Revisiting a Recent Resource-efficient Technique for Increasing the Throughput of Stream Ciphers

Frederik Armknecht and Vasily Mikhalev
Universität Mannheim, Mannheim, Germany

Keywords: Stream Ciphers, Feedback Shift Registers, Implementation, Throughput, Pipelining, Galois Configuration.

Abstract: At CT-RSA 2014, Armknecht and Mikhalev presented a new technique for increasing the throughput of stream ciphers that are based on Feedback Shift Registers (FSRs) which requires practically no additional memory. The authors provided concise sufficient conditions for the applicability of this technique and demonstrated its usefulness on the stream cipher Grain-128. However, as these conditions are quite involved, the authors raised as an open question if and to what extent this technique can be applied to other ciphers as well. In this work, we revisit this technique and examine its applicability to other stream ciphers. On the one hand we show on the example of Grain-128a that the technique can be successfully applied to other ciphers as well. On the other hand we list several stream ciphers where the technique is not applicable for different structural reasons.

1 INTRODUCTION

Stream ciphers are designed for efficiently encrypting data streams of arbitrary length. Ideally a stream cipher should not only be secure but also have a low hardware size and high throughput. Consequently several papers address the question of optimizing the hardware implementation of stream ciphers, e.g., (Gupta et al., 2013; Mansouri and Dubrova, 2010; Mansouri and Dubrova, 2013; Nakano et al., 2011; Stefan and Mitchell, 2008; Yan and Heys, 2007; Z. Liu and Pan, 2010).

Armknecht and Mikhalev (Armknecht and Mikhalev, 2014) presented a new technique for increasing the throughput of stream ciphers without (or only little) additional memory. The technique adapts the principle of pipelining of the output function. The core difference however is to identify ”unused” regions within the FSR for storing intermediate values, hence mitigating the need for additional memory. Here, ”unused” means registers of the FSRs that are only used for storing values and that are neither involved in the update function nor the output function.

The authors described concise sufficient conditions for the applicability of their technique and demonstrated it on the stream cipher Grain-128. However, as the conditions are quite complicated, the authors mentioned as an open question if and how this technique can be applied to other FSR-based stream ciphers as well. In this work, we revisit this technique and shed new light on this technique. More precisely, we provide both positive and negative results. On the positive side, we successfully apply this technique to Grain-128a. On the negative side we explain for several FSR-based stream ciphers that the technique cannot be used for structural reasons.

2 PRELIMINARIES

Let $\mathbb{F}_2$ denote the finite field $GF(2)$. For a Boolean function $f(x_0,\ldots,x_{n-1}) \in \mathbb{F}_2[x_0,\ldots,x_{n-1}]$, we define its *support* to be the smallest set of variables $\{x_{i_1},\ldots,x_{i_\ell}\} \subseteq \{x_0,\ldots,x_{n-1}\}$ which is required to specify f. That is $f \in \mathbb{F}_2[x_{i_1},\ldots,x_{i_\ell}]$ but $f \notin \mathbb{F}_2[X]$ for any real subset $X \subset \{x_{i_1},\ldots,x_{i_\ell}\}$.

A *Feedback Shift Registers (FSR)* is a regularly clocked finite state machine that is composed of a register and an update mapping $F = (f_0,\ldots,f_{n-1})$. F maps the whole state of a Feedback Shift Register to the new state and f_i refers to the individual update function of the i-th bit of the register. While in most cases the output of F only depends on the register values, in (Armknecht and Mikhalev, 2014) the authors considered a broader class of FSRs where also external values may be involved in the update procedure:

Definition 1 (FSR with external input)**.** A *FSR* FSR *with external input* of length n consists of an inter-

Armknecht F. and Mikhalev V..
Revisiting a Recent Resource-efficient Technique for Increasing the Throughput of Stream Ciphers.
DOI: 10.5220/0005059803790386
In *Proceedings of the 11th International Conference on Security and Cryptography* (SECRYPT-2014), pages 379-386
ISBN: 978-989-758-045-1

nal state of length n, an external source which produces a bit sequence $(b_t)_{t\geq 0}$, and update functions $f_i(x_0,\ldots,x_{n-1},y_0,\ldots,y_\ell)$ for $i = 0,\ldots,n-1$. Given some initial state $S_0 = (S_0[0],\ldots,S_0[n-1]) \in \mathbb{F}_2^n$, the following steps take place at each clock t:

1. The value $S_t[0]$ is given out and forms a part of the *output sequence*.
2. The state $S_t \in \mathbb{F}_2^n$ is updated to S_{t+1} where $S_{t+1}[i] = f_i(S_t, b_t, \ldots, b_{t+\ell-1})$.

We denote by $S_t[i]$ the state of the i-th register bit during the clock-cycle t, by S_t the whole state of the register during the clock-cycle t.

