On Lightweight Stream Ciphers with Shorter Internal States

Frederik Armknecht, Vasily Mikhalev

University of Mannheim, Germany

Abstract. To be resistant against certain time-memory-data-tradeoff (TMDTO) attacks, a common rule of thumb says that the internal state size of a stream cipher should be at least twice the security parameter. As memory gates are usually the most area and power consuming components, this implies a sever limitation with respect to possible lightweight implementations.

In this work, we revisit this rule. We argue that a simple shift in the established design paradigm, namely to involve the fixed secret key not only in the initialization process but in the keystream generation phase as well, enables stream ciphers with smaller area size for two reasons. First, it improves the resistance against the mentioned TMDTO attacks which allows to choose smaller state sizes. Second, one can make use of the fact that storing a *fixed* value (here: the key) requires less area size than realizing a register of the same length. We demonstrate the feasibility of this approach by describing and implementing a concrete stream cipher Sprout which uses significantly less area than comparable existing lightweight stream ciphers.

Keywords: Stream Ciphers, Lightweight Cryptography, Time-Memory-Data-Tradeoff Attacks

1 Introduction

There is a strong and growing need for cryptographic primitives that can be implemented on the devices which have very limited resources such as the area size on the chip, memory, and power consumption. During the last years several lightweight block ciphers, e.g., see [14,15], and stream ciphers [16,6,26,25,1,24] have been proposed. Stream ciphers usually allow for a higher throughput but require a larger area size compared to block ciphers. The latter is mainly caused by time-memory-data trade-off (TMDTO) attacks which aim to recover the internal state of the stream cipher [22,5,12]. The attack effort is in $O(2^{\sigma/2})$, where σ denotes the size of the internal state of a stream cipher. This results into a rule of thump that for achieving κ-bit security level, the size of internal state should be at least $\sigma = 2 \cdot \kappa$. It means that in order to implement such a cipher at least $2 \cdot \kappa$ memory gates are required which is usually the most area and power-consuming resource.

Table 1. Area Size of the eStream Finalists, Lightweight block ciphers and Sprout

Cipher	Area size (GE)	Throughput (Kb/s)*	Logic process	Source
Block ciphers				
PRESENT 80 [14]	1570	200	$0.18 \mu m$	[14]
PRESENT 80 [14]	1000	11.4	$0.35 \mu m$	[34]
KATAN32 [15]	802	12.5	$0.13 \mu m$	[15]
KATAN48 [15]	927	18.8	$0.13 \mu m$	[15]
KATAN64 [15]	1054	25.1	$0.13 \mu m$	[15]
KTANTAN32 [15]	462	12.5	$0.13 \mu m$	[15]
KTANTAN48 [15]	588	18.8	$0.13 \mu m$	[15]
KTANTAN64 [15]	688	25.1	$0.13 \mu m$	[15]
Stream ciphers				
Mickey [7]	3188	100	$0.13 \mu m$	[23]
Trivium [16]	2580	100	$0.13 \mu m$	[23]
Grain 80 [26]	1294	100	$0.13 \mu m$	[23]
Grain 80 [26]	1162	100	$0.18 \mu m$	This work
Sprout	813	100	$0.18 \mu m$	This work

*- The throughput is given for the clock frequency of 100KHz

Our Contribution In this work, we investigate an extension in the common design for stream ciphers which allows to realize secure lightweight stream cipher with an area size beyond the trade-off attack bound mentioned above. The core idea is to split the set of internal states into 2^{κ} equivalence classes such that a TMDTO attack has to consider each of these classes at least once. To achieve this goal, we suggest to involve the key into the update process of the internal state.

Theoretically, the overall approach is still to have a sufficiently large internal state which determines the keystream bits. The main difference though is that part of this state is the secret key itself and not only a state that has been derived from this key. If one considers the case that the key is fixed for the device, one can make use of the fact that storing a fixed key is significantly less area consuming than deploying a register of the same length. In fact, a similar idea has been used in the design of KATAN/KTANTAN [15]. Moreover, the approach may allow for designs where the overall state size is smaller than 2κ.

We demonstrate the feasability of this approach by describing and implementing a concrete stream cipher named Sprout. It builds upon the Grain 128a [1] cipher but uses shorter registers and aims for 80 bit security. We argue that Sprout seems to inherit the strengths of Grain 128a. However, our implementation confirms that Sprout uses significantly less area size than the eStream finalisits of the hardware portfolio and also compares favorably with many lightweight block ciphers (see Table 1 for an overview).

Outline In section 2, we describe the used model for stream ciphers and recall time-memory-data trade-off attacks. In section 3, we explain our general design approach for strengthening stream ciphers against TMDTO attacks. In section 4, we propose a concrete construction following our design approach. Section 5 addresses the security of the proposal. Section 6 concludes the paper.

2 Preliminaries

2.1 Time-Memory-Data-Trade-Off Attacks

Cryptanalysis often boils down to the following question. Given a function $F : \mathcal{N} \to \mathcal{N}$ and a value y within the image of F, find a preimage of y, i.e., determine a value $x \in \mathcal{N}$ such that $F(x) = y$. To accomplish this goal, two extreme cases are considerable. One approach would be to use brute force search, i.e., randomly pick values $x \in \mathcal{N}$ until $F(x) = y$ does hold. This process would be repeated whenever the attacker aims to invert F. The other extreme approach would be to precompute all possible values beforehand and store them in a large table, i.e., to trade recurring computation effort by memory. This would result into the situation that every subsequent attack is essentially a simple look-up.

In 1980 Hellman [27] suggested a time-memory-trade-off (TMTO) attack which is probabilistic and falls somewhere in between a brute force attack and a precomputation attack. This initiated a long line of research on different trade-off attacks. A typical trade-off attack consists of two phases: the first is the precomputation phase, often called the offline phase, while the second is referred to as the real-time, or on-line phase. In the offline phase, the attacker precomputes a large table (or sets of tables) using the function F he is trying to invert, while in the online phase the attacker captures a function output and checks if this value is located in her tables. If this attack is successful the attacker can learn the value x for which $y = F(x)$. Usually, this type of attacks is evaluated by looking at the following costs:

– $|\mathcal{N}|$ - the size of the search space $\mathcal{N}$
– T_P - the time effort of the precomputation phase
– T - the time effort of the online phase
– M - memory cost of the attack.
– D - number of usable data samples, i.e., outputs of F, during the online phase.

Trade-off attacks usually differ in the relation between these values (often expressed by a trade-off curve) and conditions that need to be met. A further distinctive feature is the concrete attack scenario. Here we are interested into two specific scenarios that we term scenario A and B, respectively, and that we explain below.

