Radboud Universiteit

Runaway Keystream Generators and their Parallelizability and G&D-Resistance

BSc. Maaike van Leuken

Supervisors:
Prof. Dr. Joan Daemen
Dr. Silvia Mella

Second Assessor:
Dr. Shahram Rasoolzadeh

Master Thesis Cyber Security
July 15, 2021

Abstract

A stream cipher is a symmetric key-scheme that turns a short key in combination with a diversifier into a arbitrarily long keystream. The keystream is used for stream encryption, i.e. the ciphertext is obtained by combining the plaintext with the keystream. In this thesis, we investigate the security and efficiency of keystream generators constructed from a b-bit permutation g, with b in the range 32 to 128, and some additions. We will call keystream generators built with this construction *runaway generators*. In this construction the state consists of n blocks of each b bits. It operates as a filtered non-linear feedback shift register. To update the state, all state blocks are shifted right and the feedback function computes the leftmost block from the state using some additions and one call to the permutation g. The construction is generic and the computation is defined by two vectors, c and d. Every cycle, the generator outputs a b-bit keystream block that is the sum of state blocks determined by the vector e.

We assess the performance of the runaway generators by looking at how many state elements can be computed in parallel. We show that parallelizability is completely dependent of the vectors c and d. We also analyze their security in terms of how many bits of guess-and-determine (G&D) resistance different runaway generators offer, where we use a generic block function. We show that finding the best G&D-attack for a runaway generator can be automated, which allows us to test the security of many specific runaway generators. From this research, we can conclude that it is possible to make a specific generator that achieves the upper bound of $(n-1)b$ bits of G&D-resistance. We make some recommendations that increase the probability of obtaining a high G&D-resistance and parallelizability. We finally use our tool to assess the G&D-resistance when a modified version of the round function of `SKINNY` is used in the update function.

Table of Contents

Chapter 1

Introduction

Keystream generators (KSGs) are used to transform a small key into an arbitrary long one. The resulting keystream is then used to encrypt some message. Stream ciphers belong to the symmetric key cipher realm and allow for stream encryption, where each plaintext bit is combined with a keystream bit to obtain a ciphertext bit. The diversifier ensures that keeping the same key for the stream cipher will not result in equal keystreams. Block ciphers in OFB or counter mode are stream ciphers where a part of the state stays fixed, namely the key. We call our KSGs "runaway generators" because the full state evolves. There is no part of the state that keeps its value as the keystream is being generated. The diversifier and key of a stream cipher can be combined into an initial state for the runaway generator. In this thesis, we assume that we are given an initial state, which is secret and uniformly distributed. Then we can expand the initial state with the generator, resulting in an arbitrary-length keystream.

There are various KSGs, such as linear feedback shift registers (LFSRs). However, these have a linear update function which allows using linear algebra to construct the initial (secret) state when having enough keystream bits. More interesting stream ciphers are irregularly clocked LFSRs and non-linear feedback shift registers (NLFSRs). Instances of the first class of LFSRs are broken, such as GSM A5/1 [Barkan and Biham, 2006] [Shah and Mahalanobis, 2012]. Examples of the latter include Trivium [Cannière, 2006] and Snow [Ekdahl and Johansson, 2003]. In this thesis we aim to add to the range of KSGs, namely the simple, lightweight and plausibly secure keystream generators. We will investigate the runaway generators, which promise a good trade-off between efficiency and cryptographic security. Bit-oriented stream ciphers, such as LFSRs, are significantly slower in software implementations than word-oriented stream ciphers like Snow [Ekdahl and Johansson, 2003]. Hence, we

define a runaway generator as word-oriented stream cipher. Whether a specific runaway generator is hardware- or software-oriented depends on the block function. What runaway generators look like will be discussed in Chapter 2. We explain there what we think is low-cost enough. In Chapter 3, we describe the performance of runaway generators as *parallelizability*. We assess the cryptographic security of the runaway generators as how well they resist guess-and-determine attacks, namely the *guess-and-determine resistance*, in Chapter 4. We do that, with use of self-made tool, which also allows us to assess the guess-and-determine resistance of runaway generators with a specific block function, an adaptation of the round function of `SKINNY`. In Chapter 6, we show that runaway generators look promising in both security and performance, compared to the state-of-the-art stream ciphers.

