On Analysis of Lightweight Stream Ciphers with Keyed Update

Orhun Kara, and Muhammed F. Esgin

Abstract—As the need for lightweight cryptography has grown even more due to the evolution of the Internet of Things, it has become a greater challenge for cryptographers to design ultra lightweight stream ciphers in compliance with the rule of thumb that the internal state size should be at least twice as the key size to defend against generic Time-Memory-Data Tradeoff (TMDT) attacks. However, Recently in 2015, Armknecht and Mikhalev sparked a new light on designing keystream generators (KSGs), which in turn yields stream ciphers, with small internal states, called *KSG with Keyed Update Function (KSG with KUF)*, and gave a concrete construction named Sprout. But, currently, security analysis of KSGs with KUF in a general setting is almost non-existent. Our contribution in this paper is two-fold. 1) We give a general mathematical setting for KSGs with KUF, and for the first time, analyze a class of such KSGs, called KSGs with Boolean Keyed Feedback Function (KSG with Boolean KFF), generically. In particular, we develop two generic attack algorithms applicable to any KSG with Boolean KFF having almost arbitrary output and feedback functions where the only requirement is that the secret key incorporation is biased. We introduce an upper bound for the time complexity of the first algorithm. Our extensive experiments validate our algorithms and assumptions made thereof. 2) We study Sprout to show the effectiveness of our algorithms in a practical instance. A straightforward application of our generic algorithm yields one of the most successful attacks on Sprout.

Index Terms—Lightweight Cipher, Keystream generator, Stream cipher, Time-Memory-Data tradeoff, Keyed Update Function, Symmetric Encryption, Sprout.

1 Introduction

Considering a resource-constrained device such as an RFID tag or a wireless sensor, a crucial dilemma arises between energy/power management and data security such as confidentiality. This dilemma introduces a real challenge to design *lightweight cryptographic algorithms* (those with extremely small costs in area, power, energy consumption, etc.) to be deployed in resource-constrained devices.

We see many examples of recent ultra lightweight block ciphers such as PRESENT [1], KTANTAN [2], LED [3], Piccolo [4], Midori [5], SIMON/SPECK [6] and their new variant Simeck [7]. The current technology is adequate to design secure ultra lightweight block ciphers (we can simply consider those ciphers as having hardware area cost less than 1000 GEs). However, there is almost no modern ultra lightweight stream cipher. The main reason behind this phenomena lies on a design principle about the internal state size of stream ciphers imposed due to the tradeoff attacks [8], [9], [10]. This principle states that the internal state size of a stream cipher must be at least 2κ bits to provide a κ-bit security level.

Armknecht and Mikhalev [11] proposed a new keystream model for stream ciphers at FSE 2015, called *keystream generators (KSGs) with Keyed Update Function (KUF)*, where the secret key is used in the state update to claim the resistance against the conventional tradeoff attacks while using a small internal state. The ideas in [11] are embodied in a stream cipher, called Sprout. Sprout may be considered as one of the first modern ultra lightweight stream ciphers in academia with its hardware area cost less than 1000 GEs. Even though this first attempt with Sprout was broken shortly after its introduction (see, for example, [12], [13]), the proposed idea immediately attracted great attention from cryptology community (see Section 1.1). However, a detailed security analysis of KSGs with KUF has not been provided up until now, leaving several open questions about the security of KSGs with KUF.

This paper contributes to filling this gap in instantiation of the ideas presented in [11] by studying the security of keystream generators with keyed update functions in a mathematical setting. More precisely, we first set a mathematical framework for KSGs with KUF and, for the first time, mount two generic internal state recovery attacks on a family of keystream generators with KUF, which we call *keystream generators with Boolean Keyed Feedback Function (KSGs with Boolean KFF)*. The main concepts along with the core of our attacks are given in Section 2, Section 3 and Section 4. We introduce a new tradeoff attack, which makes use of special states, called *weak states*, in Section 5. The complexity analyses of our attacks are given in Section 6. We validate our assumptions and results regarding our generic attacks through extensive experiments (Section 7). Then, we take Sprout as an instantiation of a KSG with Boolean KFF, and apply our generic attacks to it (Section 8). Our improved generic attack performs as efficient as the best known attacks on Sprout [13], [14].

- *Orhun Kara is with TÜBİTAK BİLGEM UEKAE National Research Institute of Cryptology and Electronics, 41470 Gebze Kocaeli/Turkey. E-mail: orhun.kara@tubitak.gov.tr*
- *Muhammed F. Esgin is with the Faculty of Information Technology, Monash University, Australia and Data61, CSIRO, Australia. E-mail: muhammed.esgin@monash.edu*

1.1 Related work

The security of a generic KSG with KUF is addressed for the first time in this work. All the previous works are related

TABLE 1
Comparison of attacks on Sprout. The time complexities are given as number of encryptions and the number of table lookups (TLs), if exists. We assume 2^{21}-bit keystream is produced in one minute on a standard PC as done in [13]. d indicates the data complexity and also the order of weak states. It is always possible to update these states at least d times independent of the key and their cardinality is roughly 2^{80-d}.

	Time	**Data**	**Memory**
[12]	2^{70}	negl.	2^{46}
[15]	2^{75}	negl.	negl.
Sec. 3 in [16]	2^{70}	negl.	negl.
Sec. 5 in [16]	$2^{66.7}$	2^{42}	negl.
Guess&Determine in [13]	2^{68}	negl.	negl.
TMDT attack in [13]	$2^d\ TLs + 2^{71-d}$	2^d	2^{86-d}
TMDT attack in [14]	$2^{d-3}\ TLs + 2^{80-d}$	2^d	2^{86-d}
Sec. 2 of this work	$2^d\ TLs + 2^{78-d}$	2^d	2^{86-d}
Sec. 3 of this work	$2^d\ TLs + 2^{71-d}$	2^d	2^{86-d}

to the analysis of a specific example, in particular, Sprout. There has been several works published in a short time related to the analysis of Sprout [12], [13], [14], [15], [16]. However, the security of the new approach Sprout's design philosophy brings has been left open.

Once realized that Sprout can be broken, a successor of Sprout is proposed in [17]. The new cipher, Plantlet, is very similar to Sprout and one of the differences is that the internal state size is a bit larger. One crucial difference is also that the bias in the secret key's incorporation into the feedback is fixed (i.e., the *guess capacity*, given in Definition 3, is set to one-half). Therefore, even though Plantlet also falls into our definition of KSGs with Boolean KFF, our attacks do not work on it.

Another recent lightweight stream cipher is Lizard [18]. It also has a Grain-like structure like Sprout and Plantlet. However, Lizard has a rather different design approach, referred as $FP(1)-mode$, as opposed to the one followed in Sprout and Plantlet. The key is not used in the state update function. Thus, Lizard does not fall into the class of KSGs with KUF.

The most successful attacks mounted on Sprout are the tradeoff attacks by Zhang and Gong [14] and by Esgin and Kara [13]. We have detected a flaw in preparing the tradeoff tables in Zhang and Gong attack in [14] and have introduced a fixed version with the accurate workloads. We compare these two attacks in more detail in Section 9. An overview of the complexities of the attacks on Sprout are given in Table 1. As can be seen from Table 1, Zhang-Gong attack is 2^9 times slower than Esgin-Kara attack for the same amount of data and memory.

Generating multiple output bits from a given internal state of Sprout has been exploited in some previous works [12], [13], [14]. We generalize this property and call it *the output capacity* as given in Definition 4. On the other hand, this property plays a minor role in our attack and our internal state recovery attacks are independent of the output generation function.

1.2 Overview of our main contribution

We analyze KSGs with Boolean KFF where the *key-dependent* part of the Keyed Update Function (KUF) is a Boolean function. That is, only one bit of the output of KUF depends on the key. The remaining feedback values, if exist, are deduced from only the internal state.

Our focus is mainly on the internal state recovery, which is the dominant part of our attack. Given an internal state candidate, the goal at each iteration is the same: either determine the next feedback value from the output (*determine case*) or check the state and then guess the next feedback value as both 0 and 1 (*check-and-guess case*). This means that the whole internal state is known at each iteration and the only unknown to be captured is the next feedback value. Any internal state can be checked through our new algorithms, Algorithm 2.1 or Algorithm 3.1, if it can produce a given output without knowing the key when the feedback function produces predictable output bits on average (i.e., with probability > 0.5). Hence, what we exploit mainly is the biased incorporation of key bits into the feedback function during internal state updates. Simply put, we call the advantage of guessing a feedback value from a given internal state alone as the *guess capacity* (see Definition 3). The generic attacks work when the average guess capacity is larger than one-half.

It is possible to recover the correct internal state without knowing the key if the register is clocked enough number of times at each test, thanks to the guess capacity. The exact feedback values are determined after recovering the internal state. The last step is solving a system of equations generated by the outputs of the feedback functions in order to recover the key. The details are given in Section 2, Section 3 and Section 4.

Both Algorithm 2.1 and Algorithm 3.1 are generic attacks and do not exploit the internal structures of output functions, feedback functions, their input sizes or tap points. So, the attacks can successfully be applied to any *KSGs with Boolean KFF* (see Definition 2 for a formal description) having average guess capacities higher than one-half. Particularly, Algorithm 2.1 and Algorithm 3.1 can be mounted on Sprout even if its output function or tap points are modified.

Furthermore, we introduce a concept of *weak internal states* so as to minimize the number of internal states to be examined through Algorithm 2.1 or Algorithm 3.1, and establish a tradeoff between data, time and memory complexities. A weak state, in simple terms, is a state that can be updated and hence can produce the output up to some degree without the key. We roughly call the number of possible consequent update steps which are independent of the key as the weakness order, d. We show that the efficiency of the tradeoff is relevant to d.