In scenario A, an attacker is given *one* image $y \in \mathcal{N}$ and tries to find a preimage under F, that is a value $x \in \mathcal{N}$ such that $F(x) = y$. This scenario-A-attacks represent the most general class of attacks. In table 2, we list the

Table 2. Overview of Trade-Off Attacks for Scenario A

Work	Trade-off curve	Restrictions	Precomputation time
Hellman [27]	$\|\mathcal{N}\|^2 = TM^2$	$1 \leq T \leq \|\mathcal{N}\|$	$T_P = \|\mathcal{N}\|$
Oeschslin et al. [33]	$\|\mathcal{N}\|^2 = 2TM^2$	$1 \leq T \leq \|\mathcal{N}\|$	$T_P = \|\mathcal{N}\|$
BG [5], [22]	$\|\mathcal{N}\| = M$	$T = 1$	$T_P = \|\mathcal{N}\|$
BS [12]	$\|\mathcal{N}\|^2 = TM^2$	$1 \leq T \leq \|\mathcal{N}\|$	$T_P = \|\mathcal{N}\|$
BSW [13]	$\|\mathcal{N}\|^2 = TM^2$	$1 \leq T \leq \|\mathcal{N}\|$	$T_P = \|\mathcal{N}\|$
Barkan et al. [10]	$\|\mathcal{N}\|^2 + \|\mathcal{N}\|M = 2TM^2$	$1 \leq T \leq \mathcal{N}$	$T_P = \|\mathcal{N}\|$
Dunkelman [20]	$\|\mathcal{N}\|^2 = TM^2$	$1 \leq T \leq \|\mathcal{N}\|$	$T_P = \|\mathcal{N}\|$

Table 3. Overview of Trade-Off Attacks for Scenario B

Work	Trade-off curve	Restrictions	Precomputation time
BG [5], [22]	$\|\mathcal{N}\| = TM$	$1 \leq T \leq D$	$T_P = M$
BS [12]	$\|\mathcal{N}\|^2 = TM^2D^2$	$D^2 \leq T \leq \|\mathcal{N}\|$	$T_P = \|\mathcal{N}\|/D$
BSW [13]	$\|\mathcal{N}\|^2 = TM^2D^2$	$(DR)^2 \leq T$	$T_P = \|\mathcal{N}\|/D$
Barkan et al. [10]	$\|\mathcal{N}\|^2 + \|\mathcal{N}\|D^2M = 2TM^2$	$D^2 \leq T \leq \|\mathcal{N}\|$	$T_P = \|\mathcal{N}\|/D$

effort of existing trade-off attacks in scenario A. As one can see, all attacks have in scenario A a precomputation effort which is equivalent to searching the complete search space $\mathcal{N}$. In short, the reason is that a trade-off attack can only be successful if the given image y has been considered during the precomputation phase.

This can be relaxed in scenario B. Here, an attacker is given D images $y_1, \ldots, y_D$ of F and the goal is to find a preimage for any of these points, i.e., a value $x_i \in \mathcal{N}$ such that $F(x_i) = y_i$. The main difference is that for a successful attack, it isn't any longer necessary to cover the whole search space $\mathcal{N}$ during the precomputation phase. Instead it is sufficient that at least one of the outputs y_i has been considered. An overview of time-memory-data-trade-off attacks for scenario B is given in table 3. Note that the parameter R mentioned in the BSW attack stands for the sampling resistance of a stream cipher. In a nutshell, it is connected to the number of special states that can be efficiently enumerated. For example, R can be defined as $2^{-\ell}$ where ℓ is the maximum value for which the direct enumeration of all the special states which generate ℓ zero bits is possible. As the sampling resistance strongly depends on the concrete design, we will not consider it in our general analysis of trade-off attacks.

2.2 Keystream Generators

Description. Stream ciphers are encryption schemes that are dedicatedly designed to efficiently encrypt data streams of arbitrary length. The most common approach for realizing a stream cipher is to design a *keystream generator* (KSG). In a nutshell, a KSG is a finite state machine using an internal state, an update function, and an output function. At the beginning, the internal state is initialized based on a secret key and, optionally, an initial value (IV). Given this, the KSG regularly outputs keystream bits that are computed from the current internal state and updates the internal state. The majority of existing KSGs are covered by the following definition:[1]

Definition 1 (Keystream Generator). *A keystream generator (KSG) comprises three sets, namely*

- *the key space* $\mathcal{K} = \mathrm{GF}(2)^\kappa$,
- *the IV space* $\mathcal{IV} = \mathrm{GF}(2)^\nu$,
- *the state space* $\mathcal{S} = \mathrm{GF}(2)^\sigma$,

and the following three functions

- *an initialization function* $\mathsf{Init} : \mathcal{IV} \times \mathcal{K} \to \mathcal{S}$
- *a bijective*[2] *update function* $\mathsf{Upd} : \mathcal{S} \to \mathcal{S}$
- *an output function* $\mathsf{Out} : \mathcal{S} \to \mathrm{GF}(2)$

A KSG operates in two phases. In the initialization phase*, the KSG takes as input a secret key k and an IV iv and sets the internal state to an initial state* $st_0 := \mathsf{Init}(iv, k) \in \mathcal{S}$. *Afterwards, the keystream generation phase executes the following operations repeatedly (for $t \geq 0$):*

1. *Output the next keystream bit* $z_t = \mathsf{Out}(st_t)$
2. *Update the internal state* st_t *to* $st_{t+1} := \mathsf{Upd}(st_t)$

In this work, we consider attackers who are given several (possibly many) keystream bits and who aim for computing the remaining keystream. To this end, we assume that the attacker has full control over the IV. This means that the IV is not only known by the attacker but in fact she can choose it. Obviously tradeoff attacks represent a possible threat in this context. We shortly address in the following tradeoff attacks against keystream generators before we discuss our proposed design in the next section.

Trade-Off Attacks Against Keystream Generators.

Recovering the Key. In principle, two different approaches can be considered for applying a trade-off attack, depending on what function the attacker aims to invert. The most obvious approach is to invert the whole cipher. That is one considers the process which takes as input a secret key $k \in \mathcal{K} = \mathrm{GF}(2)^\kappa$ and

[1] As far as we know the only exception is the A2U2 stream cipher [18], which appears to be insecure (see i.e. [2]).

[2] In fact, our discussions can be easily extended to the case of non-invertible update functions. However, assuming reversibility simplifies the explanations and is given for most designs anyhow.

outputs the first κ keystream bits as a function $F_{\mathsf{KSG}} : \mathrm{GF}(2)^\kappa \to \mathrm{GF}(2)^\kappa$. The search space would be $\mathcal{N} = \mathcal{K} = \mathrm{GF}(2)^\kappa$ in this case. As already explained, trade-off attacks for scenario A would require a precomputation time which is equivalent to exhaustive search in the key space. If we say that a security level of κ expresses the requirement that a successful attack requires at least once a time effort in $\mathcal{O}(2^\kappa)$, then such attacks do not represent a specific threat.

