J. Cryptol. (2011) 24: 427–445
DOI: 10.1007/s00145-009-9053-2

Journal of
CRYPTOLOGY

Breaking the Stream Ciphers F-FCSR-H and F-FCSR-16 in Real Time[*]

Martin Hell and Thomas Johansson
Department of Electrical and Information Technology, Lund University, Box 118 22100 Lund, Sweden
Martin.Hell@eit.lth.se; Thomas.Johansson@eit.lth.se

Communicated by Willi Meier

Received 24 March 2009 and revised 9 November 2009
Online publication 29 October 2009

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. Two variants, F-FCSR-H and F-FCSR-16, were proposed in the eSTREAM project, and F-FCSR-H v2 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 to F-FCSR-H v2 and F-FCSR-16. 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.

Key words. Stream cipher, Cryptanalysis, F-FCSR-H, F-FCSR-16, Linearization.

1. Introduction

The cryptographic scene includes 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 was not a standardization body, like the AES project or the ongoing SHA-3 project. The

[*] This paper was solicited by the Editors-in-Chief as one of the best papers from Asiacrypt 2008, based on the recommendation of the program committee.

goal was to stimulate work in the area of stream ciphers. As an example, designers were allowed to tweak their proposals in the early phases of the project.

The eSTREAM project considered two different profiles, one targeting software implemented stream ciphers, profile 1; and one for hardware implemented stream ciphers (in particular constrained devices), profile 2. 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 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 [21]. The F-FCSR family was introduced in [2], proposing four concrete constructions. These proposals were cryptanalyzed in [19]. The initial version submitted to eSTREAM, targeting hardware, was called F-FCSR-H. It was shown in [18] that this construction also had security problems. This led to a change in the initialization procedure, and the resulting algorithm was named F-FCSR-H v2. Also, a new variant called F-FCSR-16 was proposed. This variant outputs 16 bits in each register update compared to 8 bits in F-FCSR-H v2. This paper will focus on the specification of F-FCSR-H v2 and F-FCSR-16 given in [6]. A more comprehensive overview of stream ciphers based on FCSRs is given in Appendix A.

The eSTREAM class of hardware stream ciphers (and F-FCSR-H v2 in particular) prescribes a key of length 80 bits. F-FSCR-16 can be used with either 80 or 128 bit key, with the 80 bit choice targeting profile 2 (hardware) and the 128 bit choice targeting profile 1 (software) in eSTREAM. The constructions also use a public IV value of bit-size v which can be set in the interval $32 \leq v \leq 80$ for F-FCSR-H v2 and $32 \leq v \leq 128$ for F-FCSR-16.

Apart from exhaustive key search, there are a number of standard attacks that can be applied. Time-memory-data trade-off attacks of different kinds [10,13,16] are applicable, but due to fairly large state size (size of the FCSR), this does not give a successful attack. Correlation attacks and linear cryptanalysis techniques [25,26] are also possible approaches, but the nonlinearity of the carry in the FCSR makes this difficult, and no promising ideas in this direction have been proposed. Also algebraic attacks has been accounted for in the design. Another standard technique today is a chosen IV attack [14,27,29], but the latest versions of the F-FCSR have not shown weaknesses in the initialization. So apart from the initial flaws (on the IV-setup procedure, and a TMD tradeoff attack), there were no known weaknesses of the core of these algorithms until a preliminary version of this paper was presented in [17].

We present a new and severe cryptanalytic attack on the F-FCSR stream cipher family. We give the details of the attack when applied to F-FCSR-H v2. The attack is based on observing that the contribution of nonlinearity comes from the carry bits only and that sometimes this contribution is too low and the system can be linearized. We define this as an *event*, and when the event occurs, we show how we can very efficiently derive the whole state. The whole 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. The attack has been fully implemented using the designers' reference implementation. Simulations show that the attack requires $2^{24.7}$ keystream bytes for F-FCSR-H v2 and $2^{21.5}$ keystream bytes for F-FCSR-16.

Since only the initialization was changed when updating F-FCSR-H to F-FCSR-H v2, our state recovery attack will be identical for both these versions. For simplicity, we will refer to this stream cipher as just F-FCSR-H.

In Sect. 2 we give an overview of the FCSR automaton and the F-FCSR construction. In Sect. 3 we then discuss the underlying weaknesses giving the attack. In Sect. 4 we give a description of the attack on F-FCSR-H, and in Sect. 5 we give a more detailed analysis of parts of the attack and also give the estimated and simulated complexities. In Sect. 6 we apply the same attack to the stream cipher F-FCSR-16. In Sect. 7 we give a rough outline of how the key could be reconstructed from a known state. Finally, we end with some conclusions.

2. Recalling the FCSR Automaton and the F-FCSR Construction

Let us start by recalling some useful facts from the theory on p-adic numbers. In particular, we consider only 2-adic numbers. For a more detailed overview of this subject, we refer to any textbook on the subject (e.g., [24]) or the more comprehensive description in [22], from which the brief reminder hereafter is largely inspired.

A 2-adic number **a** can be represented by a sequence of bits representing a binary number

$$\mathbf{a} = \cdots a_2 a_1 a_0 . a_{-1} a_{-2} \cdots a_{-k},$$

where $a_i \in \{0, 1\}$. The sequence of bits extends infinitely to the left. A 2-adic number may also be represented as a formal power series $a = \sum_{i=-k}^{\infty} a_i 2^i$, $a_i \in \{0, 1\}$.