We analyze the complexity of Algorithm 2.1 theoretically and conduct several experiments to verify the theoretical results, and compare the performance of Algorithm 2.1 and Algorithm 3.1. Theorem 1 gives a lower bound for the success rate of Algorithm 2.1. Our experiments show that the success rates are much higher than those indicated by Theorem 1 (see Figure 2) and the time complexity can be reduced to its half to achieve the same success rate claimed in Theorem 1 (see Table 3). We also show that the performance of Algorithm 3.1 is equivalent to the performance of Algorithm 2.1 when there is no deviation among the guess capacities of individual states from the average guess capacity. One remarkable result is that the time complexity of Algorithm 3.1 gets much better as the deviation gets higher (see Figure 2 and Table 3).

To give a concrete example of our method, we apply the generic attack on Sprout as a KSG with Boolean KFF. The asymptotic complexities of Algorithm 2.1 and Algorithm 3.1 are the same and equivalent to that of the tradeoff attack in [13]. However, in practice, Algorithm 3.1 is as fast as the attack in [13] and Algorithm 2.1 is roughly 31 times slower. The workloads can be in practical limits for some values of d as given in Table 1. Our internal state recovery algorithms are generic and do not exploit the internal structures of the building blocks of Sprout. For instance, it is possible to extract the feedback value from the first output bit it appears in the backward iteration of Sprout[1] since the feedback value is incorporated as a linear term. This property is exploited in [13], [14] whereas Algorithm 2.1 and Algorithm 3.1 still work well without this property.

2 Internal State Recovery Attack

In this section, we present an internal state recovery attack to a family of KSGs with KUF, which forms the core of our contribution together with its improved version in Section 3. Armknecht and Mikhalev give a definition of a generic keystream generator (KSG) with Keyed Update Function (KUF) [11]. We repeat it in Definition 1. Note that the initialization phase is discarded since our attack is independent of that part. First, we introduce some notations to be used throughout the paper.

- s - internal state size of a KSG
- t - clock-cycle
- S_t - internal state of a KSG at clock-cycle t. When the clock-cycle is not emphasized, t is omitted.
- z_t - keystream bit at clock-cycle t
- $f_{\mathcal{F}}, f_{\mathcal{B}}$ - Boolean keyed feedback functions in forward and backward directions, respectively (Defn. 2)
- $\Pr_g$ - average guess capacity (Defn. 3)
- $\Pr_d$ - the probability of determining the next feedback value from output (Defn. 5)
- θ - output capacity (Defn. 4)
- d - the number of feedback values that can be computed independent of the secret key for *weak states* (Defn. 6)
- $\#MM$ - the number of mismatches (see Section 2)
- α_{ter} - the (maximum) number of clocks iterated until Algorithm 2.1 or Algorithm 3.1 terminates
- α_{thr} - threshold value used by Algorithm 2.1
- $\Pr_{thr}$ - threshold probability used by Algorithm 3.1

Definition 1 ([11]). *A keystream generator (KSG) with Keyed Update Function (KUF) consists of the following functions.*

- *An update function $\mathcal{F} : \mathcal{K} \times \mathcal{S} \to \mathcal{S}$ such that $\mathcal{F}_K : \mathcal{S} \to \mathcal{S}$ is bijective for any $K \in \mathcal{K}$ (i.e., registers are nondegenerate), and*
- *A Boolean output function $\mathcal{G} : \mathcal{S} \to GF(2)$*

where $\mathcal{K} = GF(2)^k$ is the key space and $\mathcal{S} = GF(2)^s$ is the internal state space.

A KSG with KUF can also be clocked in backward direction since $\mathcal{F}_K : \mathcal{S} \to \mathcal{S}$ is bijective. We denote the update function in backward direction by $\mathcal{B} : \mathcal{K} \times \mathcal{S} \to \mathcal{S}$ and $\mathcal{B}_K : \mathcal{S} \to \mathcal{S}$. Then, $\mathcal{F}_K \circ \mathcal{B}_K = \mathcal{B}_K \circ \mathcal{F}_K = id$ where id denotes the identity function. Our focus is on a specific class of KSGs with KUF described in the following definition.

Definition 2. *A KSG with KUF is called a KSG with Boolean KFF (Keyed Feedback Function) if*

- *The update function $\mathcal{F}$ and the output function $\mathcal{G}$ are executed once for each one-bit output generation, and*
- *Only a single bit of the output of $\mathcal{F}(K, S)$ depends on the key. Similarly, only a single bit of the output of $\mathcal{B}(K, S)$ in the backward direction depends on the key.*

The key bits incorporated into the update function $\mathcal{F}(K, S)$ along with the state bits form a Boolean function $f_{\mathcal{F}}(K, S)$. We call $f_{\mathcal{F}}(K, S)$ the keyed feedback function of the KSG in forward direction. Similarly, we call $f_{\mathcal{B}}(K, S)$ the keyed feedback function of the KSG in backward direction.

The family of KSGs with KUF analyzed in this paper is the model given in Definition 2. The details of neither the feedback function nor the output function is important for the internal state recovery attacks given in Algorithm 2.1 and Algorithm 3.1. We concentrate only on the *key-dependent* feedback value. Thus, we mean the output of $f_{\mathcal{F}}(K, S)$ (or $f_{\mathcal{B}}(K, S)$ in backward direction) when we say "feedback value" in the rest of the paper. The following definition plays a crucial role in our attacks.

Definition 3. *For a given KSG with Boolean KFF having s-bit internal state, k-bit key, and $f_{\mathcal{F}}$ and $f_{\mathcal{B}}$ as its Boolean keyed feedback functions, we define the guess capacity of an internal state S in the forward direction as*

$$\Pr_g(S)_f = \frac{1}{2} + \left| \frac{\#\{K : f_{\mathcal{F}}(K, S) = 0\}}{2^k} - \frac{1}{2} \right|,$$

and in the backward direction as

$$\Pr_g(S)_b = \frac{1}{2} + \left| \frac{\#\{K : f_{\mathcal{B}}(K, S) = 0\}}{2^k} - \frac{1}{2} \right|.$$

We define the average guess capacity over all the states as

$$\Pr_g = \frac{1}{2} + 2^{-s} \sum_{S} \left| \frac{\#\{K : f_{\mathcal{F}}(K, S) = 0\}}{2^k} - \frac{1}{2} \right|.$$

That is, $\Pr_g$ is the average probability of guessing an arbitrary feedback value $f_{\mathcal{F}}(K, S)$ correctly for a known arbitrary internal state S and an unknown key K. Note that $\Pr_g$ should be greater than $1/2$ in order for our attacks to be successful.

Remark 1. *A definition for the average guess capacity in the backward direction is omitted since it is simply equal to the average guess capacity in the forward direction. This fact is due to the observation that the guess capacity of an internal state in the forward direction is equal to the guess capacity of its successor in the backward direction.*

One more property of KSGs with Boolean KFF related to our attack is the latency of a feedback value (in either forward or backward direction) being incorporated into the output function.

Definition 4. *For a KSG with Boolean KFF, define θ_f as the largest integer such that the output bits $z_{t+1}, \ldots, z_{t+\theta_f}$ can be computed from the current state S_t without any state updates in*

1. We generalize this concept and call it *determine probability*. See Definition 5.

forward direction (i.e., independent of the key). Similarly, define θ_b as the largest integer such that the output bits $z_{t-1}, \ldots, z_{t-\theta_b}$ can be computed from the current state S_t without any state updates in backward direction. Also, define $\theta = \theta_f + \theta_b$. We call θ_f as the output capacity in forward direction, θ_b as the output capacity in backward direction and θ as the output capacity.

Coupled with the fact that the output bit z_t can always be produced from any given state S_t, one can compute $\theta + 1$ bits from any given state independent of the key.

Furthermore, from Definition 4, it is clear that for any state S_t, the next feedback value in forward direction $fb_{t+1} := f_{\mathcal{F}}(K, S_t)$ can appear at the output at clock-cycle $t + 1 + \theta_f$ the earliest. Otherwise, the computation of $z_{t+\theta_f}$ from S_t without any state updates would not be guaranteed. Moreover, no other next feedback value $(fb_{t+2}, fb_{t+3}, \ldots)$ can appear in $z_{t+1+\theta_f}$ (i.e., the only possible unknown in $z_{t+1+\theta_f}$ is fb_{t+1}) since S_{t+1} can produce $z_{t+1+\theta_f}$ by definition. Similar arguments can be made in backward iterations. These arguments suggest another means of computing the next feedback value through the output bit and pave the way for the following definition of the probability of determining the feedback value from the output.

Definition 5. *We define $\Pr_f$ as the probability of flipping the output bit $z_{t+1+\theta_f}$ over all states S_t when the feedback value $fb_{t+1} = f_{\mathcal{F}}(K, S_t)$ is flipped while keeping S_t unchanged. We also define $\Pr_b$ analogously in backward direction. We call $\Pr_f$ and $\Pr_b$ as the probabilities of determining the next feedback value in forward and backward directions, respectively. Let their maximum be $\Pr_d = \max\{\Pr_f, \Pr_b\}$. We call $\Pr_d$ as the probability of determining the next feedback value.*

Note that $\Pr_d$ indicates the probability of determining the feedback value from the output when the internal state is known. One can determine the feedback value fb_{t+1} from the output for the state S_t if flipping fb_{t+1} causes a change in the corresponding output bit.

Making use of these definitions, we now introduce a generic internal state recovery attack on KSGs with Boolean KFF. Without loss of generality, we assume $\Pr_d = \Pr_f$ and explain all the attacks in the forward iteration of a cipher.