Two FSRs FSR and FSR$'$ of the same length with access to the same external source are called *equivalent*, denoted by FSR $\equiv$ FSR$'$, if for any initial state S_0 for FSR there exists an initial state S_0' for FSR$'$ (and vice versa) such that both produce the same output sequence for any external bit sequence $(b_t)_{t\geq 0}$. A transformation which takes as input some FSR FSR (and possible other inputs) and outputs a FSR FSR$'$ such that FSR $\equiv$ FSR$'$ is called *preserving*.

In (Armknecht and Mikhalev, 2014) the following class of stream ciphers has been considered. They deploy one or several regularly clocked finite state machines, typically including at least one FSR. At each clock several values of these components are fed into an output function h which eventually produces the current keystream bit. The stream ciphers contains on a high level three components only: a FSR FSR of length n, an output function h, and an external block EB. The external block EB is in principle a black box which may contain further FSRs, additional memory, etc. The only requirement is that a bitstream $(b_t)_{t\geq 0}$ can be specified which contains all bits produced inside of EB which are relevant for the state updates of FSR and/or for computing the next keystream bit. The notions of *equivalence* and *preserving transformation* given for FSRs (see above) are extended for for FSR-based stream ciphers in a straightforward manner.

In general any hardware implementation can be described by circuits. The time period between getting the input and producing the output is called its *delay*: Delay(C). Observe that circuits may run in parallel, e.g., for decreasing the delay of the overall circuit. Each of the connections between inputs, registers, and outputs of a stream cipher forms a *timing path*. The path which has the biggest delay is called *critical path*. It defines the maximum operating clock frequency of the cipher. For a given cipher maximum throughput is specified by the delay of its critical path. The amount of silicon used for the hardware implementation is called *area*. A common method to make the area consumption of different circuits comparable is to calculate the Gate Equivalence (GE) which is the total area divided by the lowest power two-input NAND gate area (Good and Benaissa, 2008).

3 THE CT-RSA 2014 TRANSFORMATION

In this section we summarize the technique presented in (Armknecht and Mikhalev, 2014). In a nutshell the authors described a preserving transformation for stream ciphers that fall into the categorization given in Sec. 2. The goal of the transformation is to reduce the total delay of a circuit by parallelizing computations within the output function such that the critical path becomes as short as possible.

To this end a part of the output function is integrated into *two* different update functions f_α and f_β of the FSR. At its first occurrence, i.e., in f_α, it alters the state entry to a value which can be used directly in the output function. At its second occurrence, i.e., in f_β, this value is cancelled out again, ensuring that the entry at index 0 (which defines the FSR output) is not affected by the transformation. The latter requires that the value is still extractable from the system, giving rise to the following definition:

Definition 2 (Function with Sustainable Output)**.** Consider a FSR-based stream cipher, being composed of an FSR FSR of length n with update mapping $F = (f_0,\ldots,f_{n-1})$, an external block with bit stream $(b_t)_{t\geq 0}$, and a function $h(x_0,\ldots,x_{n-1},y_0,\ldots,y_\ell)$. We say that h produces values which are *r-sustainable* if there exists a supplemental function $h^*(x_0,\ldots,x_{n-1},y_0,\ldots,y_\ell)$ such that for all $t \geq 0$:

$$h(S_t, b_t, \ldots, b_{t+\ell-1}) = h^*(S_{t+r}, b_{t+r}, \ldots, b_{t+r+\ell-1}).$$

Informally the definition means that the output of h at some clock t can likewise be computed r clocks later by h^* without requiring additional storage. The idea is now to identify parts of the output function that are sustainable for a number of clocks and to temporarily store its output in the FSR. However, care needs to be taken that change of the registers do not affect any other computations. Thus one needs intervals which are isolated with respect to the other functions used in the stream cipher. This has been formalized as follows:

Definition 3 (Isolated Interval)**.** Consider a FSR-based stream cipher, being composed of an FSR FSR of length n with update mapping $F = (f_0,\ldots,f_{n-1})$, a function $h(x_0,\ldots,x_{n-1},y_0,\ldots,y_\ell)$, and an external block with bit stream $(b_t)_{t\geq 0}$. An interval $[\alpha\ldots\beta]$ with $1 \leq \alpha \leq \beta \leq n-1$ of the FSR-state is *isolated* with respect to F and h if the following conditions are met:

1. The feedback functions $f_{\alpha-1},\ldots,f_{\beta-1}$ have all the form

$$f_i(x_0,\ldots,x_{n-1},y_0,\ldots,y_{\ell-1}) = x_{(i+1) \mod n} + g_i(x_0,\ldots,x_{n-1},y_0,\ldots,y_{\ell-1})$$

with $\mathrm{supp}(g_i)\cap\{x_\alpha,\ldots,x_\beta\}=\emptyset$. That is feedback function f_i is independent of the values at indices $\{\alpha,\cdots,\beta\}$ except of $i+1$ and the value at index $i+1$ is simply shifted.