Chapter 2

Preliminaries

A keystream generator is defined as a triplet (S, Υ, f), with internal state S, update function Υ and output filter f. Figure 2.1 shows an iterative KSG. It is also synchronous, as the keystream is computed independent of the plaintext and ciphertext.

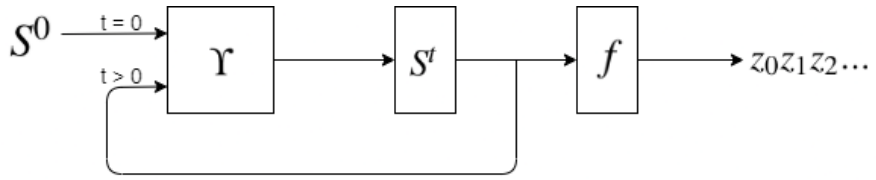

Figure 2.1: Graphical display of the keystream generator. The initial state is denoted as S^0, this is the input of the KSG. Then all other states S^t for $t > 0$ can be computed from this, using the update function Υ. Using the output filter f and S^t, the keystream block at time t is computed.

As described in Chapter 1, a KSG can be used as part of a stream cipher. A stream cipher has as input a key K and diversifier D and creates an arbitrary-length keystream. The key and diversifier will serve as the initial state. This is shown below in Figure 2.2. The resulting keystream $z_0 z_1 z_2 \ldots$ can be used for stream encryption.

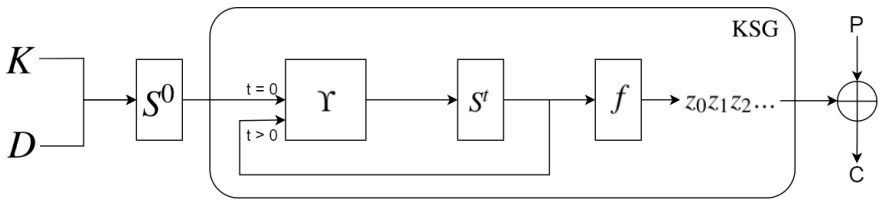

Figure 2.2: A stream cipher takes as input a key K and diversifier D. These are combined into the initial state S^0 of our KSG. The KSG then produces a keystream, which can be used as key in stream encryption. The plaintext P is bitwise XOR'ed with the keystream, to obtain the ciphertext C.

2.1 Runaway Generator

We are interested in a type of KSG with specific properties, namely the runaway generators, which we will now describe. The internal state $S^t = \{a_t, \ldots, a_{t+n-1}\}$ at time t for $t \geq 0$, consists of n elements of some finite field, in this thesis, the elements are in $GF(2)^b$. We assume we have a uniform, secret initial state $S^0 = \{a_0, \ldots, a_{n-1}\}$, consisting of n blocks of b bits. We can compute the state S^{t+1} at time $t+1$ as a function of S^t:

$$S^{t+1} = \Upsilon(S^t) = \Upsilon(a_t, a_{t+1}, \ldots, a_{t+n-1}) = (a_{t+1}, a_{t+2}, \ldots, a_{t+n-1}, a_{t+n}) \tag{2.1}$$

$$= (a_{t+1}, a_{t+2}, \ldots, a_{t+n-1}, U(a_t, a_{t+1}, \ldots, a_{t+n-1})) \tag{2.2}$$