Algorithm 2.1 is an internal state recovery procedure that checks if there exists a correct state (i.e., a state producing a given keystream sequence) in a set of given states. The algorithm proceeds as follows. The feedback values $fb_{t+1}, fb_{t+2}, \ldots, fb_{t+e}$ are predicted for a given state S_t *one by one*. At a particular step i, we examine if a feedback value fb_{t+i} can be determined from the output bit $z_{t+i+\theta_f}$ through Algorithm 2.2. Otherwise, we check if the current state S_{t+i-1} can generate the output bit $z_{t+i+\theta_f}$ and guess fb_{t+i} through Algorithm 2.3.

We make use of the guess capacity of a state to check if the feedback value is the expected one. An unexpected value is considered as a *mismatch*. We expect much fewer number of mismatches for a correct state whereas the number of mismatches for a wrong state is expected to be half of the total number of iterations.

Algorithm 2.1 continues on recovering the next feedback values for each examined state while keeping counts of their number of mismatched feedback values. The counting procedure lasts from the t-th clock up to $(t + \alpha_{ter} - 1)$-th

Algorithm 2.1 Internal State Recovery

```
1:  Input: Non-empty set of internal state candidates, $\mathbb{S}$; keystream $\{z_{t+1+\theta_f}, \ldots, z_{t+\theta_f+\alpha_{ter}}\}$; the maximum number of clocks for each test, $\alpha_{ter}$; average guess capacity, $\Pr_g$; miss event probability $\epsilon$
2:  Set $\epsilon_{ter} = \sqrt{\frac{-\ln \epsilon}{2\alpha_{ter}}}$
3:  Set $\alpha_{thr} = \lfloor \alpha_{ter}(1 - \Pr_g + \epsilon_{ter}) \rfloor$
4:  Initialize CUR and NEW as two empty sets
5:  Load all the states in $\mathbb{S}$ into CUR
6:  Set $\#MM(S)$ of each state $S$ in CUR as zero
7:  Make a copy of each state $S$ in CUR as the $root$ of $S$
8:  // Initially the root of each state $S$ is itself
9:  for each clock $i$ from $t$ to $(t + \alpha_{ter} - 1)$ do
10:     for each state $S$ in CUR do
11:         Compute $\Pr_g(S)_f$
12:         if $\Pr_g(S)_f = 0.5$ then
13:             Set $fb_{sugg} = 0$
14:             // 0 is set as the default. It may also be set as 1 or chosen randomly each time.
15:         else
16:             Set $fb_{sugg}$ as the feedback value of $S$ suggested through the keyed feedback function
17:             // Since $\Pr_g(S)_f > 0.5$, the feedback value can be predicted with probability $> 0.5$
18:         end if
19:         if the feedback value $fb_{i+1}$ of $S$ is possible to determine from the output bit $z_{i+1+\theta_f}$ then
20:             Run Determine Procedure (Algorithm 2.2)
21:             // $fb_{i+1}$ is determined, $S$ and $\#MM(S)$ are updated and loaded into NEW
22:         else
23:             Run Check-and-guess Procedure (Algorithm 2.3)
24:             // If $S$ produces the correct output bit, both possible next states are loaded into NEW with their corresponding mismatch counts
25:         end if
26:     end for
27:     Terminate if NEW is empty and give no output
28:     Copy NEW to CUR
29:     Empty NEW
30: end for
31: Output: the roots in CUR as the candidates for the correct state at clock $t$
```

Algorithm 2.2 Determine Procedure

```
1: Determine the feedback value as $fb_{det}$ from the corresponding output bit
2: Update $S$ by clocking it with the feedback value $fb_{det}$
3: if $fb_{sugg} \neq fb_{det}$ (it is a mismatch) then
4:     Increment $\#MM(S)$ by one
5: end if
6: if $\#MM(S) \leq \alpha_{thr}$ then
7:     Add updated $S$ with $\#MM(S)$ and its root to NEW
8: end if
```

Algorithm 2.3 Check-and-guess Procedure

```
1: if the output of $S$ is not equal to the actual output at the corresponding clock then
2:     Do nothing (Eliminate the state)
3: else
4:     Make two copies $S_0$, $S_1$ of $S$
5:     Set $\#MM(S_0) = \#MM(S_1) := \#MM(S)$
6:     Set the feedback value to 0 for $S_0$ and update $S_0$
7:     Set the feedback value to 1 for $S_1$ and update $S_1$
8:     if $fb_{sugg} = 0$ then
9:         Increment $\#MM(S_1)$ by one
10:    else
11:        Increment $\#MM(S_0)$ by one
12:    end if
13:    if $\#MM(S_0) \leq \alpha_{thr}$ then
14:        Add $S_0$ along with $\#MM(S_0)$ to NEW and set the root of $S$ as its root
15:    end if
16:    if $\#MM(S_1) \leq \alpha_{thr}$ then
17:        Add $S_1$ along with $\#MM(S_1)$ to NEW and set the root of $S$ as its root
18:    end if
19: end if
```

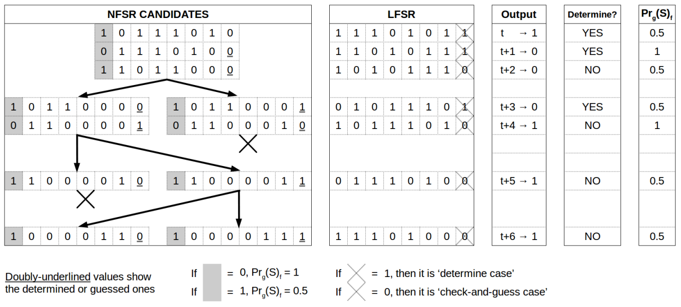
NFSR CANDIDATES
LFSR
Output
Determine?
Pr$_g$(S)$_f$
Doubly-underlined values show the determined or guessed ones
If ▨ = 0, Pr$_g$(S)$_f$ = 1
If ▨ = 1, Pr$_g$(S)$_f$ = 0.5
If ☒ = 1, then it is 'determine case'
If ☒ = 0, then it is 'check-and-guess case'

Fig. 1. Example iteration of Algorithm 2.1.

clock for each state as long as the state is not eliminated due to exceeding a threshold for the number of mismatches.

We expect approximately $\alpha_{ter}(1-\Pr_g)$ mismatches for a correct state and $\alpha_{ter}/2$ mismatches for a wrong state. Note that $\alpha_{ter}(1-\Pr_g)$ can be much smaller than $\alpha_{ter}/2$ for large values of $\Pr_g$. We denote the number of mismatches for a state S by $\#MM(S)$. One can choose a proper threshold value α_{thr} such that $\alpha_{ter}(1-\Pr_g) \leq \alpha_{thr} \leq \alpha_{ter}/2$ and eliminate any state whose number of mismatches exceeds α_{thr}. The value α_{thr} is determined in order to fine-tune the miss event and false alarm probabilities. We investigate this adjustment theoretically in Theorem 1 and experimentally in Section 7.

Example iteration of Algorithm 2.1

Let us give a toy example of how Algorithm 2.1 iterates. We define a simple keystream generator consisting of two registers: an 8-bit NFSR $N_t = (n_0^t, \ldots, n_7^t)$ and an 8-bit LFSR $L_t = (l_0^t, \ldots, l_7^t)$ (a toy version of Sprout [11]). Let $S_t = (N_t, L_t)$. LFSR runs by itself using the update function $l_7^{t+1} = l_0^t \oplus l_3^t \oplus l_4^t \oplus l_5^t$, and NFSR is updated by a function $n_7^{t+1} = l_0^t \oplus n_6^t \oplus n_0 k_t \oplus n_1^t n_2^t$ where k_t is the key bit. Finally, the output at time t is produced as $z_t = l_6^t n_7^t \oplus n_6^t$. It is easy to see that this cipher is a KSG with Boolean KFF.

Note that $n_7^{t+1} = z_{t+1} \oplus n_7^t$ when $l_7^t = 1$. Hence, we can determine the next feedback value $fb_{t+1} = n_7^{t+1}$ from the output bit z_{t+1}, with probability 1/2 (when $l_7^t = 1$). That is, $\Pr_f = 1/2$. When $l_7^t = 0$, then $z_{t+1} = n_7^t$, meaning that we can check if the current state at clock-cycle t can generate the output at $t+1$. Here, observe also that $\theta_f = 0$ as n_7^t is incorporated into the output z_t. Furthermore, $\Pr_g(S_t)_f = 1$ when $n_0^t = 0$, and $\Pr_g(S_t)_f = 1/2$ otherwise. Hence, the average guess capacity $\Pr_g = 0.75$. We give an example iteration of Algorithm 2.1 for the state $S_t = ((1,0,1,1,1,0,1,0),(1,1,1,0,1,0,1,1))$ for 6 clocks as given in Figure 1 and trace it step-by-step in detail below.

- **t** : $l_7^t = 1$, determine case: $n_7^{t+1} = z_{t+1} \oplus n_7^t = 0$.
- $\mathbf{t+1}$: $l_7^{t+1} = 1$, determine case: $n_7^{t+2} = z_{t+2} \oplus n_7^{t+1} = 0$.
- $\mathbf{t+2}$: $l_7^{t+2} = 0$, check-and-guess case: It produces the correct next output. Hence, make two copies of the state with the feedback values as 0 and 1 (see Figure 1).
- $\mathbf{t+3}$: $l_7^{t+3} = 1$, determine case for both states: $n_7^{t+4} = 1$ for the first state and $n_7^{t+4} = 0$ for the second state.
- $\mathbf{t+4}$: $l_7^{t+4} = 0$, check-and-guess case: Eliminate the second state as it cannot produce the next output. Make two copies of the other with feedback values 0 and 1.
- $\mathbf{t+5}$: $l_7^{t+5} = 0$, check-and-guess case: Eliminate the first state as it cannot produce the next output. Make two copies of the other with feedback values 0 and 1.