2. The remaining feedback functions $f_0,\ldots,f_{\alpha-2},f_\beta,\ldots,f_{n-1}$ and the output function h are completely independent of the values at indices $\{\alpha,\cdots,\beta\}$, that is $\mathrm{supp}(f_i)\cap\{x_\alpha,\ldots,x_\beta\}=\emptyset$ for all $i\in\{0,\cdots,n-1\}\setminus\{\alpha,\cdots,\beta\}$.

We are now ready to repeat the transformation given in (Armknecht and Mikhalev, 2014). In a nutshell it works as follows:

- Identify a part of the output function which produces sustainable outputs and where an appropriate isolated interval does exist
- Remove this part from the output function and insert it at the update function at the beginning of the isolated interval
- For cancelling out this modification, insert the corresponding supplementary function into the update function at the end of the isolated interval

The following theorem specifies formally this transformation:

Theorem 1 (Preserving Cipher Transformation)**.** *(Armknecht and Mikhalev, 2014) Consider a FSR-based stream cipher $\mathcal{C}$, being composed of an FSR FSR of length n with update mapping $F=(f_0,\ldots,f_{n-1})$, an external block with bit stream $(b_t)_{t\geq 0}$, and an output function $h(x_0,\ldots,x_{n-1},y_0,\ldots,y_\ell)$. Assume that h can be written as $h(x_0,\ldots,x_{n-1},y_0,\ldots,y_\ell)=x_\beta+h_1()+h_2()$ such that the outputs of h_1 could be computed one clock earlier as well. Formally, this means that there exists a function $g((x_0,\ldots,x_{n-1},y_0,\ldots,y_\ell)$ such that it holds for all clocks $t\geq 1$:*

$$h(S_t,b_t,\ldots,b_{t+\ell-1}) = S_t[\beta]+ g(S_{t-1},b_{t-1},\ldots,b_{t+\ell-2})+h_2(S_t,b_t,\ldots,b_{t+\ell-1})$$

Moreover the following conditions need to be met:

1. *There exist integers $1\leq\alpha<\beta<n-1$ such that the interval $[\alpha,\ldots,\beta]$ is isolated with respect to F and h_2 and the interval $[\alpha+1,\ldots,\beta+1]$ is isolated with respect to g.*
2. *g produces $(\beta-\alpha)$-sustainable outputs with g^* being the corresponding supplementary function.*

A second cipher is defined as $\mathcal{C}'$ with an FSR FSR' and an output function h' which are derived from FSR and h, respectively. The update mapping $F'=(f'_0,\ldots,f'_{n-1})$ of FSR' is defined as $f'_{\alpha-1}:=f_{\alpha-1}+g^*$, $f'_\beta:=f_\beta+g$, and $f'_i:=f_i$ for all $i\neq\alpha,\beta$. and the output function h' of $\mathcal{C}'$ as $h'()=x_\beta+h_2()$. Then both ciphers are equivalent.

4 ON THE APPLICABILITY OF THE TRANSFORMATION

Obviously the conditions mentioned in Section 3 are quite involved, making it hard to estimate the general applicability of this approach. In fact although in (Armknecht and Mikhalev, 2014) the authors demonstrated its principle usefulness on the stream cipher Grain-128, they stated likewise the open question if it can be applied to other ciphers as well. This question will be considered in this section.

As explained in Sec. 3 the goal of the transformation is the outsource part of the computation of the output function into the FSR. In principle this technique may be useful for stream ciphers where the output is simply the XOR of several internal bits, e.g., Trivium (Cannière and Preneel, 2008) and MICKEY 2.0 (Babbage and Dodd, 2008), for example to reduce the overall area size. However we think that its main field of application are ciphers that deploy a complicated output function. Consequently, we consider in the following several ciphers that use a more involved output function: Grain-128a (Agren et al., 2011) (Sec. 4.1), DECIM v2 and DECIM-128 (Berbain et al., 2008) (Sec. 4.2), Achterbahn-128/80 (Gammel et al., 2007) (Sec. 4.3), and Hitag2 (Courtois et al., 2009) (Sec. 4.4). It will turn out that the transformation can be applied to Grain-128a and allows to improve the throughput. However, we will also explain why for *structural reasons* the transformation (in its current form) cannot be applied to the other ciphers.

4.1 Grain-128a

Specification. In (Armknecht and Mikhalev, 2014) the transformation has been successfully applied to the stream cipher Grain-128 (Hell et al., 2006), which is included into e-Stream portfolio (eSt,). However, in order to resist recently introduced cube attacks (Dinur and Shamir, 2011) the designers have improved the algorithm and proposed the Grain-128a stream cipher (Agren et al., 2011). Both variants consists of an 128-bit LFSR L with update mappings

$F = (f_0, \ldots, f_{127})$, an 128-bit NLFSR N with update mappings $Q = (q_0, \ldots, q_{127})$, and an output function h and have the same structure.

There are two main differences, that were introduced into the structure of Grain-128a compared to Grain-128. First, Grain-128a can operate in two different ways: with and without authentication. Second, the update functions of NLFSR N used in Grain-128a contains additional terms (compared to the case of Grain-128) so that the overall degree increased from two (Grain-128) to four (Grain-128a).