If we fix $k = 0$, i.e., we do not allow nonzero bits to the right of the binary point, we get the 2*-adic integers*. A 2-adic integer a is written as $a = \sum_{i=0}^{\infty} a_i 2^i$, where $a_i \in \{0, 1\}$. The set containing all such power series forms a ring with respect to usual addition and multiplication. This ring of 2-adic integers is denoted $\mathbb{Z}_{2\text{-adic}}$ (whereas $\mathbb{Z}_2$ denotes the ring of integers modulo 2). The characterizing property of $\mathbb{Z}_{2\text{-adic}}$ is that addition in the ring has a "carry" property and can be compared to a usual integer addition of two numbers in binary representation but with infinite word size. This may be illustrated by comparing $\mathbb{Z}_{2\text{-adic}}$ and the ring $\mathbb{Z}_2[[X]]$ of formal power series in X. In $\mathbb{Z}_{2\text{-adic}}$ addition is performed by moving overflow bits to higher-order terms, since $2^i + 2^i = 2^{i+1}$. In $\mathbb{Z}_2[[X]]$ we would instead have $X^i + X^i = 0$.

Multiplication among 2-adic integers is done by a shift and add procedure. We can also see that the additive and multiplicative identities are 0 and 1 $(= 2^0)$, respectively. Moreover, using the addition rule, it is easily seen that

$$1 + \left(1 + 2^1 + 2^2 + 2^3 + \cdots\right) = 0,$$

and, hence, $1 + 2^1 + 2^2 + 2^3 + \cdots = -1$ in $\mathbb{Z}_{2\text{-adic}}$. If we write the integer q as

$$q = q_0 + q_1 2 + \cdots + q_r 2^r, \quad q_i \in \{0, 1\}, \quad 0 \le i \le r,$$

then

$$-q = \left(1 + 2^1 + 2^2 + 2^3 + \cdots\right)\left(q_0 + q_1 2 + \cdots + q_r 2^r\right).$$

We have demonstrated that $\mathbb{Z}_{2\text{-adic}}$ contains all the integers. So $\mathbb{Z} \subset \mathbb{Z}_{2\text{-adic}}$. Using long division, we can verify that every odd integer $q \in \mathbb{Z}$, i.e., integers with $q_0 = 1$, has a unique inverse in $\mathbb{Z}_{2\text{-adic}}$. This proves that every rational number p/q is in $\mathbb{Z}_{2\text{-adic}}$, provided that q is odd. This leads to the following characterization theorem.

Theorem 1 [22]. *There is a one-to-one correspondence between rational numbers $a = p/q$ (where q is odd) and eventually periodic binary sequences $\mathbf{a} = a_0a_1a_2\cdots$, which associates to each such rational number a, the bit sequence $a_0a_1a_2\cdots$ of its 2-adic expansion. The sequence $\mathbf{a}$ is strictly periodic if and only if $a \le 0$ and $|a| < 1$.*

See [22] for proof details. Furthermore, let $T = \text{ord}_q(2)$ denote the order of the element 2 in the multiplicative group $\mathbb{Z}_q \backslash \{0\}$, i.e., $T = \text{ord}_q(2)$ is the smallest integer such that $2^T \equiv 1 \pmod{q}$.

Corollary 1 (Gauss). *If p and q are relatively prime, $-q < p < 0$, and q is odd, then the period T of the bit sequence for the 2-adic expansion of $a = p/q$ is $T = ord_q(2)$.*

A Feedback with Carry Shift Register (FCSR) is a device that computes the binary expansion $\mathbf{a}$ of the 2-adic number $a = p/q$, where p and q are some fixed integers, with q odd. For simplicity, we assume that $q < 0 < p < |q|$.

Similar to LFSRs, an FCSR can be implemented either with Fibonacci or Galois representation. The Galois implementation is most suitable for hardware realization, and this is the implementation used for all stream ciphers in the F-FCSR family. Thus, we will restrict ourselves to this case in the following. Other representations in connection with F-FCSR were considered in [15].

Following the notation from [6], 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. From Corollary 1 we know that the choice of q completely determines the length of the period T of the keystream as $T = \text{ord}_q(2)$.

An FCSR is defined by choosing q to be a negative prime $2^n < -q < 2^{n+1}$ such that $T = \text{ord}_q(2) = |q| - 1$. Then, set $d = (1 - q)/2 = \sum_{i=0}^{n-1} d_i 2^i$ and check that the Hamming weight $W(d)$ of the binary expansion of d is not too small, say $W(d) > n/2$. Since $W(d)$ is the number of carry cells used in the FCSR, and the carry cells are the only components introducing nonlinearity, it is important that $W(d)$ is not too small.

An example, also used in, e.g., [2,19], of a Galois FCSR is given in Fig. 1. It contains two registers. One is the main register, denoted $\mathbf{M}$, and the other is the carry register $\mathbf{C}$. In a Galois FCSR 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 $M = \sum_{i=0}^{n-1} m_i 2^i$.

Recall the positive integer $d = (1 - q)/2$ and its binary representation $d = \sum_{i=0}^{n-1} d_i 2^i$. The carry 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 $0 \le i \le n - 2$, and $d_{n-1} = 1$ always hold. For this purpose, we write the carry register $\mathbf{C}$ as $\mathbf{C} = (c_{n-2}, c_{n-3}, \ldots, c_1, c_0)$ and associate $\mathbf{C}$ to the integer $C = \sum_{i=0}^{n-2} c_i 2^i$. Note that only l of the bits in $\mathbf{C}$ are active, and the remaining ones are set to zero. In the example given in Fig. 1, the parameters are given by $q = -347$, $d = 174$ with its binary expansion (10101110), $n = 8$, and $l = 4$.