Observe that although an attacker may have knowledge of significantly more than κ bits, scenario B trade-off attacks are not applicable here (at least not in general). To see why, let $F^t_{\mathsf{KSG}} : \mathrm{GF}(2)^\kappa \to \mathrm{GF}(2)^\kappa$ be the function that takes as input the secret key and outputs the keystream bits for clocks $t, \ldots, t + \kappa - 1$. That is it holds that $F^0_{\mathsf{KSG}} = F_{\mathsf{KSG}}$ from above. Then, the knowledge of $D + \kappa - 1$ keystream bits translates to knowing images of $F^0_{\mathsf{KSG}}, \ldots, F^{D-1}_{\mathsf{KSG}}$ and in fact, inverting one of these would be sufficient. However, these functions are all different. In particular, any precomputation done for one of these, e.g., F^i_{KSG}, cannot be used for inverting another one, e.g, F^j_{KSG} with $i \neq j$.

Recovering the Internal State. An alternative approach is to invert the output function Out only, that is used in the keystream generation phase. More precisely, let $F_{\mathsf{Out}} : \mathrm{GF}(2)^\sigma \to \mathrm{GF}(2)^\sigma$ be the function that takes the internal state $st_t \in \mathrm{GF}(2)^\sigma$ at some clock t as input and outputs the σ keystream bits $z_t, \ldots, z_{t+\sigma-1}$. The search space would be $\mathcal{N} = \mathcal{S}$. A scenario-A trade-off attack would again require a precomputation time equal to $|\mathcal{N}| = |\mathcal{S}|$ which implies that $\sigma \geq \kappa$ if one aims for a security level of κ.

We come now to the essential part. As each keystream segment $z_t, \ldots, z_{t+\sigma-1}$ is an output of the same function F_{Out} and as the knowledge of one internal state st_t allows to compute all succeeding keystreams bits z_r for $r \geq t$ (and as Upd is assumed to be reversible, the preceeding keystream bits as well), scenario B attacks are suitable. As can be seen from table 3, each attack would require at least once a time effort of about $\sqrt{|\mathcal{S}|} = 2^{\sigma/2}$. This implies the already mentioned rule of selecting $\sigma \geq 2\kappa$.

3 Our Basic Approach

3.1 Motivation

In this section, we discuss a conceptually simple adaptation of how keystream generators are commonly designed (see definition 1). The goal is to make stream ciphers more resistant against TMDTO attacks such that shorter internal states can be used. To this end, let us take another look at trade-off attacks. An attacker who is given a part of the keystream aims to find an internal state which allows to compute the remaining keystream. Let $F^{\mathrm{compl.}}_{\mathsf{Out}}$ denote the function that takes as input the initial state and outputs the complete keystream. Here, "complete" refers to the maximum number of keystream bits that are intended by the designer. If no bound is given, then we simply assume that 2^σ keystream bits are produced as this refers to the maximum possible period. From an attacker's point of view, any internal state that allows for reconstructing the keystream is equally good. This brings us to the notion of keystream-equivalent states:

Definition 2 (Keystream-equivalent States). *Consider a KSG with a function* $F_{\mathsf{Out}}^{\text{compl.}}$ *that outputs the complete keystream. Two states* $st, st' \in \mathcal{S}$ *are said to be* keystream-equivalent *(in short* $st \equiv_{\text{kse}} st'$*) if there exists an integer* $r \geq 0$ *such that* $F_{\mathsf{Out}}^{\text{compl.}}(\mathsf{Upd}^r(st)) = F_{\mathsf{Out}}^{\text{compl.}}(st')$*. Here,* Upd^r *means the* r*-times application of* Upd.

Observe that keystream-equivalence is an equivalence relation.[3] For any state $st \in \mathcal{S}$, we denote by $[st]$ its equivalence class, that is

$$[st] = \{st' \in \mathcal{S} | st \equiv_{\text{kse}} st'\} \tag{1}$$

To see why this notion is important for analyzing the effectiveness of a TMDTO attack, let us consider an arbitrary KSG with state space $\mathcal{S}$. As any state is member of exactly one equivalence class, the state space can be divided into ℓ distinct equivalence classes:

$$\mathcal{S} = \left[st^{(1)}\right] \dot\cup \ldots \dot\cup \left[st^{(\ell)}\right] \tag{2}$$

Now assume a TMDTO attacker who is given some keystream (z_t), based on an unknown initial state st_0. Recall that the strategy of a trade-off attack is not to exploit any weaknesses in the concrete design but to efficiently cover a sufficiently large fraction of the search space. In this case if none of the precomputations were done for values in $[st_0]$, the attack cannot be successful unless the online phase searches all equivalence classes that have been ignored during the precomputation phase. This leads to the following observation: a TMDTO attack on the KSG will be a union of TMDTO attacks, one for each equivalence class. That is we have ℓ TMDTO attacks with search spaces $\mathcal{N}_i = \left[st^{(i)}\right]$, $i = 1, \ldots, \ell$, respectively. As each of these attacks has a time effort of at least 1, we get a lower bound of ℓ for the attack effort. Now, if one designs a cipher such that $\ell \geq 2^\kappa$, then one has achieved the required security level against trade-off attacks. This is exactly the idea behind the design approach discussed next.

3.2 The Design Approach

We are now ready to discuss our proposed design. The basic idea is to achieve a splitting of the internal state space in sufficiently many equivalence classes. To achieve this, we divide the internal state into two parts: a variable part that may change over time and a fixed part. For practical reasons the fixed part will be realized by simply re-using the secret key (more on this later). The main difference to a KSG as given in definition 1, the update function Upd will compute the next variable state from the current variable state *and* the fixed

[3] This is due to the fact that for any state $st \in \mathcal{S}$, the sequence $(\mathsf{Upd}^r(st))_{r\geq 0}$ is cyclic and that Upd is reversible by assumption.

secret key. We call such a construction a KSG with keyed update function, to be defined below. Observe that this definition is in fact covered by definition given in [31].

Definition 3 (Keystream Generator With Keyed Update Function). *A keystream generator (KSG) with* keyed update function *comprises three sets, namely*

- *the key space $\mathcal{K} = \mathrm{GF}(2)^\kappa$,*
- *the IV space $\mathcal{IV} = \mathrm{GF}(2)^\nu$,*
- *the variable state space $\mathcal{S} = \mathrm{GF}(2)^\sigma$,*

and the following three functions

- *an initialization function $\mathsf{Init} : \mathcal{IV} \times \mathcal{K} \to \mathcal{S}$*
- *an update function $\mathsf{Upd} : \mathcal{K} \times \mathcal{S} \to \mathcal{S}$ such that $\mathsf{Upd}_k : \mathcal{S} \to \mathcal{S}$, $\mathsf{Upd}_k(st) := \mathsf{Upd}(k, st)$, is bijective for any $k \in \mathcal{K}$, and*
- *an output function $\mathsf{Out} : \mathcal{S} \to \mathrm{GF}(2)$.*

The internal state ST is composed of a variable part $st \in \mathcal{S}$ and a fixed part $k \in \mathcal{K}$. Initialization and keystream generation work analogously to definition 1 with the only difference that the state update also depends on the fixed secret key.