In (Mansouri and Dubrova, 2013) it was shown that the part of the circuit which is responsible for the authentication can be isolated from the encryption part of Grain-128a by using flip-flops. Moreover it has been confirmed that if such an instantiation is chosen, the critical path does not go through the authentication section and it has no effect on the delay of the whole circuit. For these reasons and to keep the presentation and analysis as simple as possible, we do not consider the authentication part of Grain-128a in this work and focus on the encryption part only.

We provide now a detailed specification of Grain-128a. We denote at clock t the state of the LFSR to be $L_t = (L_t[0], \cdots, L_t[127])$ and the state of the NLFSR as $N_t = (N_t[0], \cdots, N_t[127])$.

The update functions of the LFSR L and the NLFSR N are as follows:

$$
\begin{aligned}
L_{t+1}[i] &= L_t[i+1] \text{ and } N_{t+1}[i] = N_t[i+1] \\
&\quad \text{for } i = 0, \ldots, 126 \\
L_{t+1}[127] &= L_t[0] + L_t[7] + L_t[38] + L_t[70] + \\
&\quad L_t[81] + L_t[96] \\
N_{t+1}[127] &= L_t[0] + N_t[0] + N_t[26] + N_t[56] + \\
&\quad N_t[91] + N_t[96] + N_t[3]N_t[67] + \\
&\quad N_t[11]N_t[13]N_t[17]N_t[18] + \\
&\quad N_t[27]N_t[59] + N_t[40]N_t[48] + \\
&\quad N_t[61]N_t[65] + N_t[68]N_t[84] + \\
&\quad N_t[88]N_t[92]N_t[93]N_t[95] + \\
&\quad N_t[22]N_t[24]N_t[25] + \\
&\quad N_t[70]N_t[78]N_t[82]
\end{aligned}
$$

Note that both FSRs are specified in the so-called Fibonacci configuration, meaning that all bits except the 127th are updated just by shifting the adjacent value. This is a special case of the Galois configuration where these update functions can be more complex (see below).

The output function h of Grain-128a is almost the same as of Grain-128. The only difference is that in Grain-128a the variable $L_t[94]$ instead of $L_t[95]$ is used:

$$
\begin{aligned}
h &= N_t[2] + N_t[15] + N_t[36] + N_t[45] + N_t[64] + \\
&\quad N_t[73] + L_t[93] + N_t[89] + N_t[12]L_t[8] + \\
&\quad L_t[13]L_t[20] + N_t[95]L_t[42] + L_t[60]L_t[79] + \\
&\quad N_t[12]N_t[95]L_t[94]
\end{aligned}
$$

Grain-128a uses two different modes: initialization and keystream generation. In the keystream generation mode, the result of h forms the output. During the initializing mode the cipher does not produce any output for 256 clock-cycles. Instead the outputs of h are fed back to the LFSR and NLFSR.

To be able to determine possible improvements that result from the transformation, we made our own implementation of Grain-128a which serves as the basis for the further analysis. Our implementations were done using the Cadence RTL Compiler[1] for synthesis and simulation. Two implementations with different compiler settings were examined: optimizing the output for timing and optimizing for the area size. The implementation results for Grain-128a in Fibonacci configuration are the following.

1. When the compiler is set to *optimize timing*, the area size is 1888 GE and the maximum throughput in the initialization mode is 1,02 GHz, when in the keystream generation mode the maximum throughput is 1,23 GHz
2. When the compiler is set for *area optimization*, the area size is 1640 GEs and the maximum throughput in initialization mode is 0,57 GHz, when in keystream generation mode it is 0,84 GHz

Similar to the case of Grain-128 (see also (Armknecht and Mikhalev, 2014)), in the original specification of Grain-128a the critical path goes through the FSRs. Hence, a direct application of the transformation would yield no benefit. Therefore, we followed the same approach and first applied a transformation to the FSRs for reducing the delays within the FSRs. This had the effect that after the transformations of the FSRs, the criticial path were no longer going through the FSRs but through the output function. In particular a situation was created were the transformation from (Armknecht and Mikhalev, 2014) may help to further decrease the delay. In the following, we will first explain the FSRs transformation that we applied to the original instantiation and point out the throughput of the resulting instantiation (termed instantiation 1). Afterwards we explain a possible application of the transformation to instantiation

[1] See http://www.cadence.com/products/ld/rtl_compiler/pages/default.aspx

1 that results into instantiation 2. The change of the delay between instantiations 1 and 2 then gives the increase of the throughput gained by the transformation from (Armknecht and Mikhalev, 2014).