Consider the NFSR updates from 10111010 to 10000111 (the state on the right bottom in Figure 1). The feedback values are $0,0,0,1,1,1$. On the other hand, the suggested feedback values are $0,0,0,0,0,0$. Hence, the number of mismatches $\#MM = 3$ for this case.

3 Improved Internal State Recovery

We introduce another algorithm in this section for internal state recovery of a KSG with Boolean KFF. This algorithm is similar to Algorithm 2.1, but outruns Algorithm 2.1 when there are more states deviating from the average guess capacity. The comparisons can be found in Section 7.

The main tool to distinguish a wrong state from a right state in Algorithm 2.1 is counting the number of mismatches. However, we introduce a new notion *threshold probability*, $\Pr_{thr}$, to distinguish a wrong state from a right state in Algorithm 3.1. Instead of counting the number of mismatches in each clocking of the keystream of a state, we recursively multiply the *check probability*, $\Pr_{Ch}(S)$, of the state with $(1-\Pr_g(S)_f)$ in case of a mismatch. We expect that the check probability of a wrong state will converge to zero much faster than the check probability of a right state since a wrong state is expected to produce much more mismatches. The notion of $\Pr_{thr}$ is introduced to measure how fast the check probability of a state converges to zero. The check probability of a wrong state is expected to fall below $\Pr_{thr}$ immediately.

Determining the feedback from the output value in Algorithm 3.2 is similar to that in Algorithm 2.2. The only difference is that the value of $\Pr_{Ch}(S)$ is updated iteratively instead of counting the number of mismatches. The update procedure consists of multiplying $\Pr_{Ch}(S)$ with $(1-\Pr_g(S)_f)$ in case of a mismatch. The check-and-guess

Algorithm 3.1 Improved Internal State Recovery

```
 1: Input: $\mathbb{S}$; $\{z_{t+1+\theta_f}, \ldots, z_{t+\theta_f+\alpha_{ter}}\}$; $\alpha_{ter}$; $\Pr_{thr}$
 2: Initialize CUR and NEW as two empty sets
 3: Load all the states in $\mathbb{S}$ into CUR
 4: Set $\Pr_{Ch}(S)$ of each state $S$ in CUR as 1
 5: Make a copy of each state $S$ in CUR as the root of $S$
 6: for each clock $i$ from $t$ to $(t + \alpha_{ter} - 1)$ do
 7:     for each state $S$ in CUR do
 8:         Compute $\Pr_g(S)_f$
 9:         if $\Pr_g(S)_f = 0.5$ then
10:             Set $fb_{sugg} = 0$
11:         else
12:             Set $fb_{sugg}$ as the feedback value of $S$ suggested through the keyed feedback function
13:         end if
14:         if the feedback value $fb_{i+1}$ of $S$ is possible to determine from output bit $z_{i+1+\theta_f}$ then
15:             Run Determine Procedure (Algorithm 3.2)
16:         else
17:             Run Check-and-guess Procedure (Algorithm 3.3)
18:         end if
19:     end for
20:     Terminate if NEW is empty and give no output
21:     Copy NEW to CUR
22:     Empty NEW
23: end for
24: Output: the roots in CUR at clock $t$
```

Algorithm 3.2 Determine Procedure

```
1: Determine the feedback value as $fb_{det}$ from the corresponding output
2: Update $S$ by clocking it with the feedback value $fb_{det}$
3: if $fb_{sugg} \neq fb_{det}$ (it is a mismatch) then
4:     $\Pr_{Ch}(S) = \Pr_{Ch}(S) \cdot (1 - \Pr_g(S)_f)$
5: end if
6: if $\Pr_{Ch}(S) \geq \Pr_{thr}$ then
7:     Add updated $S$ with $\Pr_{Ch}(S)$ and its root to NEW
8: end if
```

procedure is modified analogously for the improved attack as given in Algorithm 3.3.

One can check that Algorithm 2.1 and Algorithm 3.1 are equivalent when the guess capacities of all the states are equal. For each mismatch, we have $\Pr_{Ch}(S) = \Pr_{Ch}(S) \cdot (1 - \Pr_g(S)_f) = \Pr_{Ch}(S) \cdot (1 - \Pr_g)$ and hence taking $\Pr_{thr} = (1 - \Pr_g)^{\alpha_{thr}}$ will result in the same miss event and false alarm probabilities with the same complexity.

The state on the right bottom of Figure 1 of the example in Section 2 has a mismatch at $t + 4$ with $\Pr_g(S)_f = 1$. Therefore, it is instantly eliminated through Algorithm 3.1, and only the state on the left bottom proceeds forward.

The performance of Algorithm 3.1 gets better with the amount of states deviating from the average guess capacity. A concrete example is the cipher Sprout. Algorithm 3.1 is much faster than Algorithm 2.1 when mounted on Sprout (see Section 8.2).

Algorithm 3.3 Check-and-guess Procedure

```
 1: if the output of $S$ is not equal to the actual output then
 2:     Do nothing (Eliminate the state)
 3: else
 4:     Make two copies $S_0$, $S_1$ of $S$
 5:     Set $\Pr_{Ch}(S_0) = \Pr_{Ch}(S_1) := \Pr_{Ch}(S)$
 6:     Set the feedback value to 0 for $S_0$ and update $S_0$
 7:     Set the feedback value to 1 for $S_1$ and update $S_1$
 8:     if $fb_{sugg} = 0$ then
 9:         $\Pr_{Ch}(S_1) = \Pr_{Ch}(S_1) \cdot (1 - \Pr_g(S)_f)$
10:     else
11:         $\Pr_{Ch}(S_0) = \Pr_{Ch}(S_0) \cdot (1 - \Pr_g(S)_f)$
12:     end if
13:     if $\Pr_{Ch}(S_0) \geq \Pr_{thr}$ then
14:         Add $S_0$ along with $\Pr_{Ch}(S_0)$ to NEW and set the root of $S$ as its root
15:     end if
16:     if $\Pr_{Ch}(S_1) \geq \Pr_{thr}$ then
17:         Add $S_1$ along with $\Pr_{Ch}(S_1)$ to NEW and set the root of $S$ as its root
18:     end if
19: end if
```

4 Key Recovery Phase

This phase depends heavily on the structure of feedback function of the concrete cipher on which the attack is mounted. The internal state recovery phase gives us the correct internal state at a certain clock t. However, we still do not know the key. Moreover, the correct state may have several subsequent state candidates produced by Algorithm 2.1 since we make guesses and several descendants of the correct state may be proposed during the subsequent clocks. Hence, we may not know the exact feedback values.

Let us assume we need e feedback values $fb_{t+1}, \ldots, fb_{t+e}$ to recover the key partially from the correct state S_t so that the remaining key bits can be deduced by an exhaustive search with an ignorable workload. Note that the parameter e depends on the structure of a specific cipher. After obtaining the correct internal state S_t at time t via Algorithm 2.1, we are supposed to solve three issues for the key recovery.

1) Recover the correct internal state S_{t+e} at clock $t + e$,
2) Compute the exact feedback values $fb_{t+1}, \ldots, fb_{t+e}$, and
3) Extract information about the key from the feedback values.

Run Algorithm 2.1 once more just for S_t. This time, keep all the candidates for the descendants of S_t at clock $t + e$, and save the feedback values for each surviving descendant. These are possible by slightly modifying Algorithm 2.1.

Run Algorithm 2.1 once again by defining all the descendants of S_t at clock $t + e$ as input states. Algorithm 2.1 gives the correct state for S_{t+e} among all the descendants of S_t. Hence, we recover the correct state at time $t + e$ and all the feedback values from time t to $t + e - 1$, namely $fb_{t+1}, \ldots, fb_{t+e}$. The last step is to extract information about the key from the system of equations

$$\begin{aligned} f_{\mathcal{F}}(K, S_t) &= fb_{t+1} \\ &\vdots \\ f_{\mathcal{F}}(K, S_{t+e-1}) &= fb_{t+e} \end{aligned}$$

for the unknown key K. The feedback function $f_{\mathcal{F}}$ for a lightweight stream cipher is expected to be algebraically simple and the system will be easy to solve. The solving procedure depends on the concrete design. However, in the worst case, if $f_{\mathcal{F}}$ is too complicated then one may guess the key bits incorporated into $f_{\mathcal{F}}$ at a given clock. Most probably, $f_{\mathcal{F}}$ does not take all the key bits as input. Hence, one may mount a divide-and-conquer attack. We have referenced to Algorithm 2.1 throughout this section, but similar arguments hold for Algorithm 3.1 as well.

In the case of Sprout, *Round Key Function* is so simple that recovering the key from the correct internal state is computationally trivial (see Section 8 for details). Even we do not need to recover S_{t+e}, thanks to its very high determine probability in backward direction ($\Pr_b = 1$ for Sprout). We refer the reader for a key recovery attack on Sprout to [13].

5 Converting State Recovery to Tradeoff

A critical question now is which states will be given as an input to Algorithm 2.1 or Algorithm 3.1. A trivial answer is to input all possible internal state candidates, which corresponds to exhaustively searching all the states. To tackle this problem, we introduce a tradeoff attack so as to minimize the time complexity.

