halved, but the attack time will remain unchanged. The same simulation was performed with this improvement and as expected the state was recovered using on average $2^{23.7}$ bytes of keystream.

6 Recovering the key

We have described a state recovery attack that completely breaks F-FCSR-H. We now outline how we can also derive the key from a known state at any time t. In order to shortly describe this, we recall the initialization from the design document (reference code). Inputs to the initialization are a key K of length 80 bits and an IV of length $v \leq 80$ bits. For simplicity we fix the IV length to 80 bits.

Key+IV setup:

1. The main register $\mathbf{M}$ is initialized with key and IV by
$$\mathbf{M} = K + 2^{80} IV = (IV || K),$$
and the carries register $\mathbf{C} = 0$.
2. A loop is iterated 20 times. Each iteration of this loop consists in clocking the FCSR and then extracting a pseudorandom byte $S_i (0 \leq i \leq 19)$ using the filter.
3. The main register $\mathbf{M}$ is reinitialized with these bytes:
$$\mathbf{M} = (S_{19}, S_{18}, \ldots, S_0),$$
and $\mathbf{C} = 0$.
4. The FCSR is clocked 162 times (output is discarded).

Keystream generation:
Keystream is produced by first clocking the FCSR, then extracting one pseudorandom byte using filter F as described before.

Let us assume that time $t = 0$ appears directly after **3.** in the initialization above, i.e.,

$$\mathbf{M}(0) = (S_{19}, S_{18}, \dots, S_0).$$

Recall from Section 2 that every state $(\mathbf{M}, \mathbf{C})$ is associated with an integer p, $1 \leq p \leq |q|$, as the state generate the 2-adic expansion of p/q, where $p = M + 2C$. Let us write the value of p at time t as $p(t)$.

Now assume that we have recovered the state $\mathbf{M}$ and the carries register $\mathbf{C}$ at some time t. So $p(t)$ is known. Thus $p(0)$ can be derived since $p(0) = p(t) \cdot 2^t \bmod q$. This gives us knowledge of $\mathbf{M}(0) = (S_{19}, S_{18}, \dots, S_0)$, since the carries register at time 0 was 0.

Recall that $(S_{19}, S_{18}, \dots, S_0)$ was the output from F-FCSR-H when the main register was initialized with IV and key bits with $\mathbf{C} = 0$. If we for simplicity assume that $IV = 0$, then the remaining problem is to reconstruct the key bits. We give a rough outline on how such a reconstruction could be done. A more careful analysis might reveal more efficient ways to solve the problem.

The main register starts as $\mathbf{M} = (0^{80} || k_{79} k_{78} \dots k_1 k_0)$ and $\mathbf{C} = 0$. The FCSR is clocked once before any output.

We start by guessing the first 8 key bits $k_7, k_6, \dots, k_0$ that control the feedback the first 8 output bytes. With known feedback we can describe how every state bit can be expressed in algebraic form. Note that as long as we have zero feedback the carries register remain zero and we just get linear equations from the output bytes. The nonlinearity starts to grow when feedback is one. So assuming that the first feedback bit is one, we can examine the equations from the output bytes.

Similarly as before, let $\hat{\mathbf{K}}_\mathbf{0} = (k_0, k_8, \dots, k_{72})$, $\hat{\mathbf{K}}_\mathbf{1} = (k_1, k_9, \dots, k_{73})$, etc. Let $\mathcal{L}_i(\hat{\mathbf{K}}_\mathbf{i})$ denote some linear function of variables in $\hat{\mathbf{K}}_\mathbf{i}$ and let $\mathcal{C}_i(\hat{\mathbf{K}}_{\mathbf{i_1}}, \hat{\mathbf{K}}_{\mathbf{i_2}}, \dots, \hat{\mathbf{K}}_{\mathbf{i_n}})$ denote some nonlinear function of variables in $\hat{\mathbf{K}}_{\mathbf{i_1}}, \hat{\mathbf{K}}_{\mathbf{i_2}}, \dots, \hat{\mathbf{K}}_{\mathbf{i_n}}$. Then the received equations for the first output byte have the form

$$\begin{aligned} (S_0)_7 &= \mathcal{L}_0(\hat{\mathbf{K}}_\mathbf{0}), \\ (S_0)_1 &= \mathcal{L}_1(\hat{\mathbf{K}}_\mathbf{1}), \\ &\vdots \\ (S_0)_6 &= \mathcal{L}_7(\hat{\mathbf{K}}_\mathbf{7}). \end{aligned}$$