Specification of the FSRs. We used the Galois configuration of Grain-128a suggested by (Mansouri and Dubrova, 2013) that we recall in the following. For an easy distinguishing between the update functions from the original instantiation and the update functions of instantiation 1, i.e., after the change, we mark the latter with an upper index (^{g}). That is $(f_0,\ldots,f_{127})$ and $(q_0,\ldots,q_{127})$ denote the original update functions of the LFSR L and the NLFSR N, respectively, while $(f_0^g,\ldots,f_{127}^g)$ and $(q_0^g,\ldots,q_{127}^g)$ refer to the update functions after changing the FSRS configuration. Likewise we denote at clock t the state of the LFSR to be $L_t^g=(L_t^g[0],\cdots,L_t^g[127])$ and the state of the NLFSR as $N_t^g=(N_t^g[0],\cdots,N_t^g[127])$. The modified update functions of the FSRs of Grain-128a in Galois configuration are given in the Table 1.

All the other update functions that are not specified are the same as in the original specification of Grain-128a.

Of course this transformation is preserving, i.e., for all initial states of the FSRs in the original specification there exist corresponding initials states for instantiation 1 such that both instantiations produce the same outputs. However these initials states are not equal but need to be changed. A general treatment of this topic can be found in (Dubrova, 2010). For our configuration the initial state needs to be changed as follows.

$$\begin{aligned}
L_0^g[i] &= L_0[i], 0\le i\le 96\\
L_0^g[i] &= L_0[i]+f_{i-1}^g(L_0)+f_{i-2|_{+1}}^g(L_0)+\cdots+\\
&\quad f_{97|_{+i-98}}^g(L_0), 97\le i\le 127\\
N_0^g[j] &= N_0[j], 0\le j\le 96\\
N_0^g[j] &= N_0[j]+q_{j-1}^g(N_0)+q_{j-2|_{+1}}^g(N_0)+\cdots+\\
&\quad q_{97|_{+j-98}}^g(N_0), 97\le j\le 127
\end{aligned}$$

were $q_{i|_{+k}}$ and $f_{i|_{+k}}$ denote that every index in the arguments of the functions $q_{i|_{+k}}$ and $f_{i|_{+k}}$ is increased by k.

We implemented this variant of Grain-128a (instantiation 1) and measured the following improvements:

1. For the *time-optimized* solution the area size decreased from 1888 to 1816 GEs. The maximum throughput in the initialization mode increased to 1,17 GHz (by 15 %) and in the keystream generation mode increased to 1,51 GHz (by 22 %)
2. For the *area-optimized* solution the area size slightly decreased from 1640 to 1632 GEs. The maximum throughput in both modes increased by 7 %. In the initialization mode it became 0,61 GHz, and in the keystream generation mode it is 0,90 GHz.

Application of the Transformation. Even though the output function has only small differences compared to Grain-128, we could not use exactly the same transformation as was done in (Armknecht and Mikhalev, 2014). The update functions of both NLFSR and LFSR are different and therefore the intervals in the FSRs, where it is possible to move the monomials from the output fucntion, had to be chosen differently. In the following we provide the results of our transformation applied to Grain-128a. We use the upper index (^{T}) to indicate the FSR states and update functions after the transformation (instantiation 2). The exact transformations are provided in the Table 2. Again we only provide these functions where the transformation induced changes. Observe that the modified output function h^T is linear as opposed to the cubic output function h of original Grain-128a.

Also here it is necessary to modify initials states of instantiation 1 such that instantiation 2 produces the same output. The concrete mapping is given in Tab. 3. State entries that are not mentioned remained unchanged.

Below we summarize the results of the implementation of Grain-128a where the transformations have been applied (instantiations 2) and state the increase of throughput compared to instantiation 1 (i.e., Grain-128a where the FSRs have been changed into Galois configuration).

1. *Timing Optimization*. With these compiler-settings the area size slightly decreased from 1816 to 1736 GEs. The maximum throughput in the initialization mode increased to 1,3 GHz (by 11%) and in the keystream generation mode it was increased to 1,78 GHz (by 18 %)
2. *Area Optimization*. The area-size slightly increased (from 1632 GE to 1652 GE), when the maximum throughput increased by 20 % in the initialization mode and became 0,73 GHz; when in the keystream generation mode it increased by 18 % and became 1,06 GHz.

As in the case for Grain-128 (see (Armknecht and Mikhalev, 2014) for details), the transformation allowed an increase of the throughput. The improvements in both cases (Grain-128 and Grain-128a) are summarized in the Table 4

Table 1: The Update Functions of the FSRs after Transforming into Galois Configuration.