In words, the state S^{t+1} consists of a linear combination of the state blocks in S^t shifted by 1. However, we then get a block a_{t+n} which can be expressed in terms of $a_t, a_{t+1}, \ldots a_{t+n-1}$ using the feedback function U. We call the list of all state elements a_t the "inner sequence". We will use both state block and inner sequence element to refer to a_t. So every state consists of n consecutive elements of the inner sequence. The feedback function U is a linear combination of state blocks added to the block function g applied to a linear combination of state blocks:

$$a_t = U(S^{t-n}) = U(a_{t-n}, a_{t-n+1}, \ldots, a_{t-1}) = \sum_{i=1}^{n} (c_i a_{t-i}) + g\Big(\sum_{j=1}^{n} d_j a_{t-j}\Big) \tag{2.3}$$

Vectors c and d determine which state blocks are present in the computation for the inner sequence element a_t. Note that vectors c and d start from index 1 and are in reversed order with respect to the index of the inner sequence element. We aim to develop a low-cost KSG, namely one where the feedback function U only has one application of the *block function* g and one or a few additions. For binary vectors c and d, we can see this as $HW(c) + HW(d)$ being small. The block function call is much more expensive than an XOR, so we want to limit ourselves to only one call to the block function in the feedback function. The block function can be a function that is

- a permutation. The output of a permutation is uniformly distributed, so there is no bias towards a specific value. We want this for our block function, otherwise imbalance accumulates every time we apply g. Any permutation is invertible, as every permutation is a bijective mapping which allows to invert the operation.

- non-linear. A non-linear block function makes our update function also non-linear. In general, no cryptographic system should be completely linear. This allows the attacker to express every bit of output as a linear function of inputs. With more equations than unknowns, the attacker is able to solve the system using linear algebra. We will also see a specific case in Section 4.2 where linearity of the block function decreases the G&D-resistance.
- low-cost. We want the block function to be low-cost, such that the runaway generator is efficient.

We want U to be invertible. If U is not invertible, it is not a bijective mapping, then we lose entropy when we apply the function. This will increase imbalance in our keystream sequence. An attacker could then exploit statistical differences with the Random Oracle ($\mathscr{RO}$) to distinguish it from a $\mathscr{RO}$ easier, i.e. using a smaller amount of keystream blocks. We need the keystream to be reasonably uniformly distributed such that we can securely use it for stream encryption. Invertibility of U requires that the block function g is invertible and that Equation 2.3 can be rearranged into the form $a_{t-n} = V(a_t, a_{t-1}, \ldots, a_{t-n-1})$, where the function V defines the right-hand side of the equation solved for a_{t-n}. This can only be done if either $c_n = 1$, then

$$a_{t-n} = a_t - \sum_{i=1}^{n-1} (c_i a_{t-i}) - g\Big(\sum_{j=1}^{n-1} d_j a_{t-j}\Big) \tag{2.4}$$

or $d_n = 1$, then

$$a_{t-n} = g^{-1}\left(a_t - \sum_{i=1}^{n-1} (c_i a_{t-i}) - g\Big(\sum_{j=1}^{n-1} d_j a_{t-j}\Big)\right) \tag{2.5}$$

This last part is possible due to invertibility of the block function. Note that if U is invertible, we are able to solve for any inner sequence element in S^t, not just for a_{t-n} as shown here.

The output filter f computes keystream block z_t based on S^t as:

$$z_t = f(S^t) = f(a_t, a_{t+1}, \ldots, a_{t+n-1}) = \sum_{i=0}^{n-1} e_i a_{t+i} \tag{2.6}$$

Where $e_i \in \{0,1\}$ indicates whether a_i is present in the equation for z_t. Note here that e starts at index 0. z_t is just a linear combination of $HW(e)$ blocks. We want the output filter to be as low-cost as possible, so we only allow a few additions. The idea behind this is that all

computation done in the output filter is only used once, namely to compute one keystream block. Because a_{t+1} is dependent of the inner sequence elements in state S^t, computations performed in the feedback function U are used in all future inner sequence elements. The inner sequence elements are also used to compute the keystream blocks. This is why we perform the block function call in the feedback function, and not in the output filter. This also means that we have a non-linear feedback function, but a linear output filter.