If there are internal states that can produce many output bits independent of the key, then we can acquire these states together with their outputs in a precomputation and run Algorithm 2.1 or Algorithm 3.1 to check their validity in the online phase. In this way, we can optimize time, data and memory workloads. Therefore, we are interested in the capacity of clocking a state without the key and introduce the concept of weak internal states.

Definition 6. *For a given KSG with Boolean KFF, a state S_t at a clock-cycle t is called a weak state of order d in forward direction if $S_{t+1}, \dots, S_{t+d}$ can be computed from S_t without knowing the key (independently from the key). Analogously, a state S_t is called a weak state of order d in backward direction if $S_{t-1}, \dots, S_{t-d}$ can be computed from S_t without knowing the key.*

It is straightforward from the definition that a weak state of order d is also a weak state of order r for any $r \le d$. Also, for a weak state S_t of order d in forward direction, S_{t+i} is a weak state of order $d-i$ in forward direction and a weak state of order i in backward direction for any $i \le d$. Hence, without loss of generality, it is enough to consider weak states of order d in backward direction.

As mentioned before, any state can produce $(\theta+1)$-bit output without any state updates. On the other hand, a weak state of order d can produce $(d+\theta+1)$-bit output without knowledge about the key.

Now, the tradeoff attack works as follows. All weak internal states of order d (in backward direction) with their $(d+\theta+1)$-bit outputs are loaded in a table in the offline phase, sorted according to the outputs. The goal is then to recover a weak internal state in keystream generation and then check its correctness through Algorithm 2.1 or Algorithm 3.1. The online phase of the tradeoff attack is summarized in Algorithm 5.1 and we describe the offline phase next.

Algorithm 5.1 Online Phase of the Tradeoff Attack

```
for each $d+\theta+1$ bit output part $(z_{t-d-\theta_b}, \dots, z_{t+\theta_f})$ do
    Load all the precomputed states producing $(z_{t-d-\theta_b}, \dots, z_{t+\theta_f})$ into a set $\mathbb{S}$
    Run Algorithm 2.1 (with $\alpha_{ter}$, $\Pr_g$ and $\epsilon$) or Algorithm 3.1 (with $\alpha_{ter}$ and $\Pr_{thr}$) for the set $\mathbb{S}$ and the output $z_{t+1+\theta_f}, \dots, z_{t+\theta_f+\alpha_{ter}}$.
    // $\mathbb{S}$ can be divided into multiple sets and the algorithm can be run for each set in case there is not enough memory.
    if there is a correct state captured then
        Return the state as output and terminate
    end if
end for
```

5.1 Offline Phase

Let $\mathcal{T}$ be a KSG with Boolean KFF with an internal state size of s bits and 2^{s_d} weak internal states of order d. Find all the weak states of order d (in backward direction) and load them in a table with their $(d+\theta+1)$-bit outputs. The tables are sorted with respect to the outputs. The resulting memory complexity is 2^{s_d}.

The most generic way of executing the offline phase is just trying all the internal states and determining the weak states of order. Note that no information about the key is needed for preparing the table since producing the $(d+\theta+1)$-bit output is possible for the weak states without knowing the key. The complexity of preparing the table is 2^s attempts at worst, which consists of iterating over all the internal states where each attempt is at most $(d+\theta+1)$ clocks of the cipher. It is possible that one may exploit special properties of a concrete design of $\mathcal{T}$ to improve the time complexity of the offline phase as in the case of the cipher Sprout [13]. The low sampling resistance in Sprout is exploited and a system of specific nonlinear equations, whose solutions yield the weak states of the cipher, is solved. The offline phase in TMDT attack in [13] is around 2^{40} encryptions or less, which is much faster than 2^{80}, where the internal state size of Sprout is 80 bits.

Some states may be weak of larger order, making it possible to produce more than $(d+\theta+1)$ bits of output. These extra output bits can also be loaded for such states as a further improvement of the attack. However, we disregard this improvement to give the generic concept of the attack.

6 Complexity Analysis

Let $\mathcal{T}$ be a KSG with Boolean KFF with an internal state size of s bits, which is expected to be small for a lightweight stream cipher. We give the complexity numbers with respect to the weakness order d. One can take $d = 0$ and the number of weak states of order d to be $2^{s_d} = 2^s$ for the case when there are no weak states, including in Theorem 1. We assume that the weak states of $\mathcal{T}$ are uniformly distributed with respect to any given $(d+\theta+1)$-bit output[2]. That is, the probability that a $(d+\theta+1)$-bit output is produced by a weak state is 2^{s_d-s}. So, if we have 2^{s-s_d} output pieces, we expect one of them to be produced by a weak state and this weak state is captured. Hence, the data complexity is roughly $2^{s-s_d}+d+\theta$ bits (considering the $(d+\theta+1)$-bit pieces where each consecutive piece overlaps in $d+\theta$ bits).

For each output piece, we check if there is any weak state in the table producing it. These weak states are examined to see if they can further produce the output bits in forward direction consistently. There may be insufficient amount of output bits, particularly, for the pieces at the end. In this case, such states can be examined in backward direction or can simply be ignored.

There are roughly $2^{s_d-d-\theta-1}$ weak states that produce a given $(d+\theta+1)$-bit output $(z_{t-d-\theta_b},\ldots,z_{t+\theta_f})$. Among $\Pr_d \cdot 2^{s_d-d-\theta-1}$ of those states, we expect the feedback values to be determined from the $(t+\theta_f+1)$-th output. The remaining $(1-\Pr_d)\cdot 2^{s_d-d-\theta-1}$ weak states go into the check-and-guess procedure. Altogether, we expect $2^{s_d-d-\theta-1}$ states to survive after each clock. That is, we expect roughly the same number of remaining states through these determine or check-and-guess procedures.

An arbitrary wrong state produces mismatches at roughly half of the clocks it is iterated through. So, we expect a wrong state to be eliminated in roughly $2\alpha_{thr}$ clocks for a threshold α_{thr}. Therefore, testing all of them costs $\alpha_{thr}\cdot 2^{s_d-\theta-d}$ clocks on average for one specific output. There are 2^{s-s_d} output pieces to be examined. Hence, the total number of clockings during the internal state recovery phase is $\alpha_{thr}\cdot 2^{s-d-\theta}$. Theorem 1 suggests a lower bound for α_{ter}, the maximum number of clocks to be iterated, and α_{thr} to recover the right state and to eliminate all the wrong states through Algorithm 2.1.

Theorem 1. *Let $\Pr_g$ be the guess capacity of a given KSG with Boolean KFF having internal state size s and output capacity θ. We make the following list of assumptions.*

- *$\Pr_g > \frac{1}{2}$ and there is at least one weak state of order d occurring in the keystream generation.*
- *The probability that a given output is produced by a weak state is equal to the probability of choosing a weak state randomly among all the states*[2].
- *The probability that a mismatch occurs for a wrong state is one-half and the average probability of an occurrence of a mismatch for correct states at a given clock is $1-\Pr_g$ for a fixed key, independent of mismatch occurrence at other clocks.*

For a given $0<\epsilon<1$, if α_{ter} is greater than or equal to

$$\frac{1}{(2\Pr_g-1)^2}\left(\sqrt{-2\ln\epsilon}+\sqrt{2\ln 2\cdot(s-\theta-d-1)}\right)^2,$$

then the success rate of the attack in Algorithm 5.1 using Algorithm 2.1 is at least $1-\epsilon$ and the number of false alarms is less than one in total.

2. This assumption is required for determining how much data is needed. The attack still works without this assumption, and even better by prioritizing the analysis of the most probable weak states.

Proof. We want the miss event probability

$$\sum_{i=\alpha_{thr}+1}^{\alpha_{ter}}\binom{\alpha_{ter}}{i}(1-\Pr_g)^i\Pr_g^{(\alpha_{ter}-i)}$$

to be less than ϵ. Let us take

$$\alpha_{ter}\geq\frac{1}{(2\Pr_g-1)^2}\left(\sqrt{-2\ln\epsilon}+\sqrt{2\ln 2\cdot(s-\theta-d-1)}\right)^2.$$

Then, we can write

$$\frac{\ln 2\cdot(s-\theta-d-1)}{2}\leq\alpha_{ter}\left(\Pr_g-\frac{1}{2}-\sqrt{\frac{-\ln\epsilon}{2\alpha_{ter}}}\right)^2$$

since the value

$$\frac{1}{2\Pr_g-1}\left(\sqrt{-2\ln\epsilon}+\sqrt{2\ln 2\cdot(s-\theta-d-1)}\right)$$

is the largest root of the equation

$$\left(\Pr_g-\frac{1}{2}\right)^2x-2\left(\Pr_g-\frac{1}{2}\right)\left(\sqrt{\frac{-\ln\epsilon}{2}}\right)\sqrt{x}+\frac{-\ln\epsilon}{2}-\frac{\ln 2\cdot(s-\theta-d-1)}{2}$$

with the unknown parameter x. So, if we take $\epsilon_{ter}=\sqrt{\frac{-\ln\epsilon}{2\alpha_{ter}}}$, the miss event probability is bounded above by $\exp(-2\epsilon_{ter}^2\alpha_{ter})$ by Hoeffding's inequality where $\lceil x\rceil$ is the smallest integer greater than or equal to x. But, $\exp(-2\epsilon_{ter}^2\alpha_{ter})=\epsilon$. So, the success probability is at least $1-\epsilon$. On the other hand, $\alpha_{thr}=\lfloor\alpha_{ter}(1-\Pr_g+\epsilon_{ter})\rfloor$ where $\lfloor x\rfloor$ is the greatest integer less than or equal to x. Then, the false alarm probability for one state is given as

$$2^{-\alpha_{ter}}\sum_{i=0}^{\alpha_{thr}}\binom{\alpha_{ter}}{i}\leq\exp\left(-2\frac{(\alpha_{ter}/2-\alpha_{thr})^2}{\alpha_{ter}}\right)\tag{1}$$

by Hoeffding's inequality. Also, plugging

$$\alpha_{thr}\leq\alpha_{ter}(1-\Pr_g+\epsilon_{ter})$$

in Inequality 1, we obtain an upper bound for the false alarm probability for one wrong state as

$$\exp(-2\alpha_{ter}(\Pr_g-1/2-\epsilon_{ter})^2).$$

There are $2^{s-\theta-d-1}$ states in total to be examined. So, the average number of false alarms is bounded above by

$$2^{s-\theta-d-1-2\alpha_{ter}\log_2 e\left(\Pr_g-\frac{1}{2}-\sqrt{\frac{-\ln\epsilon}{2\alpha_{ter}}}\right)^2}$$

which is less than one when its exponent is a negative number. Recalling our assumption for the lower bound for α_{ter}, we get

$$\frac{\ln 2\cdot(s-\theta-d-1)}{2}\leq\alpha_{ter}\left(\Pr_g-\frac{1}{2}-\sqrt{\frac{-\ln\epsilon}{2\alpha_{ter}}}\right)^2.$$