Let us take a look at the minimum time effort for a TMDTO attack against a KSG with keyed update function. We make in the following the assumption that any two different states $ST = (st, k)$ and $ST' = (st', k')$ with $k \neq k'$ never produce the same keystream, that is $F_{\mathsf{Out}}^{\text{compl.}}(ST) \not\equiv_{\text{kse}} F_{\mathsf{Out}}^{\text{compl.}}(ST')$. Hence, we have at least 2^κ different equivalence classes. As the effort grows linearly with the number of equivalence classes, we assume in favor of the attacker that we have exactly 2^κ equivalence classes. This gives a minimum time effort of 2^κ.

Observe that similar techniques are present in stream cipher modes for block ciphers like OFM or CTR. However, as far as we know it has never been discussed for directly designing stream ciphers with increased resistance against TMDTO-attacks. In this context, we think that this approach has two interesting consequences with respect to saving area size in stream cipher implementations:

1. Apparently one can achieve a security level of κ independent of length σ of the variable state. This allows to use a shorter internal state which directly translates to saving area size.[4]
2. For technical reasons, storing a fixed value (here: the key) can be realized with significantly less area size than is necessary for storing a variable value. This effect has been used for example in the construction of the block cipher KTANTAN ([15]). It allows for further savings compared to KSGs with an register of length $\geq 2\kappa$.

We use these in the following section for proposing a concrete cipher named Sprout. Our implementations showed that Sprout needs significantly less area size than existing ciphers with comparable security level.

[4] Of course, σ shouldn't be too small. Otherwise, the period of the KSG may become too short and the cipher may also become vulnerable for other attacks like guess-and-determine.

4 The Stream Cipher Sprout

Within this section, we describe and discuss a concrete keystream generator which follows the design strategy presented in the previous section. We start with an overview of the overall structure in section 4.1, give the full specification in section 4.2, explain the design rationale in 4.3, and present the implementations results in 4.4. The security of the scheme will be discussed in the next section, i.e., in section 5.

4.1 Overall Structure.

The design of Sprout is an adaptation of the basic design used for the Grain family of stream ciphers [25,1,26,24]. More precisely, we used Grain 128a as the starting point as this is the newest member of the Grain family which overcomes some weaknesses found for previous version. Each Grain cipher is composed of a linear feedback shift register (LFSR), a non-linear feedback shift register (NLFSR), and an output function. The LFSR and NLFSR states represent the internal state which at the beginning are initialized with the key and the IV. The output of the LFSR is fed into the NLFSR to ensure a minimum period with respect to the internal states while the purpose of the NLFSR is to make certain standard attacks like algebraic attacks infeasible. Moreover, several bits are taken from both FSRs as input to the output function. During the initialization phase, the outputs of the output function are fed back into the FSRs, while in the keystream generation phase, they represent the keystream bits.

For Sprout, we adopted this design. That is Sprout likewise uses an LFSR, an NLFSR, and an output function, and these components are connected in a similar way (see figure 1). However, several changes have been taken as well. The main differences are:

Round Key Bits: To involve the secret key into the update function, a round key function has been added. In a nutshell, in each clock it cyclically selects the next key bit and adds it to the state of the NLFSR if the sum of certain LFSR and NLFSR bits is equal to 1. More precisely, it is added to the LFSR output which in turn goes into the NLFSR.

Counter: Like in Grain (or any other stream cipher), we use a counter to determine the number of rounds during the initialization phase. Part of the counter is re-used in Sprout for selecting the current round key bit. In addition, we use one of the counter bits also in the update process. The reason is to avoid situations where shifted keys result into shifted keystreams (and hence violating our assumption that two states with different keys are not keystream-equivalent).

Register Lengths: The sizes of the FSRs have been reduced to 40 bits each. This sums up to 80 bits which is equal to the key length.

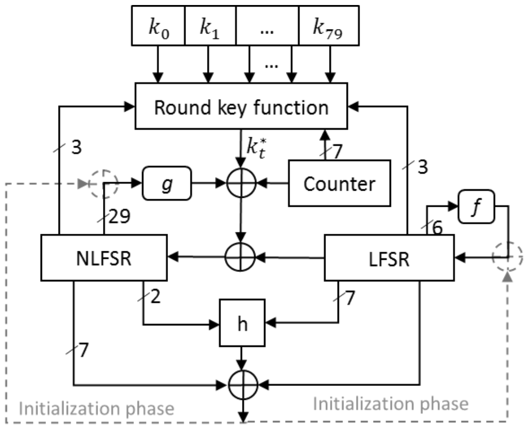

Fig. 1. The Structure of Sprout.

4.2 Specifications.

We give now the full specification of Sprout. We use the following notation:

- t - the clock-cycle number;
- $L_t = (l_t, l_{t+1}, \cdots, l_{t+39})$ - state of the LFSR during the clock-cycle t
- $N_t = (n_t, n_{t+1}, \cdots, n_{t+39})$ - state of the NLFSR during the clock-cycle t
- $C_t = (c_t^0, c_t^1, \cdots, c_t^8)$ - state of the counter during the clock-cycle t
- $k = (k_0, k_1, \cdots, k_{79})$ - key.
- $iv = (iv_0, iv_1, \cdots, iv_{69})$ - initialization vector.
- k_t^* .- the round key bit generated during the clock-cycle t
- z_t - the keystream bit generated during the clock-cycle t.

Feedback shift registers. Both the LFSR and the NLFSR are 40-bits long. The LFSR uses the following primitive feedback polynomial which guarantees a period of $2^{40} - 1$:

$$P(x) = x^{40} + x^{35} + x^{25} + x^{20} + x^{15} + x^6 + 1 \tag{3}$$

We denote the corresponding feedback function by f, that is $l_{t+40} = f(L_t)$.