The vectors c, d and e of length n define the runaway generator, where we keep the block function generic. In a concrete implementation of the runaway generator, the state can be stored in n memory cells. Each of these cells contain an element in $GF(2)^b$. When we apply the update function Υ, we shift the state S^t to S^{t+1}. This can be implemented in a shift register. The new highest inner sequence element a_t is computed using the function U on S^t. As the block function works on blocks of b bits, it can be stored in less amount of memory than the internal state.

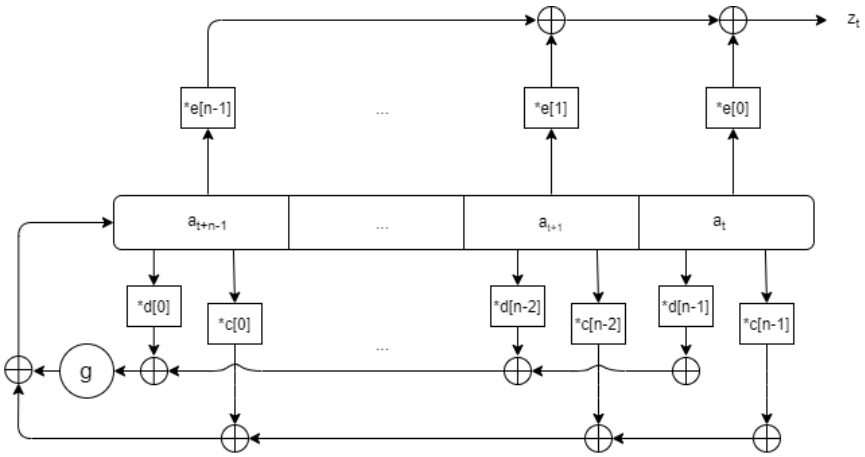

Figure 2.3: The generic runaway generator. The shift register contains the inner sequence elements $S^t = (a_t, \ldots a_{t+n-1})$ at time t. To obtain state S^{t+1}, the register is shifted to the right. The leftmost element will then become a_{t+n}, which is computed using Equation 2.3.

Example 1. *Consider the case with $n = 2$, so $S^0 = (a_0, a_1)$, and with $c = (0,1)$, $d = (1,0)$ and $e = (1,1)$. We can write this as the recurrence relation $a_t = a_{t-2} + g(a_{t-1})$. Then the keystream sequence is defined as $z_t = a_t + a_{t+1}$. Table 2.1 shows the inner sequence elements expressed as a function of the initial state blocks a_0 and a_1.*

t	a_t
0	a_0
1	a_1
2	$a_0 + g(a_1)$
3	$a_1 + g(a_2) = a_1 + g(a_0 + g(a_1))$
4	$a_2 + g(a_3) = a_0 + g(a_1) + g(a_1 + g(a_0 + g(a_1)))$

Table 2.1: The inner sequence for the runaway generator defined by $c = (0,1)$, $d = (1,0)$ and $e = (1,1)$. The inner sequence elements are expressed in terms of the initial state $S^0 = (a_0, a_1)$.

2.2 Attacker Model

The goal of the attacker is to distinguish our keystream from a random sequence. The runaway generator is initialized with a uniformly random state. The attacker makes one query giving a desired output length l to the runaway generator/$\mathscr{RO}$. She then receives the keystream sequence Z of length l. This is the data complexity. The computational complexity is the amount of queries to the block function g and the output filter f. The attacker can query g and f as these are public due to Kerckhoffs's principle. We also assume that the attacker can perform any other computation. Figure 2.4 displays the situation.

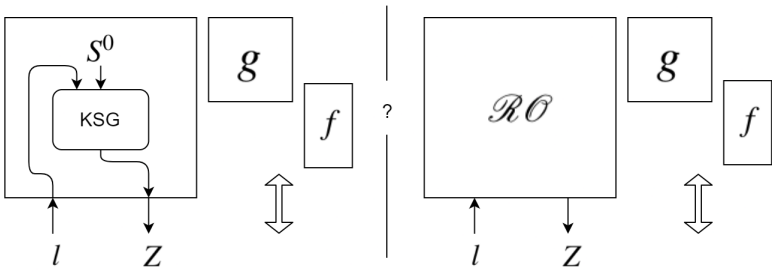

Figure 2.4: Graphical display of the attacker model.