Hence, the average number of false alarms is less than one. □

Theorem 1 gives us a lower bound for α_{ter} to promise a success rate of at least $1-\epsilon$. Indeed the lower bound is not sharp enough and can be further improved. We have

TABLE 2
Some examples of α_{ter} for different ϵ, $\Pr_g$, d, s and θ parameters.

ϵ	$\Pr_g$	d	s	θ	α_{ter}
0.10	0.75	10	80	2	555.40
0.10	0.55	40	80	2	8663.65
0.45	0.75	10	80	2	475.35
0.45	0.55	40	80	2	7099.13
0.10	0.90	40	80	2	135.37

conducted several experiments and have shown that it is not necessary to take α_{ter} as large as the value that Theorem 1 dictates. See Section 7 for a comparison of experimental results with the theoretical results. Besides, Theorem 1 suggests an average time complexity for Algorithm 2.1 since it proposes a bound for the value α_{ter}, and α_{thr} is set by α_{ter}.

Corollary 1. *With the assumptions as in Theorem 1, the average time complexity of Algorithm 5.1 using Algorithm 2.1 is bounded above by* $2^{s_d-\theta-d}\cdot(\alpha_{thr}+1)$ *for a required success rate of* $1-\epsilon$ *where* α_{thr} *is set in Algorithm 2.1.*

Proof. We should set α_{ter} to be larger than or equal to

$$\frac{1}{(2\Pr_g-1)^2}\left(\sqrt{-2\ln\epsilon}+\sqrt{2\ln 2\cdot(s-\theta-d-1)}\right)^2$$

to have a success rate of at least $1-\epsilon$ by Theorem 1. Then, α_{thr} is determined by α_{ter} as

$$\alpha_{thr}=\lfloor\alpha_{ter}(1-\Pr_g+\epsilon_{ter})\rfloor$$

where $\epsilon_{ter}=\sqrt{\frac{-\ln\epsilon}{2\alpha_{ter}}}$. Theorem 1 ensures a success rate of at least $1-\epsilon$ for this value of α_{thr}. Also, the probability that a wrong state produces a mismatch for each feedback is one-half. So, around $\alpha_{thr}+1$ mismatches are expected when a wrong state is clocked roughly $2(\alpha_{thr}+1)$ times. So, we expect a wrong state to be eliminated in $2(\alpha_{thr}+1)$ clocks on average as the number of mismatches exceeds the threshold. For a given output, we have $2^{s_d-\theta-d-1}$ wrong states and the time complexity of testing all of them through Algorithm 2.1 is on average $2^{s_d-\theta-d}\cdot(\alpha_{thr}+1)$ clocks. On the other hand, we have 2^{s-s_d} output pieces for the outer loop of Algorithm 5.1. Therefore, the average time complexity is $(\alpha_{thr}+1)\cdot 2^{s-d-\theta}$ clockings. □

Remark 2. *We assumed that a wrong state produces roughly* $\frac{\alpha_{ter}}{2}$ *mismatches. This is indeed the expected value if each iteration of a wrong state is also a wrong state. However, it may turn out that a wrong state becomes a correct state after several clocks. However, such cases are too rare and can be ignorable. We have done several experiments and have never witnessed such a case. This case can be considered as a technical detail of the attack that may be overcome straightforwardly.*

The success rate and the false alarm probabilities do not depend on the number of weak states. Some numerical examples of α_{ter} values are given in Table 2. These α_{ter} values are those in the worst-case, and we observe better numbers in the experiments.

7 Experiments

In this section, we provide the results of the extensive experiments to verify our assumptions and results from previous sections. Adapting the notation in Theorem 1, we have chosen $s=16$ and $\theta=d=0$ for all the experiments. So, we have no weak states in our examples. In order to set a $\Pr_{thr}$ value for Algorithm 3.1, we usually use $\Pr_{thr}=(1-\Pr_g)^{\alpha_{thr}}$ unless otherwise indicated. We use the term *deviation* (abbreviated as dev.) to indicate, for example, that if deviation is $1/8$, then guess capacity for a state is $\Pr_g - {}^1/_8$ with probability one-half, and it is $\Pr_g + {}^1/_8$ elsewhere.

We have taken long enough randomly generated keys (of a thousand bits) for the experiments so as to ensure that no additional bias is introduced due to the periodic reuse of the key bits. The way key is incorporated into the feedback is as follows. Randomly generate a binary array, arr, where the probability of encountering a zero in one half is $(\Pr_g - dev)$ and that in the other half is $(\Pr_g + dev)$. Then, an *index* is calculated at each clock for the state update where the MSB of the index is determined by a cell in the internal state, and the rest of the bits are determined by some (distinct) bits from the key. Finally, $arr[index]$ is output as the output of KFF. For example, if $\Pr_g=3/4$ and $dev=1/4$, then an array of length 8 could be $[0,0,1,1,0,0,0,0]$.

The experimental success rate is calculated by repeatedly (at least 1000 times) inputting a correct state along with its corresponding keystream sequence into the algorithm (the key and the initial state from which the keystream is generated are chosen randomly every time) and calculating the percentage of how many times it is output as a candidate. We only count the cases where the root iterates as the correct state at each clock.

To summarize, our experiments show that the experimental success rate with the parameters chosen as in Theorem 1 is always higher than the theoretical success rate. Moreover, we have seen that the theoretical success rate can be reached experimentally (while upper-bounding the average number of false alarms by 1) even when α_{thr} suggested by Theorem 1 is halved. This means that the time complexity can be decreased by a factor of about two. Further experiments show that Algorithm 3.1 performs better as the deviation increases whereas Algorithm 2.1 is not affected by the deviation, as expected. Because Algorithm 2.1 does not exploit the individual guess capacities of each state.

Focusing on Figure 2, we can easily see that both Algorithm 2.1 and Algorithm 3.1 always perform better than $1-\epsilon$ value as indicated by Theorem 1. Another important result from this figure is that the success rate of Algorithm 3.1 becomes much better as the deviation increases. We see that Algorithm 2.1 is not affected by the deviation and both of the algorithms behave similarly when there is no deviation.

Table 3 can be used to compare the time complexities of Algorithms 2.1 and Algorithm 3.1. It is easy to see that one can achieve higher success rates with a lower time complexity using Algorithm 3.1 rather than Algorithm 2.1. We note that although α_{thr} values are not given in Table 3, we have verified that average number of clocks iterated for a wrong state in Algorithm 2.1 is about $2\cdot(\alpha_{thr}+1)$ as expected (see Corollary 1). Furthermore, if we set $\epsilon=0.10$ in Theorem 1, then we get $\alpha_{thr}=299$. This means that one would need to run about 600 clocks on average for each state to achieve a 90% success rate according to Theorem 1. However, if we look at Table 3, one can achieve a greater