The NLFSR feedback function has almost the same form as the NLFSR update function of Grain 128a but different indexes are used due to the fact that the NLFSR is shorter. This function is XORed with the output of the LFSR, with the round key bit k_t^* and with the counter bit c_t^4. The full specification is

$$\begin{aligned} n_{t+40} &= g(N_t) + k_t^* + l_t + c_t^4 \\ &= k_t^* + l_t + c_t^4 + n_t + n_{t+13} + n_{t+19} + n_{t+35} + n_{t+39} \\ &+ n_{t+2}n_{t+25} + n_{t+3}n_{t+5} + n_{t+7}n_{t+8} + n_{t+14}n_{t+21} + n_{t+16}n_{t+18} \\ &+ n_{t+22}n_{t+24} + n_{t+26}n_{t+32} + n_{t+33}n_{t+36}n_{t+37}n_{t+38} \\ &+ n_{t+10}n_{t+11}n_{t+12} + n_{t+27}n_{t+30}n_{t+31} \end{aligned} \tag{4}$$

Counter. The 9-bit counter is composed of 2 parts. The first seven bits ($c_t^0 \cdots c_t^6$) (with c_t^0) indicating the LSB) are used to compute the index of the key bit which is selected during the current round. Hence, these bits count from 0 to 79 before being set to zero again. The remaining two bits are only used within the initialization phase to count until $4 \times 80 = 320$.
Round key function. The round key function is responsible for making the update function key dependent. At each clock t, the round key function computes one round key bit k_t^* as follows:

$$\begin{aligned} &k_t^* = k_t,\ \ 0 \leq t \leq 79; \\ &k_t^* = (k_{t\ mod\ 80}) \cdot (l_{t+4} + l_{t+21} + l_{t+37} + n_{t+9} + n_{t+20} + n_{t+29}),\ \ t \geq 80; \end{aligned} \tag{5}$$

That is in the first 80 clocks, each key bit is involved exactly once in the update function. Afterwards, the round key function cyclically selects the next key bit and adds it if $l_{t+4} + l_{t+21} + l_{t+37} + n_{t+9} + n_{t+20} + n_{t+29} = 1$. Otherwise the key bit is skipped in this round.
Output function. The output of the cipher is a nonlinear function which takes several LFSR and NLFSR bits as its input. The nonlinear part of the output function has the form $h(x) = x_0x_1+x_2x_3+x_4x_5+x_6x_7+x_0x_4x_8$ where $x_0, \cdots, x_8$ corresponds to the state variables: n_{t+4}, l_{t+6}, l_{t+8}, l_{t+10}, l_{t+32}, l_{t+17}, l_{t+19}, l_{t+23}, n_{t+38}, respectively. The keystream bit of the cipher is computed as

$$z_t = h(x) + l_{t+30} + \sum_{j \in B} n_{t+j} \tag{6}$$

where $B = \{1, 6, 15, 17, 23, 28, 34\}$.
Initialization phase. In the initialization phase the 40 NLFSR stages are loaded with the first 40 IV bits, i.e., $n_i = iv_i$ for $i = 0 \leq i \leq 39$, and the first 30 LFSR stages are loaded with the remaining IV bits, i.e., $l_{i-40} = iv_i$ for $40 \leq i \leq 69$. To avoid the all-zero state, the last 10 bits of the LFSR are filled with constant values '0' and '1' as follows: $l_{30} = \ldots = l_{38} = 1$ and $l_{39} = 0$. Then, the cipher is clocked 320 times without producing any keystream. Instead the output function is fed back and XORed with the input, both to the LFSR and to the NFSR, i.e., $l_{t+40} = z_t + f(L)$ and $n_{t+40} = z_t + k_t^* + l_t + c_t^4 + g(N_t)$.

Keystream generation phase. After 320 clock-cycles the initialization phase is over and the cipher starts operating in accordance with equations (3, 4, 5, 6) generating the keystream.

4.3 Design rationale

Choice of General Design. As already mentioned, our design adopts the generic idea behind the Grain family. This has been done for several reasons. First of all, our primary goal was to show the feasibility of the approach discussed in section 3. Therefore, we decided against designing a new cipher from scratch (which may have eventually turned out to be vulnerable against other attacks) but rather to build upon an existing established design. To this end, our focus was to pick a stream cipher that is already lightweight, is scalable (at least to some extent), and has undergone already some cryptanalysis.

State Size. Our goal was to show that is possible to develop a secure stream cipher that uses a register of size σ significantly below 2κ. A further goal was however to keep the process of including the key into the update procedure rather simple. While more involved mechanisms are possible in principle, this would come at the cost of an increased area size. However, using a simple key inclusion procedure could make a cipher subject to guess-and-determine attacks, i.e., attacks where the adversary guesses the internal state and tries to derive the key from the keystreams. Therefore, we decided for a conservative choice of $\sigma = \kappa$. This implies that guessing the complete register has the same effort as guessing the key. We leave it as open problem to come up with designs that use a significantly smaller register.

LFSR Update Function. The main reason for involving an LFSR in the Grain family is to ensure a minimum period. Consequently, the feedback polynomial of the LFSR used in Sprout is primitive to guarantee a maximum period of $2^{40} - 1$. A further design criteria was to choose a polynomial with not too few terms in order to increase the resistance of the cipher against correlation attacks.

NLFSR Update Function. The update function of the NLFSR $g(N)$ is XORed with the LFSR output l_t, the round key bit k_t^*, and the counter bit c_t^4. Each of these parts has different purpose and we will discuss them separately.

$g(N)$ is the nonlinear function which has the same form as in Grain 128a, where it was carefully selected in order to resist against different types of attacks [1]. As the used NLFSR is shorter than the one of Grain 128a, different indexes had to be chosen. Nonetheless, the relevant cryptographic properties remained: It is balanced, has a nonlinearity of 267403264, a resiliency of 4, and the set of the best linear approximations is of size 2^{14}. Each of the functions from this set has a bias of $63 \cdot 2^{-15}$ [1]. In fact, because of the involvement of the round key bits we suspect that not all of these properties are still required in Sprout. For example the attacks based on the linear approximations should not work anymore (see section 5). Nonetheless, this function hasn't revealed any unexpected weaknesses over the time why we decided to stick to it.

The LFSR output is XORed with the NLFSR update function the same way as it is done in Grain family so that each of the NLFSR state bits is balanced.

The main goal of using the round key bit is to improve the resistance against TMDTO attacks as explained in section 3.2.

As explained, we aimed for a key involvement procedure that is as simple as possible to save area size. The probably most simple one is to select the key bits cyclically what we consider here (see also the discussion below). However, this would result into the situation that two keys where one is only a shifted version of the other also produce the same keystream (but only shifted). This would clearly violate the basic requirement, namely that two states with different keys are not keystream-equivalent. To avoid this situation, the counter bit c_t^4 is included as well. By doing so, even if one key is just a shifted copy of the other, due to the different counter bits it should not result into the same keystream.

Output Function. The output function has the same form as the one used in Grain 128a. This function has nonlinearity of 61440. The best linear approximation of the nonlinear part h has a bias of 2^5, and there are 2^8 such linear approximations [1].

Round Key Function. The design criteria for the round key function were that over the time, each key bit has been involved into the update function, and that the mechanism is lightweight, i.e. does not consume a lot of area and power. A straightforward approach would be to involve all key bits simultaneously. However, this would require a prohibitively large number of logic gates. Therefore, we decided to involve the key bit by bit, that is at each clock exactly one key bit is involved. In order to make sure that the initial state of both registers depends on all of the key bits after reasonable number of clocks, at first we XOR each of the key bits with the NLFSR update function with the first 80 clocks. Only afterwards the more involved round key function is used.