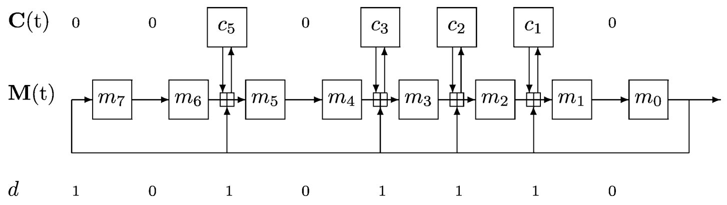

Fig. 1. Example of an FCSR.

For all defined variables, we also introduce a time index t and denote by $\mathbf{M}(t)$ the content of $\mathbf{M}$ at time t. Similarly, $\mathbf{C}(t)$ denotes the content of $\mathbf{C}$ at time t. The contents at time t of the individual cells in the FCSR are denoted $m_i(t)$ and $c_i(t)$.

The addition with carry, denoted $\boxplus$ in Fig. 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 sets the new carry value to one if the integer sum of the three inputs is two or three.

Let the integer p be written as $p = \sum_{i=0}^{n-1} p_i 2^i$, where $p_i \in \{0, 1\}$. Then the 2-adic expansion of the number p/q is computed by the automaton given in Fig. 1. To see this we write the output sequence $\mathbf{a}$ in its 2-adic representation $a = \sum_{t=0}^{\infty} a_t 2^t$. Each state value of the two registers represent a certain value of the integer p. The initial value of p is denoted $p(0)$, and as the FCSR is updated, p is updated as $p(0), p(1), p(2), \ldots$. Assume that a is the 2-adic expansion of p/q. Then we can write $p = q\sum_{t=0}^{\infty} a_t 2^t$. We define the sequence $p(t)$ as

$$p(t+1) = \frac{p(t) - a_t q}{2} \equiv 2^{-1} p(t) \pmod{q}. \tag{1}$$

To see why this definition is natural, consider

$$p(0) = q\big(a_0 + a_1 2 + a_2 2^2 + \cdots\big),$$

which corresponds to the sequence $a_0 a_1 a_2 \cdots$. According to (1), $p(1)$ is written as

$$p(1) = \frac{p(0) - a_0 q}{2} = \frac{q(a_0 + a_1 2 + a_2 2^2 + \cdots) - a_0 q}{2} = q\big(a_1 + a_2 2 + a_3 2^2 + \cdots\big),$$

which shows that $p(1)/q$ corresponds to the sequence $a_1 a_2 a_3 \cdots$, i.e., the sequence $\mathbf{a}$ shifted one step in time. Generalizing this, it is easy to see that $p(t)/q$ gives the sequence $a_t a_{t+1} a_{t+2} \cdots$ and $p(t) = q\sum_{j=t}^{\infty} a_j 2^{j-t}$. Also note that $a_t = p(t) \pmod 2$.

The integer M corresponding to the main register is updated as

$$M(t+1) = \frac{M(t) - m_0(t)}{2} + m_0(t)d + C(t) = \frac{M(t) + 2C(t) - m_0(t)q}{2}, \tag{2}$$

using the fact that $d = (1-q)/2$. Comparing (1) and (2), we see that if we write

$$p(t) = M(t) + 2C(t) \tag{3}$$

and let $p = p(0) = M(0)$, i.e., $C(0) = 0$, then the FCSR indeed generates the 2-adic expansion of p/q. Referring to Fig. 1, the output of the FCSR at time t is $m_0 = p(t) \pmod 2 = a_t$.

Finally, we note 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.

2.1. *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 eight different filters, now called subfilters, used to extract an 8-bit 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 eight 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 an IV of bitsize v with $32 \le v \le 80$. The FCSR length (size of the main register) is $n = 160$. The carry 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 carry cells are present at the positions matching the binary ones in the binary expansion of d. To extract one keystream byte, F-FCSR-H uses the static filter

$$F = d = (\text{AE985DFF 26619FC5 8623DC8A AF46D590 3DD4254E}).$$

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

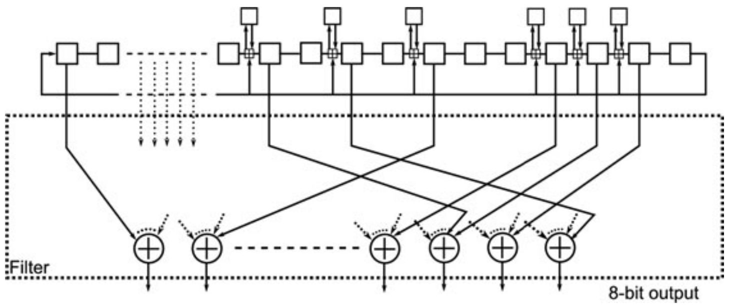

Fig. 2. An overview of F-FCSR-H and how the linear filter is used to produce the keystream.

$$F_0 = (0011011110100101010), \qquad F_4 = (0111001000100011100),$$
$$F_1 = (1001101011011000001), \qquad F_5 = (1001110001001001010),$$
$$F_2 = (1011101110101110111), \qquad F_6 = (0011010100100110010),$$
$$F_3 = (1111001000111000100), \qquad F_7 = (1101001110111011010).$$

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

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

Figure 2 gives an overview of the F-FCSR-H stream cipher and how the filter function is used to extract the output bits.

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 Sect. 7.

The second relevant construction in the F-FCSR family, called F-FCSR-16, is constructed in a similar manner. More details on this are given in Sect. 6.