LFSR L^g:

$f^g_{127} = L^g_t[0] + L^g_t[7]$ $\quad$ $f^g_{123} = L^g_t[124] + L^g_t[34]$

$f^g_{119} = L^g_t[120] + L^g_t[62]$ $\quad$ $f^g_{115} = L^g_t[116] + L^g_t[85]$

$f^g_{111} = L^g_t[112] + L^g_t[80]$

$f^g_i = L^g_t[i+1], 0 \leq i \leq 127, \quad i \notin \{127, 123, 119, 115, 111\}$

NLFSR N^g:

$q^g_{127} = L^g_t[0] + N^g_t[0]$ $\quad$ $q^g_{126} = N^g_t[127] + N^g_t[39]N^g_t[47]$

$q^g_{125} = N^g_t[126] + N^g_t[59]N^g_t[63]$ $\quad$ $q^g_{124} = N^g_t[125] + N^g_t[0]N^g_t[64]$

$q^g_{123} = N^g_t[124] + N^g_t[52]$ $\quad$ $q^g_{116} = N^g_t[117] + N^g_t[0]N^g_t[2]$

$q^g_{110} = N^g_t[111] + N^g_t[0]N^g_t[1]$ $\quad$ $q^g_{105} = N^g_t[106] + N^g_t[0]N^g_t[2]N^g_t[3]$

$q^g_{102} = N^g_t[103] + N^g_t[71]$ $\quad$ $q^g_{101} = N^g_t[102] + N^g_t[0]$

$q^g_{100} = N^g_t[101] + N^g_t[0]N^g_t[32]$ $\quad$ $q^g_{99} = N^g_t[100] + N^g_t[63]$

$q^g_{98} = N^g_t[99] + N^g_t[59]N^g_t[63]N^g_t[64]N^g_t[66]$ $\quad$ $q^g_{97} = N^g_t[98] + N^g_t[38]N^g_t[54]$

$q^g_{96} = N^g_t[97] + N^g_t[39]N^g_t[47]N^g_t[51]$

$q^g_i = N^g_t[i+1] \;\; i \notin \{127, 126, 125, 124, 123, 116, 110, 105, 102, 101, 100, 99, 98, 97, 96\}$

Table 2: The Update and Output Functions after our Transformation.

$q^T_{89} = N^T_t[90] + N^T_t[3]$ $\quad$ $q^T_{87} = N^T_t[88] + N^T_t[1]$

$q^T_{73} = N^T_t[74] + N^T_t[13]L^T_t[9]$ $\quad$ $q^T_{71} = N^T_t[72] + N^T_t[11]L^T_t[7]$

$q^T_{46} = N^T_t[47] + L^T_t[14]L^T_t[21]$ $\quad$ $q^T_{45} = N^T_t[46] + L^T_t[13]L^T_t[20]$

$q^T_{36} = N^T_t[37] + N^T_t[96]L^T_t[43]$ $\quad$ $q^T_{34} = N^T_t[35] + N^T_t[94]L^T_{41}$

$q^T_{15} = N^T_t[16] + N^T_t[13]N^T_t[96]L^T_t[95]$ $\quad$ $q^T_{14} = N^T_t[15] + N^T_t[12]N^T_t[95]L^T_t[94]$

$f^T_{93} = L^T_t[94] + L^T_t[61]L^T_t[80]$ $\quad$ $f^T_{91} = L^T_t[92] + L^T_t[59]L^T_t[78]$

$h^T = N^T_t[15] + N^T_t[36] + N^T_t[45] + N^T_t[64] + N^T_t[73] + N^T_t[89] + L^T_t[93]$

Table 3: Mapping of the Initial States after our Transformation.

$N^T_0[89] = N^g_0[89] + N^g_0[2]$ $\quad$ $N^T_0[88] = N^g_0[88] + N^g_0[1]$

$N^T_0[73] = N^g_0[73] + N^g_0[12]L^g_0[8]$ $\quad$ $N^T_0[72] = N^g_0[72] + N^g_0[11]L^g_0[7]$

$N^T_0[45] = N^g_0[45] + L^g_0[13]L^g_0[20]$

$N^T_0[36] = N^g_0[36] + N^g_0[95]L^g_0[42]$ $\quad$ $N^T_0[35] = N^g_0[35] + N^g_0[94]L^g_0[41]$

$N^T_0[15] = N^g_0[15] + N^g_0[12]N^g_0[95]L^g_0[94]$

$L^T_0[93] = N^g_0[93] + L^g_0[60]L^g_0[79]$ $\quad$ $L^T_0[92] = N^g_0[92] + L^g_0[59]L^g_0[78]$

Table 4: The Perfomance Results for Grain-128 (Armknecht and Mikhalev, 2014) and Grain-128a.

Grain-128						
Configuration	Area-opimizing			Time-optimizing		
	Initialization		Keystream gen.	Initialization		Keystream gen.
	Area size*	Througput**	Througput	Area size	Througput	Througput
Original (Fibonacci)	1626	0,42	0,89	1853	1,03	1,29
Galois	1627	0,60 (+42%)	0,90	1794	1,11 (+8 %)	1,45 (+12 %)
Our Transformation	1656	0,73 (+20 %)	1,06 (+18 %)	1748	1,31 (+18 %)	1,8 (+24 %)