To avoid situations where some key bits are not (or rarely) selected, we decided that the function goes cyclically through the key bits and always chooses the next key bit. However, to thwart possible guess-then-determine attacks, the key involvement needs to be coupled with the register states somehow (in a preferably simple way). As mentioned above the states of both registers are almost balanced. The idea is that at each clock, a number of register bits are taken and XORed. The currently selected key bit is inserted if and only if the sum of the register bits is equal to 1. In other words, the register bits do not influence which key bit is selected but decide if it is inserted. The advantage is that this requires only a counter and multiplexers. As a counter is required anyway for the initialization phase, one can further save area by simply reusing the counter (what we do). In the concrete realization, we use three bits from the LFSR and NLFSR, each. The three NLFSR bits have been selected in such a way that none of these bits is involved in any other function used in this cipher.

Initialization Phase. The initilaization phase is in principle the same as for Grain 128a. For example, fixing some of the LFSR bits is done to avoid that an attacker can simply set the internal state to the all zero state by choosing an appropriate IV. The initialization phase runs 320 clock cycles, which is the same number as in Grain 128a (where 256 clock-cycles are used in the initialization phase and 64 clock-cycles are required for the authentication process). Observe that we use smaller registers though, hence probably even increasing the level of diffusion. That is the number of clock cycles used for the initialization phase is 4 times the state length. This is a stronger ratio than for Grain 80 and Grain 128 where the number of clock cycles is equivalent to the state size.

Naming. The name Sprout has been inspired by two facts. First, it builds upon the Grain cipher which already suggested to choose a plant-related name. Second, the main difference is that we "plant" during the update process key round bits into the middle of the cipher. In some sense, this can be seen as a kind of additional key-related seed that (hopefully) quickly sprouts and expands over the whole state.

4.4 Implementation Results

In order to demonstrate the feasibility of our design, we implemented Sprout and compared the area size with the eStream finalists of portfolio 2 (hardware oriented stream ciphers), being Grain 80 [26], Mickey 2.0 [6], and Trivium [16]. All of them use 80-bits keys.

For the implementation, we used the Cadence RTL Compiler [5] and the technology library UMCL18G212T3 (UMC 0.18μm process) for synthesis and simulation and aimed for an implementation operating on the clock-frequency of 100 KHz. We stress that in our implementation, we do not consider any area for storing the fixed key. The reason is twofold here. First, storing a fixed value does not require any volatile memory such as flip-flops and can be simply realized by burning it into the device using for example fuses (see [4] for more discussions). Second, other lightweight stream ciphers would, if used on restricted devices such as RFID, likewise require to store the key in non-volatile memory so that when the device is restarted the cipher has to be initialized with the original key. When implementations of stream ciphers are presented, the area required for storing the key is likewise ignored. Observe that the design of KTANTAN [15] likewise is based on the fact that storing fixed keys is less area consuming.

Our implementation of Sprout requires an area size of 813 GEs. In comparison, [23] states implementations of the eStream finalists using 0.13μm CMOS and the standard cell library. These require for Grain 80 an area size of 1294 GE, for Trivium 2580 GE, and for Mickey v.2 3188 GE. We note that there exist an implementation of Trivium [32] using dynamic logic which requires only 749 GEs. However the lower bound on the clock frequency of this implementation is 1 MHz, which is not achievable in restricted devices [4].

As the choice of the library impacts the area size, for the sake of a fair comparison we made our own implementation of Grain 80 (as it is the most lightweight cipher among the finalists with respect to the area size) using the same technology as has been applied for the Sprout implementation. This gave an implementation of Grain 80 that requires 1162 GE, i.e., being slightly less than the implementation stated in [23] but still being 349 GE larger compared to Sprout.

In general, realizing one register bit requires about 6 GE. Keeping in mind that Grain 80 uses registers of a total length of 160, that Trivium uses a 288 bit state, and that the internal state of Mickey v.2 is 200 bits gives a strong indication that none of these ciphers can be implemented with less than 960 GE.

As explained above, the comparisons are made under the assumption that storing fixed keys can be realized with negligible costs, following [15]. If the key is stored in registers instead, the size would increase to 1170 GE. As expected

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

there would be no gain in such cases. But even then, we think that the discussed approach has new value as it may allow for new designs with a reduced internal state size (including the key).

5 Security Discussion

In this section we discuss the security of Sprout with respect to key recovery attacks. Due to the relatively short period, coming from the fact that we aim for short internal states, Sprout is subject to distinguishing attacks. Thus, we do not consider these any further and argue that the common use case for a lightweight cipher is not to encrypt extremely large chunks of data. Moreover, as the key is assumed to be fixed, related-key attacks are out of scope as well.

We consider an attack only to be successful if the effort is clearly less than for a brute force attack, i.e. if it is less than 2^{κ}. Recall that we allow an attacker to freely choose the IV.

Algebraic Attacks. Algebraic attacks against LFSR-based keystream generators were introduced in [17] and became a powerful tool in cryptanalysis of stream ciphers. The goal of such attacks is to construct systems of algebraic equations which describe the operations used in the stream cipher, and to solve these systems for the unknown state variables. The classical algebraic attacks require that the degree of such equations is constant and therefore they usually do not work against the NLFSR-based constructions, where the algebraic degree of describing equations continuously grows. This holds in particular for Sprout.

In [3] an algebraic attack was combined with guess and determine technique in the cryptanalysis of Grain family of stream ciphers. The authors claimed that for Grain 80 and Grain 128 it is possible to construct a system of algebraic equations and to solve it for approximately half of the state-bits, when the other half is guessed. As in total 160 bits are involved in the update process (the registers of total length of 80 bits and the fixed secret key of 80 bits as well), guessing half the bits would result into an effort of at least $2^{80} = 2^{\kappa}$.

Guess and Determine Attacks. Another potential attack could be a guess and determine attack. Here, an attacker guesses parts of the internal state and/or the key and aims to recover the remaining bits more efficiently. In the following we list some arguments why we do not see an immediate application of such attacks here.

We begin our analysis of the complexity of such attack by assuming that the whole internal state is already known to the attacker who also has access to the output bits. The newly inserted key bit is not used for the output until it propagates to the position n_{t+38}. There it will become part of the monomial $n_{t+4}n_{t+38}l_{t+32}$ of the output function. Even if the attacker knows the other values of the output function, she can only recover this key bit when n_{t+4} and l_{t+32} are both equal to 1. Hence in average only one bit out of four can be recovered in this straightforward way. Observe in addition that before this particular key bit is involved in the output function for the first time, it has been used as linear terms in the NLFSR update function when it was at the position n_{39}. Thus, the key bit influences the state of the NLFSR *before* it could be recovered (which is the case with probability 1/4 only). Guessing the key bits that cannot be recovered would hence induce an additional effort of $O(2^{3\kappa/4}) = O(2^{60})$.