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 relies 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 carry 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 carry vector $\mathbf{C}$. We know that $\mathbf{C}$ has $l = 82$ active cells (carry bits), and we can expect that on average $\mathbf{C}$ 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 $\mathbf{C}$ that are zero must remain zero, whereas those with value one have a 50% chance of becoming zero. So a zero feedback bit at time t gives a carry 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 $\mathbf{C}$.

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 carry register to the all-zero content. Then we have 19 more zero feedback bits to keep $\mathbf{C}$ 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 $\mathbf{C}$ 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 $\mathbf{C}$ to zero is wrong. By simply running the generator we could see that this never happened. Looking at the details, there is a simple explanation for this. If one considers the FCSR automaton as illustrated in Fig. 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 [9]. It was shown that this situation can not occur if the FCSR automaton 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. Slightly modifying the approach will make it 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 $\mathbf{C}$ to one (the last active carry cell is always one). So it is natural to define the following event:

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

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, it is 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 $t+1, t+2, \dots, t+19$ 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 $\mathbf{z}(t), \mathbf{z}(t+1), \dots, \mathbf{z}(t+19)$. 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 $m_0, m_8, m_{16}, \dots, m_{152}$, there are 20 equations that only include $m_1, m_9, m_{17}, \dots, m_{153}$, 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. Note that, for simplicity, in the presentation we disregard from the fact that we need to add a constant 1 to some equations, due to the fact that $c_1 = 1$. 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

$$\mathbf{z}(t) = (\underbrace{\mathbf{z}(t)_7}_{MSB}, \mathbf{z}(t)_6, \mathbf{z}(t)_5, \mathbf{z}(t)_4, \mathbf{z}(t)_3, \mathbf{z}(t)_2, \mathbf{z}(t)_1, \underbrace{\mathbf{z}(t)_0}_{LSB}). \tag{4}$$

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

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

Similar equations containing only the main register bits m_i such that $i \equiv 1 \pmod 8$ can also be listed. The same can then be done for equations using only bits m_i where $i \equiv 2$

(mod 8), etc. Altogether, we can for simplicity write

$$\begin{aligned}\mathbf{W}_0 &= \big(\mathbf{z}(t)_0, \mathbf{z}(t+1)_7, \dots, \mathbf{z}(t+19)_5\big),\\ \mathbf{W}_1 &= \big(\mathbf{z}(t)_1, \mathbf{z}(t+1)_0, \dots, \mathbf{z}(t+19)_6\big),\\ &\vdots\\ \mathbf{W}_7 &= \big(\mathbf{z}(t)_7, \mathbf{z}(t+1)_6, \dots, \mathbf{z}(t+19)_4\big).\end{aligned}$$

The vector of main register values $m_0, m_8, m_{16}, \dots, m_{152}$ is denoted $\hat{\mathbf{M}}_0$. Then we get

$$\mathbf{W}_0 = \hat{\mathbf{M}}_0 P_0, \tag{5}$$

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

$$\mathbf{W}_0 = \hat{\mathbf{M}}_0 P_0, \quad \mathbf{W}_1 = \hat{\mathbf{M}}_1 P_1, \quad \dots, \quad \mathbf{W}_7 = \hat{\mathbf{M}}_7 P_7. \tag{6}$$

The idea is now to precompute, for each linear system, the solution $\hat{\mathbf{M}}_i$ for each possible value of the vector of keystream bits $\mathbf{W}_i$. This would require eight 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 $\hat{\mathbf{M}}_i$. Then a full candidate state can be found by just XORing together the eight saved contributions.