A $\mathscr{RO}$ has the property that its output, the keystream sequence, is perfectly random. This means that if the output of the runaway generator is not uniformly distributed, it can be distinguished from the $\mathscr{RO}$. The attacker can perform a distinguishing attack by looking at

the statistical differences between the keystream generated by a KSG and a perfectly random keystream. For any real-life keystream generator, it is the case that the keystream is not perfectly uniformly distributed. However, the KSG provides enough security against distinguishing attacks if the attacker needs more than 2^{64} keystream bits to distinguish the KSG from the $\mathscr{RO}$.

Another way of distinguishing can be to recover n consecutive elements of the inner sequence. If the attacker can do this, she can generate all other states S^t, for any time t. Then she can generate the entire keystream again to check for consistency. Typically, if the attacker has knowledge of some S^t, she can also recover the initial state, if the update function is invertible. Since runaway generators have an invertible update function, we can say that the attack is successful if the attacker can reconstruct any state. The attacker is then able to distinguish the runaway generator from $\mathscr{RO}$. Recovering a state can be done using a guess-and-determine attack. We will discuss this in Chapter 4.

Chapter 3

Parallelizability

High-end CPUs support parallelizing operations, i.e. performing multiple operations at the same time, using pipelined or SIMD instructions. To exploit that, it is preferable to support parallelism in the computation of the inner sequence of the runaway generators. The requirement for two state blocks a_i and a_j to be able to be computed in parallel is that a_i is not dependent of a_j and vice versa. Actually, for $i < j$, we can say that a_j should not be dependent of a_i to be computed in parallel with a_i. Since a_i is computed before a_j, a_i can never be dependent of a_j.

Definition 1 (Parallelizability)**.** *We define the inner sequence to be* x-parallelizable *if* x *inner sequence blocks can be computed in parallel.*

Inner sequence element a_t can never be dependent of a_t, this would lead to self-recursion. Parallelizability is thus always at least 1, so there is no parallelizability, namely we can always compute one inner sequence element. Parallelism in this setting is highly dependent of what state blocks are used in the function U. Recall that U is defined by vectors c and d. These vectors fully determine the parallelizability for runaway generators. Let's first look at an example.

Example 2. *Consider the case with* $n = 4$*,* $c = (0,0,0,1)$ *and* $d = (0,1,0,0)$*. Recall from Chapter 2 that vectors c and d are in reversed order with respect to the state index and that the*

vectors start at index 1. Since parallelizability only concerns state blocks, we do not have to define an output filter. See the following table:

t	a_t
0	a_0
1	a_1
2	a_2
3	a_3
4	$a_0 + g(a_2)$
5	$a_1 + g(a_3)$
6	$a_2 + g(a_4) = a_2 + g(a_0 + g(a_2))$
7	$a_3 + g(a_5) = a_3 + g(a_1 + g(a_3))$

We see that a_6 dependent is of a_4. So a_4 needs to be computed before a_6 can be computed. a_5 is not dependent on a_4, so a_4 and a_5 can be computed in parallel. This holds for all pairs of elements of the inner sequence: a_t and a_{t+1} can be computed in parallel.