Grain-128a						
Configuration	Area-opimizing			Time-optimizing		
	Initialization		Keystream gen.	Initialization		Keystream gen.
	Area size	Througput	Througput	Area size	Througput	Througput
Original (Fibonacci)	1640	0,57	0,84	1888	1,02	1,23
Galois	1632	0,61 (+7%)	0,90 (+7%)	1816	1,17 (+15 %)	1,51 (+22 %)
Our Transformation	1652	0,73 (+20 %)	1,06 (+18 %)	1736	1,3 (+11 %)	1,78 (+18 %)

* Area size is given in gate equivalents (GE)

** Throuhput is given in gigahertz (GHz)

4.2 DECIM

Description. DECIM is a stream cipher proposal that passed to phase 3 of the e-Stream competition but was not selected for the final e-Stream portfolio. After the first version of DECIM (Berbain et al., 2005) was cryptanalyzed (Wu and Preneel, 2006), two improved versions were proposed: DECIM v2 and DECIM 128 (Berbain et al., 2008). The DECIM ciphers are composed of an LFSR of length n, a filtering Boolean function h, a mechanism called ABSG decimation (Gouget et al., 2005), and a buffer.

During the initialization phase for the certain number of clock-cycles the output of h is XORed with the update function of the LFSR and the cipher doesn't produce any keystream.

During the keystream generation phase the output of h is XORed with the bit with index 1 of the LFSR and the result is passed to the input of the ABSG mechanism.

The ABSG decimation mechanism provides a method for irregular decimation of bit sequences, and the buffer is used to ensure that at each clock-cycle there exist an output. However, for the considered transformation the only important parts are the LFSR and the filtering function h.

The filtering function of DECIM is given as:

$$h(\alpha_1,\cdots,\alpha_{13}) = \bigoplus_{1\le i<j\le 13} \alpha_i\alpha_j \oplus \bigoplus_{1\le i\le 13} \alpha_i \qquad (1)$$

where $\alpha_1 \cdots \alpha_{13}$ correspond to several LFSR bits.

The differences between DECIM v2 and DECIM-128 are minor. In DECIM v2 a LFSR of length 192 bits is used while in DECIM the length of the LFSR is 288 bits. The second difference is the choice of the bits associated to the inputs of the filtering function $(\alpha_1,\cdots\alpha_{13})$.

Applicability. We discuss now the applicability of the transformation to DECIM v2 and/or DECIM-128. Theorem 1 requires that the output of the filtering function h can be represented by

$$h(S_t) = S_t[\beta] + h_1(S_t) + h_2(S_t) = S_t[\beta] + g(S_{t-1}) + h_2(S_t) \qquad (2)$$

where S_t denotes the state of the LFSR at clock t, and the existence of an index α such that the interval $[\alpha,\cdots,\beta]$ is isolated with respect to h_2, and the interval $[\alpha+1,\cdots,\beta+1]$ is isolated with respect to g. However, due to the fact that any variable of h appears also in other non-linear terms, this is impossible. The reason is that for any choice of β and for any chosen splitting $h_1 \oplus h_2$, the variable corresponding to $S_t[\beta]$ appears at least either in h_1 and h_2. Hence, no interval can meet the isolation requirement.

Therefore, the transformation is not applicable for DECIM v2 and DECIM-128.

Please notice, that if during the initialization phase the value of the bit 1 was included into the computation of the update function of the LFSR, $S_t[1]$ could be chosen as $S_t[\beta]$ and the transformation would be possible.

4.3 Achterbahn-128/80

Description. Achterbahn-128/80 (Gammel et al., 2007) are two keystream generators with key sizes of 128 bits (Achterbahn-128) and 80 bits (for Achterbahn-80). They have been submitted to the e-Stream project and passed to phase 2. Both generators have a similar structure, consisting of several feedback shift registers whose outputs are combined by a non-linear output function. We refer to (Gammel et al., 2007) for a detailed description.

Applicability. The transformation described in Theorem 1 requires that for at least one of the FSRs there exist an interval $[\alpha,\ldots,\beta]$ such that the index β corresponds to one of the inputs of the output function. However, in Achterbanh-128/80 the output function takes its inputs from the last bits of each of the FSRs - bits with the index 0, meaning that β can only be equal to 0. Therefore, there exist no FSR bit with the index $\alpha < \beta$. Thus, the straightforward application of the transformation cannot be applied to Achterbahn-128/80.

Nevertheless, in order to solve this problem one can use the following trick :

1. Instead of using bits with index 0 in the output function, for each FSR i bit with index r_i is used, such that intervals $[0,\cdots,r_i]$ are isolated;
2. The initial states of each FSR i are modified. The new states are equal to the ones that each FSR i would take if it was clocked back r_i times.

If this modification is made the output of the cipher will not change.

Unfortunately, for Achterbahn-128/80 we face the similar problem as with DECIM (see Sec. 4.2). That is all of the FSR outputs are used as linear terms and also included in the non-linear terms of the filtering function, which makes it impossible to find an appropriate splitting. Hence, the conditions of Theorem 1 cannot be met and the transformation cannot be applied.