Finding the main register content would then require only to compute the vectors $\mathbf{W}_i$, $0 \le i \le 7$, from the keystream and then eight 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 eight 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 $\mathbf{z}(t), \mathbf{z}(t+1), \dots, \mathbf{z}(t+19)$.
      for $i = 0$ to 7
        Compute $\mathbf{W}_i$
        if TABLE$_i[\mathbf{W}_i]$ has no solutions
          go to 0.
        else
          store all possible values for $\hat{\mathbf{M}}_i$.
      end for
3.    "Check candidate states": Test all possible values of $(\hat{\mathbf{M}}_0, \hat{\mathbf{M}}_1, \dots, \hat{\mathbf{M}}_7)$
      by checking if a candidate value generates $\mathbf{z}(t+1), \mathbf{z}(t+2), \dots$.
4.  go to 0.
```

Note that we are still guessing a part of the state (the carry bits), so the computed candidate main register state may not produce $\mathbf{z}(t+1), \mathbf{z}(t+2), \dots, \mathbf{z}(t+19)$.

5. Improving the Attack Complexity

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

$$(\mathbf{M}, \mathbf{C})(t) = (xx \cdots xx0\underbrace{11 \cdots 11}_{16}00, 000 \cdots 0010).$$

Then, the state will be updated as

$$(\mathbf{M}, \mathbf{C})(t+1) = (xx \cdots xx0\underbrace{11 \cdots 11}_{15}00, 000 \cdots 0010),$$

$$(\mathbf{M}, \mathbf{C})(t+2) = (xx \cdots xx0\underbrace{11 \cdots 11}_{14}00, 000 \cdots 0010),$$

$$\vdots$$

$$(\mathbf{M}, \mathbf{C})(t+15) = (xxxxxxxx \cdots xx0100, 000 \cdots 0010),$$

$$(\mathbf{M}, \mathbf{C})(t+16) = (xxxxxxxx \cdots xxx000, 000 \cdots 0010),$$

$$(\mathbf{M}, \mathbf{C})(t+17) = (xxxxxxxx \cdots xxxx10, 000 \cdots 0000),$$

$$(\mathbf{M},\mathbf{C})(t+18) = (xxxxxxxx\cdots xxxxx1, 000\cdots 0000),$$
$$(\mathbf{M},\mathbf{C})(t+19) = (xxxxxxxx\cdots xxxxxx, ??????????).$$

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 $(0,0,\dots,0,1,0)$ at least 17 consecutive time instances. Thus, we update the definition of E_{zero} to

$$\text{Event } E_{\text{zero}} : \mathbf{C}(t) = \mathbf{C}(t+1) = \cdots = \mathbf{C}(t+16) = (0,0,\dots,0,1,0). \qquad (7)$$

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

$$P(E_{\text{zero}}) = 2^{-25.3}. \qquad (8)$$

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

Further, we can note that there is a dual of the event E_{zero}, denoted E_{one}. Analogously to (7), we define this event as

$$\text{Event } E_{\text{one}} : \mathbf{C}(t) = \mathbf{C}(t+1) = \cdots = \mathbf{C}(t+16) = d' \text{ XOR } 2_{16}, \qquad (9)$$

where d' is the hexadecimal representation of the least $n-1$ significant bits of d. In other words, this is the event when all *active* carries are set to one, except the last. In this case we will have ones in the feedback, and the main register state at time t is given by

$$\mathbf{M}(t) = (xx\cdots xx1\underbrace{00\cdots 00}_{16}11).$$

Simulations show that this event occurs with the same probability, i.e., $P(E_{\text{one}}) = P(E_{\text{zero}}) = 2^{-25.3}$. When recovering the internal state, we do not know which event has occurred so both events need to be tested. Thus, the computational complexity remains the same, but the amount of expected keystream will be halved, and we expect that we need about $2^{24.3}$ keystream bytes.

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. Using 5000 random keys, the state was recovered using on average $2^{23.7}$ 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 $\mathbf{C}(t) = \mathbf{C}(t+1) = \mathbf{C}(t+15) = (0,0,\dots,0,1,0)$ but $\mathbf{C}(t-1) \neq (0,0,\dots,0,1,0)$, then $(\mathbf{M},\mathbf{C})_{t-1}$ can be recovered if it is equivalent to another state $(\mathbf{M}',\mathbf{C}')$ with $\mathbf{C}' = (0,0,\dots,0,1,0)$. Since the two states will merge after a few clocks, the attack will also recover the real state. The simulations were done using an AMD Athlon 64 X2 Dual Core Processor 4200+, 2.21 GHz and 4 GB RAM. The average time to recover the state was 10 seconds.

6. Applying the Attack to F-FCSR-16

The variant denoted F-FCSR-16 was proposed in [6]. By increasing the register size to 256 bits it was argued that it is possible to output 16 bits in each clock cycle instead of eight bits as in F-FCSR-H. There are in total 16 filter functions, $F_0, \dots, F_{15}$, each filtering out one bit from the main register. The filter F_j uses main register cells m_i that satisfies $i = j \pmod{16}$. For the exact definition of the filter functions, we refer to [6]. However, we note that, similar to F-FCSR-H, the last carry bit is not active and is thus not used in the filter either.

The attack as described on F-FCSR-H is immediately applicable to F-FCSR-16. However, since each output gives us 16 equations, and there are 256 main register bits, we only need $256/16 = 16$ consecutive time instances during which we know the register update. Thus, we define the two events

$$\text{Event } E_{\text{zero}} : \mathbf{C}(t) = \mathbf{C}(t+1) = \cdots = \mathbf{C}(t+12) = (0, 0, \dots, 0, 1, 0),$$
$$\text{Event } E_{\text{one}} : \mathbf{C}(t) = \mathbf{C}(t+1) = \cdots = \mathbf{C}(t+12) = d' \text{ XOR } 2_{16}.$$

Simulations show that $P(E_{\text{one}}) = P(E_{\text{zero}}) = 2^{-22.0}$, and we expect that we need $2^{21.0}$ keystream words ($2^{22.0}$ keystream bytes) to recover the state.

The state recovery attack was also implemented for F-FCSR-16, and since we sometimes recover equivalent states, the actual attack complexity is slightly lower than expected. The state is recovered using on average $2^{20.5}$ keystream words, i.e., $2^{21.5}$ keystream bytes. The average attack time is less than 2 seconds using an AMD Athlon 64 X2 Dual Core Processor 4200+, 2.21 GHz and 4 GB RAM.

7. Recovering the Key