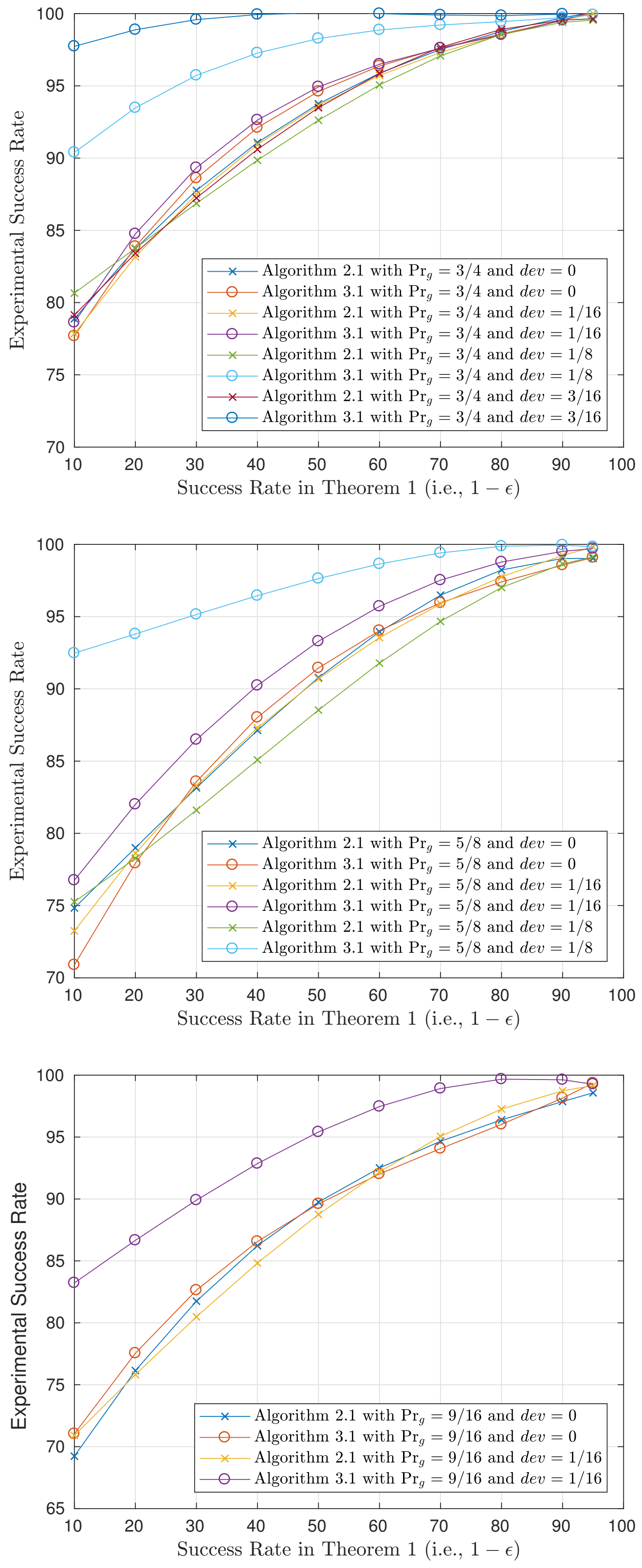

Fig. 2. Comparison of Algorithm 2.1 and Algorithm 3.1 for various ϵ values with different deviations. We have done a curve fitting as a degree-3 polynomial in the graphs and the rates are in percentages. Very few false alarms in total (almost always none for individual tests) were encountered.

TABLE 3
Average number of clocks (denoted as **Avg. Clocks**) iterated for a wrong state with respect to different α_{ter} and $dev.$ values. $\Pr_g = {}^5/_8$. **Exp. SR** denotes *experimental success rate*, and **Alg** and **Dev** are used as abbreviations for *Algorithm* and *Deviation*, respectively. α_{thr} are chosen to have no false alarm and $\Pr_{thr} = (1 - \Pr_g)^{\alpha_{thr}}$.

α_{ter}	**Dev.**	**Exp. SR (Alg-2.1, Alg-3.1)**	**Avg. Clocks (Alg-2.1, Alg-3.1)**
200	${}^1/_{16}$	(32.1, 33.1)	(143.9, 140.9)
200	${}^1/_8$	(30.7, 55.8)	(144.1, 135.0)
250	${}^1/_{16}$	(41.3, 49.0)	(185.9, 182.1)
250	${}^1/_8$	(42.1, 68.9)	(185.9, 175.1)
300	${}^1/_{16}$	(60.4, 66.5)	(230.0, 225.8)
300	${}^1/_8$	(60.4, 82.1)	(229.8, 216.3)
350	${}^1/_{16}$	(78.7, 82.9)	(276.1, 270.3)
350	${}^1/_8$	(75.5, 91.9)	(275.7, 259.0)
400	${}^1/_{16}$	(83.8, 87.8)	(319.4, 314.5)
400	${}^1/_8$	(84.0, 95.9)	(319.4, 300.9)
450	${}^1/_{16}$	(89.2, 93.6)	(363.3, 357.7)
450	${}^1/_8$	(90.4, 97.6)	(362.9, 342.5)
500	${}^1/_{16}$	(94.4, 96.8)	(409.6, 402.6)
500	${}^1/_8$	(95.3, 99.6)	(409.1, 386.1)

success rate by iterating about 260 clocks on average using Algorithm 3.1, which implies a speed-up of a factor greater than 2. We witnessed a similar situation with different parameters as well. Thus, it seems likely to reduce the time complexity indicated by Theorem 1 by half, especially as the deviation gets higher.

8 Application to Sprout

In this part, we place Sprout into the mathematical setting of a KSG with Boolean KFF and apply our generic algorithms to it. We give a quick overview of Sprout and refer the reader to [11] for further details.

8.1 Sprout as a KSG with Boolean KFF

Sprout [11] is one of the recent lightweight stream ciphers, whose design rationale is highly affected by Grain family [19]. An 80-bit key is used to update the internal state which consists of a 40-bit LFSR and a 40-bit NFSR. A 70-bit IV is used to initialize registers. The details of the update functions and the output function are not important for the application of Algorithm 2.1 or Algorithm 3.1 since we do not exploit their internal structures. Our attacks are generic and we only investigate the guess capacity, the output capacity and the determine probabilities of Sprout to introduce our attacks.

Sprout is a KSG with Boolean KFF since it has two feedback bits for each clock as key-dependent one for the N register (NFSR) and key-independent one for the L register (LFSR). L register is updated by itself and key is not incorporated into L.

What is important for our purposes is that the *round key function* outputs a bit as $k^*_t = k_{t \bmod 80} \cdot \delta_t$ where δ_t is a bit calculated as an XOR sum of some bits from the registers. Now, the weak states of order d for Sprout are those where δ_t vanishes for d consecutive clocks. This means that the key bits are ignored starting at clock $t-9$ until $(t+d-10)$-th clock. As a result, the number of weak states of order d is around 2^{80-d}.

TABLE 4
Experimental results for a simulation of Sprout for $\alpha_{ter} = 101$. **CSR:** Calculated Success Rate using the formula $\sum_{i=0}^{\alpha_{thr}} (1 - \mathrm{Pr}_g)^i \, \mathrm{Pr}_g^{\alpha_{ter}-i}$, **ANC:** Averege Number of Clockings for examining one state. **Alg:** Algorithm.

CSR	ANC (Alg-2.1)	ANC (Alg-3.1)	α_{thr} (Alg-2.1)
99.5%	121.7	4	60
77.5%	57.7	4	28

Let us denote $N_t := (n_0^t, n_1^t, \ldots, n_{39}^t)$ as the state value of the nonlinear register NFSR at clock t. $\mathrm{Pr}_f = 1/4$ and $\mathrm{Pr}_b = 1$ for Sprout since n_{38}^t appears in a term of degree 3 and n_1^t appears linearly in the output function whereas n_0^t and n_{39}^t do not affect the output at all. This also implies that $\theta_b = \theta_f = 1$. Therefore, for any given state of Sprout, it is possible to produce 3 bits of output without any state updates. More interestingly, the guess capacity is one for the states where $\delta_t = 0$ and one-half for the states where $\delta_t = 1$. Hence, the average guess capacity $\mathrm{Pr}_g = 0.75$.

8.2 Generic Attacks on Sprout

In this section, we illustrate the generic attacks through Algorithm 2.1 and Algorithm 3.1 on Sprout. Recall that the weak states of order d are those where $\delta_t = 0$ for d consecutive clocks and $d + 3$ bits of output can be produced.

One mismatch is enough for a wrong state with $\mathrm{Pr}_g(S)_f = 1$ to be eliminated by Algorithm 3.1 since the probability $\mathrm{Pr}_{Ch}(S)$ drops to zero immediately in that case. We expect this to happen in 4 clocks on average since half of the states have a guess capacity one and the probability of entering into a check-and-guess procedure is one-half. Hence, the average number of clocks through Algorithm 3.1 for a wrong state of Sprout is 4.

Taking $\epsilon = 0.01$, if we load the weak states of order d, then $\alpha_{ter} > 4(\sqrt{2 \cdot \ln 2}\sqrt{77 - d} + 3.03)^2$ by Theorem 1. In particular, $\alpha_{ter} = 414$ and $\alpha_{thr} = 134$ for $d = 40$. So, the average time complexity will be at most $2 \cdot 134 \cdot 2^{37} \approx 2^{45}$ Sprout clockings for recovering a correct internal state with Algorithm 2.1. Thus, Algorithm 2.1 is at most 67 times slower than the attack in [13] and at least 7.6 times faster than the attack in [14] according to Theorem 1.

We have simulated Sprout with a register where the guess capacity is one-half for half of the states and one for the remaining half. The results verify our conclusions from previous sections. Algorithm 3.1 can eliminate a wrong state in approximately 4 clocks, meaning that Algorithm 3.1 is as fast as the best attack given in [13]. Similarly, a wrong state can be eliminated in roughly 121.7 clocks by Algorithm 2.1 as given in Table 4, which is roughly 31 times slower when we consider the success rate as 99%. So, Algorithm 3.1 is roughly 2^9 times and Algorithm 2.1 is roughly 16,5 times faster than the attack in [14].

It is worth emphasizing that both Algorithm 2.1 and Algorithm 3.1 are generic attacks and can successfully be applied not only to Sprout but also to any KSG with Boolean KFF having a guess capacity greater than one-half. Still, the best attack in terms of computational complexity on Sprout is achieved with Algorithm 3.1. Furthermore, the attacks in [13], [14] makes use of the fact that $\mathrm{Pr}_d = 1$ in backward direction. However, we have shown that the complexities of our generic internal state recovery attacks do not depend on the value Pr_d. So, these attacks would still work even though Sprout would be fixed so that Pr_d is strictly less than one whereas the attacks in [13], [14] would not work.

9 Comparison of Tradeoff Attacks

After [20] appeared in IACR's e-print and presented at SAC 2015 [13], Zhang and Gong described another TMD tradeoff attack on Sprout at Asiacrypt 2015 [14]. Both attacks are based on checking if one of the weak states of order $d + 3$ is used in keystream generation where the weak states are determined and loaded in tables in the precomputation. However, the Asiacrypt paper imposes additional conditions on the weak states. In this section, we give a detailed comparison of the TMD tradeoff attack in [13] (Esgin-Kara attack), the one in [14] (Zhang-Gong attack).