4.4 Hitag2

Description. The Hitag2 stream cipher (Courtois et al., 2009) is used for many practical applications

based on RFIDs. The most common use cases are building access control mechanisms and car immobilizers. Hitag2 consists of one 48-bit LFSR and a non-linear filtering function. The external block part is used during the initialization phase and it is not important with respect to the considered transformation.

Applicability. In principle the same problem as in the two previous examples takes place with Hitag2 and we could not apply the transformation.

REFERENCES

eSTREAM: the ECRYPT stream cipher project http://www.ecrypt.eu.org/stream/.

Agren, M., Hell, M., Johansson, T., and Meier, W. (2011). A new version of Grain-128 with authentication. In *Symmetric Key Encryption Workshop*.

Armknecht, F. and Mikhalev, V. (2014). On increasing the throughput of stream ciphers. In *Topics in Cryptology–CT-RSA*.

Babbage, S. and Dodd, M. (2008). The mickey stream ciphers. In (Robshaw and Billet, 2008), pages 191–209.

Berbain, C., Billet, O., Canteaut, A., Courtois, N., Debraize, B., Gilbert, H., Goubin, L., Gouget, A., Granboulan, L., Lauradoux, C., et al. (2005). Decim–a new stream cipher for hardware applications. *ECRYPT Stream Cipher Project Report 2005*, 4.

Berbain, C., Billet, O., Canteaut, A., Courtois, N., Debraize, B., Gilbert, H., Goubin, L., Gouget, A., Granboulan, L., Lauradoux, C., et al. (2008). Decim v2. In *New Stream Cipher Designs*, pages 140–151. Springer.

Cannière, C. D. and Preneel, B. (2008). Trivium. In (Robshaw and Billet, 2008), pages 244–266.

Courtois, N. T., ONeil, S., and Quisquater, J.-J. (2009). Practical algebraic attacks on the hitag2 stream cipher. In *Information Security*, pages 167–176. Springer.

Dinur, I. and Shamir, A. (2011). Breaking grain-128 with dynamic cube attacks. In *Fast Software Encryption*, pages 167–187. Springer.

Dubrova, E. (2010). Finding matching initial states for equivalent NLFSRs in the Fibonacci and the Galois configurations. *Information Theory, IEEE Transactions on*, 56(6):2961–2966.

Gammel, B., Göttfert, R., and Kniffler, O. (2007). Achterbahn-128/80: Design and analysis. In *ECRYPT Network of Excellence-SASC Workshop Record*, pages 152–165.

Good, T. and Benaissa, M. (2008). Hardware performance of eSTREAM phase-III stream cipher candidates. In *Proc. of Workshop on the State of the Art of Stream Ciphers (SACS08)*.

Gouget, A., Sibert, H., Berbain, C., Courtois, N., Debraize, B., and Mitchell, C. (2005). Analysis of the bit-search generator and sequence compression techniques. In *Fast Software Encryption*, pages 196–214. Springer.

Gupta, S. S., Chattopadhyay, A., Sinha, K., Maitra, S., and Sinha, B. P. (2013). High-performance hardware implementation for RC4 stream cipher. *IEEE Transactions on Computers*, 62(4):730–743.

Hell, M., Johansson, T., Maximov, A., and Meier, W. (2006). A stream cipher proposal: Grain-128. In *Information Theory, 2006 IEEE International Symposium on*, pages 1614–1618. IEEE.

Mansouri, S. S. and Dubrova, E. (2010). An improved hardware implementation of the Grain stream cipher. In *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, pages 433 –440.

Mansouri, S. S. and Dubrova, E. (2013). An improved hardware implementation of the Grain-128a stream cipher. In Kwon, T., Lee, M.-K., and Kwon, D., editors, *Information Security and Cryptology ICISC 2012*, volume 7839 of *Lecture Notes in Computer Science*, pages 278–292. Springer Berlin Heidelberg.

Nakano, Y., Fukushima, K., Kiyomoto, S., and Miyake, Y. (2011). Fast implementation of stream cipher K2 on FPGA. In *International Conference on Computer and Information Engineering (ICCIE)*, pages 117–123.

Robshaw, M. J. B. and Billet, O., editors (2008). *New Stream Cipher Designs - The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*. Springer.

Stefan, D. and Mitchell, C. (2008). On the parallelization of the MICKEY-128 2.0 stream cipher. *The State of the Art of Stream Ciphers, SASC*, pages 175–185.

Wu, H. and Preneel, B. (2006). Cryptanalysis of the stream cipher decim. In *Fast Software Encryption*, pages 30–40. Springer.

Yan, J. and Heys, H. M. (2007). Hardware implementation of the Salsa20 and Phelix stream ciphers. In *Electrical and Computer Engineering, 2007. CCECE 2007. Canadian Conference on*, pages 1125–1128. IEEE.

Z. Liu, L. Zhang, J. J. and Pan, W. (2010). Efficient pipelined stream cipher ZUC algorithm in FPGA. In *The First International Workshop on ZUC Algorithm, December 2-3, Beijing, China,*.