We have described a state recovery attack that completely breaks F-FCSR-H and F-FCSR-16. We now outline how we can also derive the key from a known state at any time t. We give the details for F-FCSR-H v2, but the reasoning will also apply to F-FCSR-16. Note that the initialization was changed between F-FCSR-H and F-FCSR-H v2, and in this section we will focus on the latter. In order to shortly describe how to recover the key, 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

$$\mathbf{M} = K + 2^{80}\text{IV} = (\text{IV} \parallel K),$$

and the carry register is initialized by $\mathbf{C} = 0$.

2. A loop is iterated 20 times. Each iteration of this loop consists in clocking the FCSR and then extracting a pseudorandom byte $S_i\,(0 \le i \le 19)$ using the filter.
3. The main register **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 Step 3 in the initialization above, i.e.,

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

Recall from Sect. 2 that every state $(\mathbf{M}, \mathbf{C})$ is associated with an integer p, $1 \le p \le |q|$, as the state generates the 2-adic expansion of p/q, where $p = M + 2C$. The value of p at time t is written as $p(t)$.

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

Recall that $(S_{19}, S_{18}, \ldots, S_0)$ was the output from F-FCSR-H v2 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 $\mathbf{M} = (0^{80} \parallel k_{79}k_{78}\cdots k_1k_0)$ and $\mathbf{C} = 0$. The FCSR is clocked once before any output.

We start by guessing the first eight key bits $k_7, k_6, \ldots, k_0$ that control the feedback during the generation of the first eight 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 carry register remains zero, and we just get linear equations from the output bytes. The nonlinearity starts to grow when the 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 $\hat{\mathbf{K}}_0 = (k_0, k_8, \ldots, k_{72})$, $\hat{\mathbf{K}}_1 = (k_1, k_9, \ldots, k_{73})$, etc. Let $\mathcal{L}_i(\hat{\mathbf{K}}_i)$ denote some linear function of variables in $\hat{\mathbf{K}}_i$, and let $\mathcal{C}_i(\hat{\mathbf{K}}_{i_1}, \hat{\mathbf{K}}_{i_2}, \ldots, \hat{\mathbf{K}}_{i_n})$ denote some nonlinear function of variables in $\hat{\mathbf{K}}_{i_1}, \hat{\mathbf{K}}_{i_2}, \ldots, \hat{\mathbf{K}}_{i_n}$. Then the received equations for the first output byte have the form

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

The next output byte is written

$$\begin{aligned}(S_1)_6 &= \mathcal{L}_8(\hat{\mathbf{K}}_0) + \mathcal{C}_8(\hat{\mathbf{K}}_7),\\ (S_1)_7 &= \mathcal{L}_9(\hat{\mathbf{K}}_1) + \mathcal{C}_9(\hat{\mathbf{K}}_0),\\ \vdots\ &\ \ \vdots\\ (S_1)_5 &= \mathcal{L}_{15}(\hat{\mathbf{K}}_7) + \mathcal{C}_{15}(\hat{\mathbf{K}}_6),\end{aligned}$$

and then

$$\begin{aligned}(S_2)_5 &= \mathcal{L}_{16}(\hat{\mathbf{K}}_0) + \mathcal{C}_{16}(\hat{\mathbf{K}}_6, \hat{\mathbf{K}}_7),\\ (S_2)_6 &= \mathcal{L}_{17}(\hat{\mathbf{K}}_1) + \mathcal{C}_{17}(\hat{\mathbf{K}}_7, \hat{\mathbf{K}}_0),\\ \vdots\ &\ \ \vdots\\ (S_2)_4 &= \mathcal{L}_{23}(\hat{\mathbf{K}}_7) + \mathcal{C}_{23}(\hat{\mathbf{K}}_5, \hat{\mathbf{K}}_6),\end{aligned}$$

and so on. The last one we use is

$$\begin{aligned}(S_7)_0 &= \mathcal{L}_{56}(\hat{\mathbf{K}}_0) + \mathcal{C}_{56}(\hat{\mathbf{K}}_1, \ldots, \hat{\mathbf{K}}_6, \hat{\mathbf{K}}_7),\\ (S_7)_6 &= \mathcal{L}_{57}(\hat{\mathbf{K}}_1) + \mathcal{C}_{57}(\hat{\mathbf{K}}_2, \ldots, \hat{\mathbf{K}}_7, \hat{\mathbf{K}}_0),\\ \vdots\ &\ \ \vdots\\ (S_7)_4 &= \mathcal{L}_{63}(\hat{\mathbf{K}}_7) + \mathcal{C}_{63}(\hat{\mathbf{K}}_0, \ldots, \hat{\mathbf{K}}_5, \hat{\mathbf{K}}_6).\end{aligned}$$

When $\hat{\mathbf{K}}_i$ 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 $\hat{\mathbf{K}}_7$ we have eight such equations. Since we guessed the first key byte $\hat{\mathbf{K}}_7$ contains nine unknown variables. By leaving or guessing one bit in $\hat{\mathbf{K}}_7$ we can derive the remaining ones as functions $\mathcal{C}(\hat{\mathbf{K}}_0, \ldots, \hat{\mathbf{K}}_5, \hat{\mathbf{K}}_6)$. These functions are inserted instead of $\hat{\mathbf{K}}_7$ variables in the remaining equations. Then examining the equations and looking for those with $\hat{\mathbf{K}}_6$ only in the linear part gives seven more equations that can be used to eliminate $\hat{\mathbf{K}}_6$ variables. Then the same for $\hat{\mathbf{K}}_5$ gives six 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.

8. Conclusions

We have given a very strong attack on the F-FCSR-H v2 and the F-FCSR-16 stream ciphers. F-FCSR-H v2 was selected for the eSTREAM portfolio [11]. The state recovery attack has been fully implemented to attack both variants using the designers reference code. For F-FCSR-H (and F-FCSR-H v2), it succeeds in a few seconds using on average $2^{23.7}$ bytes ($\approx$13 Mbyte) of keystream. For F-FCSR-16, the amount of keystream needed is $2^{21.5}$ keystream bytes ($\approx$3 Mbyte).

Instead of having a single keystream from one IV, the attack can also be applied if we have only a few bytes, but instead from several different IVs. In that case we expect that one of the IVs will initialize the cipher such that our events occur almost immediately.

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.

Due to the weaknesses presented in this paper, F-FCSR-H v2 was removed from the eSTREAM portfolio [12].

Acknowledgements

We thank the reviewers for several useful comments.

Appendix A. A Brief History of FCSR Based Stream Ciphers

There are several examples of stream ciphers based on FCSRs. In particular, the stream cipher family Filtered FCSR (F-FCSR) consists of several different constructions. In this appendix we give a brief overview of the different constructions and cryptanalysis of these. The idea of using FCSRs as a building block in stream ciphers was first given by Klapper and Goresky in [21]. A more comprehensive introduction to FCSRs was later given in [22], a paper that also gives an algorithm for FCSR synthesis. This synthesis shows, similar to LFSR synthesis, how to reconstruct an FCSR from known output. However it was only shown for the case where we are in $\mathbb{Z}_{p\text{-adic}}$ where p is prime. A synthesis algorithm that could be applied to any p were later given in [23] and [4]. A survey of results on FCSRs prior to 2004 can be found in [20].