Moreover, we do not see any straightforward approach for reconstructing the whole internal state with an effort better than guessing. Observe that all but 8 bits of the NLFSR are used in either the update function or the round key function. Therefore, at least these need to be known if an attacker wants to know the next NLFSR state. Moreover, the NLFSR state depends on the LFSR output so that all LFSR bits need to be known as well on the long run. Of course, not all of these bits need to be known exactly. For example a few state bits can be computed directly from the output function (2 bits can be computed before the key bit propagated to the position n_{t+38} and some other register bits can be computed when the round key bits are equal to zero). However, the number of the state bits which can be recovered is relatively small and there are definitely more than 20 state bits which need to be guessed together with in average 60 key bits.

Linear Approximations. In [30] it is explained how to find a time-invariant biased linear relation between the LFSR bits and the keystream bits for Grain family of stream ciphers. This bias depends on the nonlinearity and the resiliency of the NLFSR update and of the output functions.

In the case of Sprout, such a relation would have to include the round key bits as well. However, in this case the mentioned relation will also contain the round key bits which are included non-linearly (if considered as a function of key bits and state bits). Moreover, we use in Sprout functions with the same cryptographic properties as the ones deployed in Grain 128a. These have been selected such that the bias is sufficiently small in order to make these attacks less efficient than exhaustive key search. Therefore we do not expect that they will work against Sprout.

Chosen IV Attacks. In [35] a distinguishing attack on the whole 256 round version of Grain 128 was presented. This indicates that the number of rounds during initialization and/or the nonlinearity of the functions used in Grain 128 are not high enough.

However, the update function of the NLFSR of Grain 128a (and hence of Sprout) was improved with respect to this attack. Moreover, the ratio of the number of rounds during the initialization phase to the state size is also considerably larger in case of Sprout compared to Grain family.

Dynamic Cube Attacks. A dynamic cube attack is (besides of fault attacks) the best publicly known attack against Grain 128 (see [19]). So far, no mechanisms are known to show the resistance against such attacks, especially as these rely on finding "good" cubes by chance. Hence, we cannot exclude that cube attacks may be possible. However, no cube attacks are known so far against Grain 128a and Sprout uses similar functions. Moreover, the initialization phase is longer which should further strengthen the security of the full cipher.

Time Data Memory Trade-off Attacks. The main conceptual change compared to Grain 128a, namely the involvement of the key bits into the update function, was to increase the security against time data memory trade-off attacks, following the thoughts from section 3.2. To this end, it is important that two different keys always yield states that are not keystream equivalent. Due to the fact that we use the key bit by bit, different keys should influence the keystream generation differently on the long run. However, two keys which are just shifted may result into shifted keystreams. To avoid this we include the counter bit as well. Even if the key is just shifted, due to the different counter values there will be situations where different counter bits are used in the update. The hope is that in the long run, this will ensure that different keys always produce different keystreams. However, as we have no formal proof that this is achieved for Sprout, further analysis needs to be made.

Weak Key-IV pairs. Recently [36] a distinguishing attack against different versions of Grain was proposed, that exploits the existence of weak Key-IV pairs. These result into the situation that after the initialization phase is over, the LFSR is in the all-zero state. In their attack the number of required keystream bits depends on the best linear approximation for the NFSR update function g. For Grain-128 this results into the case that 2^{86} bits are required to build a distinguisher. In case of Grain 128a (and hence of Sprout as well) the nonlinearity of the NLFSR update function is even higher meaning that even more keystream bits will be required to detect a weak pair. Furthermore, we are aiming for considerably shorter periods of the keystream. As it was mentioned we cannot guarantee that the period is higher than the 2^{40} and therefore do not recommend to produce longer keystreams under the same IV.

Fault attacks. The systematic study of fault attacks against stream ciphers was done in [28]. Usually, it is assumed that the attacker can flip one random FSR bit (without knowing its position), produce the required number of keystream bits from this faulted internal state, and then compare it with the keystream, which was produced without any faults in internal state. This process of resetting the device and introducing one fault can be done as many times as it is required for the attacker.

All members of the Grain family have been broken using this type of attack [11,8,9]. However, all these attacks aim to recover the internal state. As elaborated above, knowing the content of the register bits of Sprout does not automatically allow for efficiently recovering the secret key. Moreover, the involvement of the round key bits should make this type of attacks harder.

Side-Channel Attacks. The security of the cryptographic primitives with respect to side-channel attacks depends on the actual implementation. Hence, nothing can be said about the general vulnerability. However, a secure implementation against power-analysis attacks usually leads to a high overhead in the area and power consumption [21] which commonly depends on the number of flip-flops used [29].

Therefore, the resources required for a secure implementation even increase the necessity of developing a stream cipher solutions which require as few flip-flops as possible. Moreover, when less area and power are used to implement the scheme itself, there is more space left in order to implement the necessary countermeasures and still stay feasible in the context of restricted devices.

6 Conclusion

In this work, we discussed a different approach for realizing keystream generators. The core idea is to design a cipher where the set of internal states is split into a large number of equivalence classes such that any trade-off attack has to consider every class at least once. As a concrete approach for realizing this property, we suggest to involve the secret key not only in the initialization process but in the update procedure as well. Although the change is conceptually simple, it may allow to avoid the rule of thumb that the internal state size needs to be at least twice the key length.

Exploiting the fact that storing fixed values is less area consuming than using registers, we were able to present a new stream cipher named Sprout which has a significantly smaller area size. Sprout is considered as a proof of concept to demonstrate the feasibility of this approach. To this end, it exhibits a rather conservative design where the choice of some parameters like the register lengths has been possibly overcautious. We see it as an interesting open question if and to what extend the area size can be further reduced for stream ciphers. At the moment, we do not see any reason why stream ciphers with comparatively short registers shouldn't be possible. Consequently we see this work as a a first step towards alternative design approaches that hopefully initiates further research on this question.

Another interesting direction is the following. To thwart TMDTO attacks, it is not necessary to achieve 2^{κ} equivalence classes. Even if a cipher achieves less than these, it may still be sufficient if the effort for identifying these classes is sufficiently high. In other words, if the effort for finding all equivalence classes times the effort for executing a TMDTO attack is above the effort for exhaustive key search, we are on the safe side.