Let $\mathcal{T}$, $\mathcal{T}_P$, $\mathcal{M}$ and $\mathcal{D}$ denote the online time, the precomputation time, the memory and the data complexities of an attack, respectively. The comparison is done with respect to the attack given in Section 4 of [14], which details the attack on Sprout.

In [14], two parameters x and y are defined as the number of forward and backward clockings in the attack, respectively. It is easy to see that these parameters satisfy the relation $d = x + y$. Thus, we use d instead of $x + y$. First, we provide the time/data/memory workloads of Zhang-Gong attack (for $d \geq 30$) as given in [14]:

- $\mathcal{T} = 2^{70.66-d}$ encryptions $+ 2^{d+6}$ TLs,
- $\mathcal{T}_P = 2^{74-d}$,
- $\mathcal{M} = Count(|C'|) \cdot 2^{71-d} \cdot \left[61 + \frac{Count(d)}{Count(|C'|)}(2d - 58)\right]$,
- $\mathcal{D} = 2^{d+9}$.

The special states loaded in the tables and then examined are the middle states in [14]. That is, the states are stored for time t and δ_i being equal to 0 for $i = t - y, \ldots, t + x - 1$ is assumed. That's why checking each state during the key recovery attack in backward direction does not end in approximately 4 clocks, but rather does in $y + 4$ clocks. So, either the online time complexity increases by a factor of around 2^2 for $y \geq 12$ or there is an additional time complexity in the precomputation in order to wind back the registers to their corresponding states at time $t - y$. The latter process can be considered more efficient as it is run in the offline phase. Then, an additional workload of $y \cdot 2^6 \cdot 2^{71-d} \cdot 2^{-8.33} = y \cdot 2^{68.67-d}$ encryptions[3] must be added to the precomputation.

Furthermore, the memory complexity of Zhang-Gong attack is disputable. From [14], it seems that $\mathcal{M}$ is decreased by a factor less than 4 for the same $\mathcal{D}$ in comparison to Esgin-Kara attack. The following is quoted from [14]:

"As for the memory, we need $Count(|C'|) \times 2^{13}$ tables $\mathrm{T}_{C',i}$, each having 2^{58-x-y} rows in the first column to store 61-bit "special" states ... and $2^{58-x-y} \times \frac{Count(|C|)}{Count(|C'|)}$ rows in the second column to store the corresponding output bits."

The internal states are stored in the first column and the second column contains sub-rows each storing output bits with respect to different counter arrays involved in output generation. Therefore, it is not possible to store the

3. One clock of Sprout is equivalent to $2^{-8.33}$ encryptions of the cipher as shown in [13].

tables sorted with respect to the outputs. So, the memory workload decreases by a factor of about four but at the cost of making (linear-time) searches in unsorted tables. This will increase the workload of table lookups enormously, making the attack extremely inefficient. This fact is not mentioned in [14].

If the tables are sorted, then the total number of rows in all the tables become equal in both of the attacks, which implies that the memory complexities are equivalent in both attacks. Table 1 summarizes the actual complexities, where we naturally assume that the tables are sorted.

One can consider Zhang-Gong attack as a special case of Esgin-Kara attack: Impose any additional 9-bit condition on the states to be collected such that a 3-bit part of this condition is observable by the output. Moreover, do not use a 3-bit part of the condition when solving the system of equations. The same complexities as those given in Table 1 are obtained in this case.

To sum up, the additional artificial conditions imposed in [14] render the attack cumbersome. Indeed, it can be seen from Table 1 that Zhang-Gong attack is roughly 2^9 times slower than Esgin-Kara attack with the same amount of data or requires 2^9 times more data with the same time complexity. The same result can be seen from Table 3 of Zhang and Gong's work [14].

10 Conclusion and Discussion

We have studied the security of KSGs with KUF in a mathematical framework. Several open problems await further research. The bound for α_{ter} given in Theorem 1 for Algorithm 2.1 is not sharp. A theoretical statement proving that Algorithm 3.1 is faster than Algorithm 2.1 is also another open issue. We do not exploit the probability of determine and may be further improved by leveraging particular values. The natural question is whether it is possible to introduce a new attack that works even if the guess capacity is equal to one-half. One concrete example to be studied for such a case is the recent cipher Plantlet [17].

Acknowledgments

We thank the anonymous reviewers for their helpful comments and suggestions, which greatly improved the manuscript. The first author's work is based upon work from COST Action CRYPTACUS, supported by COST (European Cooperation in Science and Technology).

References

[1] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe, "PRESENT: An ultra-lightweight block cipher," in *CHES 2007*, ser. LNCS. Springer Berlin Heidelberg, 2007, vol. 4727, pp. 450–466.

[2] C. De Cannière, O. Dunkelman, and M. Knežević, "KATAN and KTANTAN a family of small and efficient hardware-oriented block ciphers," in *CHES 2009*, ser. LNCS. Springer Berlin Heidelberg, 2009, vol. 5747, pp. 272–288.

[3] J. Guo, T. Peyrin, A. Poschmann, and M. J. B. Robshaw, "The LED block cipher," in *CHES 2011*, ser. LNCS, vol. 6917. Springer, 2011, pp. 326–341.

[4] K. Shibutani, T. Isobe, H. Hiwatari, A. Mitsuda, T. Akishita, and T. Shirai, "Piccolo: An ultra-lightweight blockcipher," in *CHES 2011*, ser. LNCS, vol. 6917. Springer, 2011, pp. 342–357.

[5] S. Banik, A. Bogdanov, T. Isobe, K. Shibutani, H. Hiwatari, T. Akishita, and F. Regazzoni, "Midori: A block cipher for low energy," in *ASIACRYPT*. Springer, 2014, pp. 411–436.

[6] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, "The SIMON and SPECK families of lightweight block ciphers," *IACR Cryptology ePrint Archive*, vol. 2013, p. 404, 2013.

[7] G. Yang, B. Zhu, V. Suder, M. D. Aagaard, and G. Gong, "The Simeck family of lightweight block ciphers," in *CHES 2015*, ser. LNCS, vol. 9293. Springer, 2015, pp. 307–329.

[8] M. E. Hellman, "A cryptanalytic time-memory trade-off," *IEEE Transactions on Information Theory*, vol. 26, no. 4, pp. 401–406, 1980.

[9] S. Babbage, "Improved exhaustive search attacks on stream ciphers," Security and Detection 1995, European Convention IET, pp. 161–166, 1995.

[10] J. D. Golić, "Cryptanalysis of alleged A5 stream cipher," in *EUROCRYPT '97*, ser. LNCS, vol. 1233. Springer, 1997, pp. 239–255.

[11] F. Armknecht and V. Mikhalev, "On lightweight stream ciphers with shorter internal states," in *Fast Software Encryption*, ser. LNCS. Springer Berlin Heidelberg, 2015, vol. 9054, pp. 451–470.

[12] V. Lallemand and M. Naya-Plasencia, "Cryptanalysis of full Sprout," in *Advances in Cryptology – CRYPTO 2015*, ser. LNCS. Springer Berlin Heidelberg, 2015, vol. 9215, pp. 663–682.

[13] M. F. Esgin and O. Kara, "Practical cryptanalysis of full Sprout with TMD tradeoff attacks," in *Selected Areas in Cryptography - SAC 2015*, 2015, pp. 67–85.

[14] B. Zhang and X. Gong, "Another tradeoff attack on Sprout-like stream ciphers," in *ASIACRYPT 2015*, ser. LNCS, vol. 9453. Springer, 2015, pp. 561–585.

[15] S. Maitra, S. Sarkar, A. Baksi, and P. Dey, "Key recovery from state information of Sprout: Application to cryptanalysis and fault attack," Cryptology ePrint Archive, Report 2015/236, 2015.

[16] S. Banik, "Some results on Sprout," in *INDOCRYPT 2015*, ser. LNCS, vol. 9462. Springer, 2015, pp. 124–139.

[17] V. Mikhalev, F. Armknecht, and C. Müller, "On ciphers that continuously access the non-volatile key," *IACR Transactions on Symmetric Cryptology*, vol. 2016, no. 2, pp. 52–79, 2017.

[18] M. Hamann, M. Krause, and W. Meier, "Lizard a lightweight stream cipher for power-constrained devices," *IACR Transactions on Symmetric Cryptology*, vol. 2017, no. 1, 2017.

[19] M. Hell, T. Johansson, A. Maximov, and W. Meier, "The Grain family of stream ciphers," in *New Stream Cipher Designs*, ser. LNCS. Springer Berlin Heidelberg, 2008, vol. 4986, pp. 179–190.

[20] M. F. Esgin and O. Kara, "Practical cryptanalysis of full Sprout with TMD tradeoff attacks," Cryptology ePrint Archive, Report 2015/289, 2015, http://eprint.iacr.org/.

Orhun Kara received his Ph.D. with a thesis about code construction on modular curves from the Department of Mathematics, Bilkent University in 2003. He has been with TUBITAK National Research Institute of Electronics and Cryptology since 2000. His current research interest covers mostly the design and analysis of symmetric ciphers. He gives graduate lectures related to cryptology in several universities such as METU, Istanbul Şehir University and Gebze Technical University.

Muhammed F. Esgin received his B.Sc. in Mathematics from Boğaziçi University and his M.Sc. in Cybersecurity Engineering from Istanbul Şehir University. He is currently doing his Ph.D. in Faculty of Information Technology at Monash University. His main focus is on mathematical aspects of information security, more specifically mathematical cryptography and cryptanalysis.