Neither LFSRs nor FCSRs can be used alone as a stream cipher. Both need another component in order not to be trivially breakable. An FCSR can be combined with a linear component, and an LFSR has to be combined with a nonlinear component. Based on these facts, a pseudorandom generator and a self-synchronizing stream cipher were proposed in 2002, combining an LFSR and an FCSR [3]. The self-synchronizing stream cipher was later broken using a chosen ciphertext attack [30]. In [1] Arnault and Berger proposed to use an FCSR together with a linear output filter. The idea was to filter out a subset of the main register cells by XORing them together. More specifically, all cells immediately to the right of a feedback tap were included in this filter. The paper did not specify an initialization algorithm, and how to incorporate an initialization vector was not discussed. Since this was not an explicit stream cipher proposal, the construction was not given a name other than the F-FCSR generator and can thus be seen as a general method of how to construct stream ciphers using FCSRs with linear output filters. Instead, four more explicit designs were proposed in [2]. The first was called F-FCSR-SF1, which was equivalent to the design discussed in [1]. The second construction, called F-FCSR-SF8, used eight static filters and output eight bits/clock. The last two constructions, F-FCSR-DF1 and F-FCSR-DF8, were similar, but instead of a static filter, it was proposed that the output filter would be key dependent. The initialization procedure proposed for these stream ciphers was to put the key in the main register, the IV in the carry register, and then to clock the generator six times, discarding the output, in order to make all main register cells dependent on the IV. Clocking six times as initialization does not prevent an attacker from learning differences between keystream bytes for initializations only differing in one IV bit. This was shown and exploited by Jaulmes and Muller in [19]. By guessing a subset of the key bits, wrong key candidates could be discarded using the knowledge of keystream differences. In the same paper, also a time-memory-data tradeoff attack was given, exploiting the fact that not all parts of the state provided entropy. All versions proposed in [2] were broken using these attacks.

The stream cipher proposals submitted to eSTREAM were denoted F-FCSR-H and F-FCSR-8. The former was targeting the hardware profile (profile 2), while the latter was targeting the software efficient profile (profile 1). The size of the main register in F-FCSR-H was 160 bits, and the initialization algorithm loaded key and IV into the main register, put the carry register to the zero vector, and clocked the generator 160 times, discarding the output. The filter function was static, outputting eight bits/clock. Thus, it was based on the idea behind F-FCSR-SF8. The software oriented variant, F-FCSR-8, had a main register of only 128 bits. It was based on the idea behind F-FCSR-DF8, outputting eight bits/clock using a key dependent filter. The filter was constructed by loading the main register with the key and clocking six times. If the content of the register passed a suitability test it was used as filter, otherwise the FCSR was clocked six more times, etc. The initialization algorithm clocked the FCSR 64 or 128 times depending on the size of the IV. While the number of initialization clocks were substantially increased compared to the previous version, it was shown in [18] that the generators were still prone to attacks on the initialization. A resynchronization attack applied to F-FCSR-H resulted in a distinguishing attack and on F-FCSR-8 it could also recover the key. The small state of F-FCSR-8 also made it prone to attacks.

In a short note [5] the designers presented how the algorithms could be tweaked in order to prevent the suggested attacks. The initialization algorithm of F-FCSR-H was changed. Instead of just clocking the register 160 times, as in the original version, the register was first clocked 20 times. The output was used to reinitialize the main register, and the FCSR was then clocked 162 times. For F-FCSR-8, the tweak was more substantial. The main register was increased to 256 bits. The initialization was tweaked similar to F-FCSR-H with the difference that it was clocked 258 times after reinitialization instead. Increasing the main register to 256 bits meant that the speed (in software) was halved. To compensate for this, the output filter (still key dependent) was designed to output 16 bits/clock instead of previously eight. The name was also changed to F-FCSR-8.2.

Before entering the second phase of eSTREAM, designers were given the opportunity to tweak algorithms. This opportunity was taken by the F-FCSR designers, and [6] describes the algorithms that were official in phase 2. The tweak to F-FCSR-H proposed in [5] were kept. Later, in phase 3, the algorithm is referred to as F-FCSR-H v2 on the eSTREAM web page, although this name is not explicitly mentioned in [6]. F-FCSR-8.2

as described in [5] was dropped. Instead, this stream cipher also adopted the idea of a static filter since a key dependent filter did not provide as high security as first expected. Other features from F-FSCR-8.2 were kept, namely 16 bits/clock with a register of size 256 bits. This new version was called F-FCSR-16 and was claimed suitable for both hardware and software.