The next output byte is written

$$\begin{aligned} (S_1)_6 &= \mathcal{L}_8(\hat{\mathbf{K}}_\mathbf{0}) + \mathcal{C}_8(\hat{\mathbf{K}}_\mathbf{7}), \\ (S_1)_7 &= \mathcal{L}_9(\hat{\mathbf{K}}_\mathbf{1}) + \mathcal{C}_9(\hat{\mathbf{K}}_\mathbf{0}), \\ &\vdots \\ (S_1)_5 &= \mathcal{L}_{15}(\hat{\mathbf{K}}_\mathbf{7}) + \mathcal{C}_{15}(\hat{\mathbf{K}}_\mathbf{6}), \end{aligned}$$

and then

$$
\begin{aligned}
(S_2)_5 &= \mathcal{L}_{16}(\hat{\mathbf{K}}_\mathbf{0}) + \mathcal{C}_{16}(\hat{\mathbf{K}}_\mathbf{6}, \hat{\mathbf{K}}_\mathbf{7}),\\
(S_2)_6 &= \mathcal{L}_{17}(\hat{\mathbf{K}}_\mathbf{1}) + \mathcal{C}_{17}(\hat{\mathbf{K}}_\mathbf{7}, \hat{\mathbf{K}}_\mathbf{0}),\\
\vdots \quad &\vdots\\
(S_2)_4 &= \mathcal{L}_{23}(\hat{\mathbf{K}}_\mathbf{7}) + \mathcal{C}_{23}(\hat{\mathbf{K}}_\mathbf{5}, \hat{\mathbf{K}}_\mathbf{6}),
\end{aligned}
$$

and so on. The last one we use is

$$
\begin{aligned}
(S_7)_0 &= \mathcal{L}_{56}(\hat{\mathbf{K}}_\mathbf{0}) + \mathcal{C}_{56}(\hat{\mathbf{K}}_\mathbf{1}, \ldots, \hat{\mathbf{K}}_\mathbf{6}, \hat{\mathbf{K}}_\mathbf{7}),\\
(S_7)_6 &= \mathcal{L}_{57}(\hat{\mathbf{K}}_\mathbf{1}) + \mathcal{C}_{57}(\hat{\mathbf{K}}_\mathbf{2}, \ldots, \hat{\mathbf{K}}_\mathbf{7}, \hat{\mathbf{K}}_\mathbf{0}),\\
\vdots \quad &\vdots\\
(S_7)_4 &= \mathcal{L}_{63}(\hat{\mathbf{K}}_\mathbf{7}) + \mathcal{C}_{63}(\hat{\mathbf{K}}_\mathbf{0}, \ldots, \hat{\mathbf{K}}_\mathbf{5}, \hat{\mathbf{K}}_\mathbf{6}).
\end{aligned}
$$

When $\hat{\mathbf{K}}_\mathbf{i}$ appears in the linear expression but not in the nonlinear expression in an equation, we can use the equation to eliminate one variable. Starting with $\hat{\mathbf{K}}_\mathbf{7}$ we have 8 such equations. Since we guessed the first key byte $\hat{\mathbf{K}}_\mathbf{7}$ contains 9 unknown variables. By leaving or guessing one bit in $\hat{\mathbf{K}}_\mathbf{7}$ we can derive the remaining ones as functions $\mathcal{C}(\hat{\mathbf{K}}_\mathbf{0}, \ldots, \hat{\mathbf{K}}_\mathbf{5}, \hat{\mathbf{K}}_\mathbf{6})$. These functions are inserted instead of $\hat{\mathbf{K}}_\mathbf{7}$ variables in the remaining equations. Then examining the equations and looking for those with $\hat{\mathbf{K}}_\mathbf{6}$ only in the linear part gives 7 more equations that can be used to eliminate $\hat{\mathbf{K}}_\mathbf{6}$ variables. Then the same for $\hat{\mathbf{K}}_\mathbf{5}$ gives 6 more equations etc. Altogether we can remove 36 variables in this way and we have to do a work effort of trying 2^{44} choices of certain key bits. The algebraic expressions we need to test can be precomputed. Observe that if the first feedback bit is zero (probability 1/2) the complexity drops to 2^{36}, two zero feedback bits give complexity 2^{28}, etc.

The key recovery part has not been fully implemented but the given arguments show that also key recovery can be done with low complexity.

7 Conclusions

We have given a very strong attack on the F-FCSR-H stream cipher, a cipher that has been selected for the eSTREAM portfolio. The state recovery attack has been fully implemented to attack F-FCSR-H using the designers reference code. It succeeds in a few seconds using on average $2^{23.7}$ bytes (≈ 13 Mbyte) of keystream.

The weakness that was exploited is that the FCSR automata sometimes temporarily (almost) behaves as a regular LFSR. Together with the fact that the output filter is linear, the complete cipher became temporarily linear, which allowed us to recover the internal state.

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. Berger, and M. Minier. Some results on FCSR automata with applications to the security of FCSR-based pseudorandom generators. IEEE-IT, p.836-840, v.54, N.2, February 2008.
4. 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*.
5. 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.
6. 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.
7. S. Fischer, W. Meier, D. Stegemann. Equivalent representations of the F-FCSR Keystream Generator. In *SASC 2008, Workshop Record*, pages 87–96.