References

1. Martin Ågren, Martin Hell, Thomas Johansson, and Willi Meier. Grain-128a: a new version of grain-128 with optional authentication. *International Journal of Wireless and Mobile Computing*, 5(1):48–59, 2011.
2. Mohamed Ahmed Abdelraheem, Julia Borghoff, Erik Zenner, and Mathieu David. Cryptanalysis of the light-weight cipher a2u2. In *Cryptography and Coding*, pages 375–390. Springer, 2011.
3. Mehreen Afzal and Ashraf Masood. Algebraic cryptanalysis of a nlfsr based stream cipher. In *Information and Communication Technologies: From Theory to Applications, 2008. ICTTA 2008. 3rd International Conference on*, pages 1–6. IEEE, 2008.
4. Frederik Armknecht, Matthias Hamann, and Vasily Mikhalev. Lightweight authentication protocols on ultra-lightweight RFIDs – myths and facts. In *Workshop on RFID Security – RFIDSec'14*, Oxford, UK, July 2014.

5. Steve Babbage. Improved exhaustive search attacks on stream ciphers. In *Security and Detection, 1995., European Convention on*, pages 161–166. IET, 1995.
6. Steve Babbage and Matthew Dodd. The stream cipher mickey 2.0, 2006.
7. Steve Babbage and Matthew Dodd. The mickey stream ciphers. In *New Stream Cipher Designs*, pages 191–209. Springer, 2008.
8. Subhadeep Banik, Subhamoy Maitra, and Santanu Sarkar. A differential fault attack on grain-128a using macs. In *Security, Privacy, and Applied Cryptography Engineering*, pages 111–125. Springer, 2012.
9. Subhadeep Banik, Subhamoy Maitra, and Santanu Sarkar. A differential fault attack on the grain family of stream ciphers. In *Cryptographic Hardware and Embedded Systems–CHES 2012*, pages 122–139. Springer, 2012.
10. Elad Barkan, Eli Biham, and Adi Shamir. Rigorous bounds on cryptanalytic time/memory tradeoffs. In *Advances in Cryptology-CRYPTO 2006*, pages 1–21. Springer, 2006.
11. Alexandre Berzati, Cecile Canovas, Guilhem Castagnos, Blandine Debraize, Louis Goubin, Aline Gouget, Pascal Paillier, and Stephanie Salgado. Fault analysis of grain-128. In *Hardware-Oriented Security and Trust, 2009. HOST'09. IEEE International Workshop on*, pages 7–14. IEEE, 2009.
12. Alex Biryukov and Adi Shamir. Cryptanalytic Time/Memory/Data tradeoffs for stream ciphers. In *Advances in Cryptology–ASIACRYPT 2000*, pages 1–13. Springer, 2000.
13. Alex Biryukov, Adi Shamir, and David Wagner. Real time cryptanalysis of a5/1 on a pc. In *Fast Software Encryption*, pages 1–18. Springer, 2001.
14. Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: an ultra-lightweight block cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2007.
15. Christophe De Cannière, Orr Dunkelman, and Miroslav Knezevic. KATAN and KTANTAN - A family of small and efficient hardware-oriented block ciphers. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*, pages 272–288. Springer, 2009.
16. Christophe De Canniere and Bart Preneel. Trivium specifications. *eSTREAM, ECRYPT Stream Cipher Project*, 2006.
17. Nicolas T Courtois and Willi Meier. Algebraic attacks on stream ciphers with linear feedback. In *Advances in Cryptology - EUROCRYPT 2003*, pages 345–359. Springer, 2003.
18. Mathieu David, Damith C Ranasinghe, and Torben Larsen. A2u2: a stream cipher for printed electronics rfid tags. In *RFID (RFID), 2011 IEEE International Conference on*, pages 176–183. IEEE, 2011.
19. Itai Dinur, Tim Güneysu, Christof Paar, Adi Shamir, and Ralf Zimmermann. An experimentally verified attack on full grain-128 using dedicated reconfigurable hardware. In *Advances in Cryptology–ASIACRYPT 2011*, pages 327–343. Springer, 2011.
20. Orr Dunkelman and Nathan Keller. Treatment of the initial value in time-memory-data tradeoff attacks on stream ciphers. *Information Processing Letters*, 107(5):133–137, 2008.

21. Wieland Fischer, Berndt M Gammel, Oliver Kniffler, and Joachim Velten. Differential power analysis of stream ciphers. In *Topics in Cryptology–CT-RSA 2007*, pages 257–270. Springer, 2006.
22. JovanDj. Goli. Cryptanalysis of alleged a5 stream cipher. In Walter Fumy, editor, *Advances in Cryptology EUROCRYPT 97*, volume 1233 of *Lecture Notes in Computer Science*, pages 239–255. Springer Berlin Heidelberg, 1997.
23. Tim Good and Mohammed Benaissa. Hardware performance of estream phase-iii stream cipher candidates. In *Proc. of Workshop on the State of the Art of Stream Ciphers (SACS08)*, 2008.
24. Martin Hell, Thomas Johansson, Alexander Maximov, and Willi Meier. The grain family of stream ciphers. In *New Stream Cipher Designs*, pages 179–190. Springer, 2008.
25. Martin Hell, Thomas Johansson, W Meier, and A Maximov. A stream cipher proposal: Grain-128. estream, ecrypt stream cipher project (2006). *Available from http://www.ecrypt.eu.org/stream/p3ciphers/grain/Grain128_p3.pdf.*
26. Martin Hell, Thomas Johansson, and Willi Meier. Grain: a stream cipher for constrained environments. *International Journal of Wireless and Mobile Computing*, 2(1):86–93, 2007.
27. Martin E Hellman. A cryptanalytic time-memory trade-off. *Information Theory, IEEE Transactions on*, 26(4):401–406, 1980.
28. Jonathan J Hoch and Adi Shamir. Fault analysis of stream ciphers. In *Cryptographic Hardware and Embedded Systems-CHES 2004*, pages 240–253. Springer, 2004.
29. Shohreh Sharif Mansouri and Elena Dubrova. An architectural countermeasure against power analysis attacks for fsr-based stream ciphers. In *Constructive Side-Channel Analysis and Secure Design*, pages 54–68. Springer, 2012.
30. Alexander Maximov. Cryptanalysis of the grain family of stream ciphers. In *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, pages 283–288. ACM, 2006.
31. Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography.* CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.
32. Nele Mentens, Jan Genoe, Bart Preneel, and Ingrid Verbauwhede. A low-cost implementation of trivium. *Preproceedings of SASC*, 2008:197–204, 2008.
33. Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. In *Advances in Cryptology-CRYPTO 2003*, pages 617–630. Springer, 2003.
34. Carsten Rolfes, Axel Poschmann, Gregor Leander, and Christof Paar. Ultra-lightweight implementations for smart devices–security for 1000 gate equivalents. In *Smart Card Research and Advanced Applications*, pages 89–103. Springer, 2008.
35. Paul Stankovski. Greedy distinguishers and nonrandomness detectors. In *Progress in Cryptology-INDOCRYPT 2010*, pages 210–226. Springer, 2010.
36. Haina Zhang and Xiaoyun Wang. Cryptanalysis of stream cipher grain family. *IACR Cryptology ePrint Archive*, 2009:109, 2009.