While F-FCSR-16 targeted both the hardware and software profile of eSTREAM, it has no apparent advantages over AES in software as it runs at less than half the speed. Instead, two other stream ciphers based on FCSRs, called X-FCSR-128 and X-FCSR-256, were proposed in [7]. These ciphers were very efficient in software and used two FCSR together with an S-box. In [28], an attack on X-FCSR-256 were given. The attack used the weakness presented in this paper as one component in the attack.

In 2008, further results on FCSRs and FCSR-based stream ciphers were given in [9], which is the journal version of [8]. Among other things, the possibility that the carry register could be zero for several consecutive register updates was discussed. It was shown that all carry and register cells to the right of the rightmost feedback cell (in the case of the carry the cell at the same position as the feedback cell is included) could not be all zero at the same time if we are on the main cycle. We gave an intuitive explanation of this in Sect. 4.

It is clear that the weakness given in this paper can be used to attack several ciphers using FCSRs. However, the attractive properties of the sequences, together with the inherent nonlinearity of the FCSR, motivate further research on stream ciphers incorporating this building block.

References

[1] F. Arnault, T. Berger, Design and properties of a new pseudorandom generator based on a filtered FCSR automaton. *IEEE Trans. Comput.* **54**, 1374–1383 (2005)

[2] 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

[3] F. Arnault, T. Berger, A. Necer, A new class of stream ciphers combining LFSR and FCSR architectures, in *Progress in Cryptology—INDOCRYPT 2002*, ed. by A. Menezes, P. Sarkar. Lecture Notes in Computer Science, vol. 2551/2002 (Springer, Berlin, 2002), pp. 22–33

[4] F. Arnault, T. Berger, A. Necer, Feedback with carry shift registers synthesis with the Euclidean algorithm. *IEEE Trans. Inf. Theory* **50**(5), 910–917 (2004)

[5] F. Arnault, T. Berger, C. Lauradoux, Preventing weaknesses on F-FCSR in IV mode and tradeoff attack on F-FCSR-8. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/075 (2005). http://www.ecrypt.eu.org/stream

[6] 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

[7] 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

[8] F. Arnault, T. Berger, M. Minier, On the security of FCSR-based pseudorandom generators. eSTREAM, ECRYPT Stream Cipher Project, Report 2007/022 (2007). http://www.ecrypt.eu.org/stream

[9] F. Arnault, T. Berger, M. Minier, Some results on FCSR automata with applications to the security of FCSR-based pseudorandom generators. *IEEE Trans. Inf. Theory* **54**(2), 836–840 (2008)

[10] S. Babbage, A space/time tradeoff in exhaustive search attacks on stream ciphers. In *European Convention on Security and Detection*. IEE Conference Publication, vol. 408 (1995)

[11] S. Babbage, C. De Cannière, A. Canteaut, C. Cid, H. Gilbert, T. Johansson, M. Parker, B. Preneel, V. Rijmen, M.J.B. Robshaw, The eSTREAM portfolio (2008). Available via http://www.ecrypt.eu.org/stream

[12] S. Babbage, C. De Cannière, A. Canteaut, C. Cid, H. Gilbert, T. Johansson, M. Parker, B. Preneel, V. Rijmen, M.J.B. Robshaw, The eSTREAM portfolio (rev. 1) (2008). Available via http://www.ecrypt.eu.org/stream

[13] A. Biryukov, A. Shamir, Cryptanalytic time/memory/data tradeoffs for stream ciphers, in *Advances in Cryptology—ASIACRYPT 2000*, ed. by T. Okamoto. Lecture Notes in Computer Science, vol. 1976 (Springer, Berlin, 2000), pp. 1–13

[14] H. Englund, T. Johansson, M.S. Turan, A framework for chosen IV statistical analysis of stream ciphers, 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. 268–281

[15] S. Fischer, W. Meier, D. Stegemann, Equivalent representations of the F-FCSR keystream generator. The State of the Art of Stream Ciphers, Workshop Record, SASC 2008, Lausanne, Switzerland, February 2008

[16] J.D. Golić, Cryptanalysis of alleged A5 stream cipher, in *Advances in Cryptology—EUROCRYPT'97*, ed. by W. Fumy. Lecture Notes in Computer Science, vol. 1233 (Springer, Berlin, 1997), pp. 239–255

[17] 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

[18] 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

[19] 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

[20] A. Klapper, A survey of feedback with carry shift registers, in *Sequences and Their Applications—SETA 2004*, ed. by T. Helleseth, D. Sarwate, H. Song, K. Yang. Lecture Notes in Computer Science, vol. 3486/2005 (Springer, Berlin, 2004), pp. 56–71

[21] 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

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

[23] A. Klapper, J. Xu, Register synthesis for algebraic feedback shift registers based on non-primes. *Des. Codes Cryptogr.* **31**(3), 227–250 (2004)

[24] N. Koblitz, *P-adic Numbers, p-adic Analysis, and Zeta-Functions* (Springer, Berlin, 1996)

[25] M. Matsui, Linear cryptanalysis method for DES cipher, in *Advances in Cryptology—EUROCRYPT'93*, ed. by T. Helleseth. Lecture Notes in Computer Science, vol. 765 (Springer, Berlin, 1994), pp. 386–397

[26] W. Meier, O. Staffelbach, Fast correlation attacks on certain stream ciphers. *J. Cryptol.* **1**(3), 159–176 (1989)

[27] M.-J.O. Saarinen, Chosen-IV statistical attacks on eSTREAM stream ciphers. eSTREAM, ECRYPT Stream Cipher Project, Report 2006/013 (2006). http://www.ecrypt.eu.org/stream

[28] P. Stankovski, M. Hell, T. Johansson, An efficient state recovery attack on X-FCSR-256. Fast Software Encryption 2009 (2009). Preproceedings

[29] M. Vielhaber, Breaking ONE.FIVIUM by AIDA an algebraic IV differential attack (2007). Available at http://eprint.iacr.org/2007/413

[30] B. Zhang, H. Wu, D. Feng, F. Bao, Chosen ciphertext attack on a new class of self-synchronizing stream ciphers, in *Progress in Cryptology—INDOCRYPT 2004*, ed. by A. Canteaut, K. Viswanathan. Lecture Notes in Computer Science, vol. 3348/2004 (Springer, Berlin, 2004), pp. 73–83