Definition 2 (Recursion Gap)**.** *For the relation for $a_t = \sum_{i=1}^{n}(c_i a_{t-i}) + g\left(\sum_{j=1}^{n} d_j a_{t-j}\right)$ from Equation 2.3, we define the* recursion gap *as the distance from the highest indexed state block present in the expression for a_t to t. The recursion gap rg is the lowest index in vectors c and d to be non-zero. We denote by k the lowest index such that all c_x with $x < k$ are zero and by l the lowest index such that all d_x with $x < l$ are zero, for $1 < k, l \leq n$. The recursion gap is then $rg = \min\{k,l\}$. The highest state block present in the expression for a_t has index a_{t-rg}.*

Theorem 1 (Parallelizability of Runaway Generator)**.** *The parallelizability of the inner sequence is rg blocks, i.e. the inner sequence is rg-parallelizable.*

Proof. The inner sequence at time t as described in Equation 2.3 can be rewritten as $a_t = \sum_{i=rg}^{n}(c_i a_{t-i}) + g\left(\sum_{j=rg}^{n} d_j a_{t-j}\right)$. In words, the inner sequence element at time t is only dependent of state blocks $a_{t-n}, \ldots, a_{t-rg}$ and not of the elements $a_{t-rg+1}, \ldots, a_{t-1}$. The latter consists of $t-1-(t-rg+1)+1 = rg-1$ state blocks. So a_t can be computed in parallel with $rg-1$ elements. Then in total rg blocks can be computed in parallel. □

Let's look back to Example 2. From the vectors $c = (0,0,0,1)$ and $d = (0,1,0,0)$ we see that $rg = \min\{4,2\} = 2$. Recall that the indices of c and d start at 1. From Theorem 1 we know that this runaway generator is 2-parallelizable. If we are not satisfied with a parallelizability of 2 blocks, we could take the runaway generator with $d = (0,0,1,0)$. Now we have that the recursion gap $rg = \min\{4,3\} = 3$, such that this new runaway generator is 3-parallelizable.

Chapter 4

Guess-and-Determine Attack Resistance

If the attacker knows the initial state, or any n consecutive inner sequence elements, the attacker can generate all keystream blocks. This is undesirable, since it allows the attacker to distinguish the runaway generator from a Random Oracle. To obtain the initial state, the attacker could perform an exhaustive search on the initial state. This can be done by guessing a value for S^0, generating the keystream that follows from that guess and compare it with the given keystream. We saw in Section 2.1 that the initial state consists of n blocks, each of length b. The security strength is therefore upper bounded to nb bits.

However, the runaway generator is based on fixed functions U and f, which are defined by the three vectors c, d and e. Note that to completely define the generator also a concrete block function g has to be specified. The attacker can use the knowledge of the algorithm to perform a more efficient attack than brute forcing S^0.

Definition 3 (Guess-and-determine attack in KSG setting)**.** *In a guess-and-determine (G&D) attack, an attacker guesses some inner sequence elements. Combined with the knowledge of the keystream elements, other elements of the inner sequence can be determined. If the attacker is able to determine n consecutive inner sequence blocks, she can generate an output sequence and compare it with the observed keystream sequence.*

Definition 4 (G&D-resistance)**.** *The G&D-resistance of a KSG is the number of bits an attacker has to guess in order to recover n consecutive inner sequence elements. If the attacker has to guess x inner sequence elements of b bits to successfully recover n consecutive inner sequence elements, the G&D-resistance is $x \cdot b$ bits or equivalently, x blocks.*

The attack for recovering any n consecutive inner sequence elements is similar for recovering the n initial state blocks. In this chapter, we will focus on recovering the initial state S^0 for simplicity.

Remark 1. *In this thesis, we restrict ourselves to only making guesses for initial state blocks. We will not guess combinations of initial state blocks. Guessing combinations of initial state blocks, in some specific cases, could lead to a lower G&D-resistance than our approach.*

Let's find the G&D-resistance of the runaway generator from Example 1. Recall that $n = 2$, so $S^0 = (a_0, a_1)$, and $c = (0,1)$, $d = (1,0)$ and $e = (1,1)$. Thus we can compute the next element of the inner sequence as $a_t = a_{t-2} + g(a_{t-1})$. Then the keystream sequence is defined as $z_t = a_t + a_{t+1}$. An exhaustive search for S^0 has a worst-case complexity of 2^{2b}. However, we can reduce our expected workload from 2^{2b} to 0 by using the knowledge of the KSG! We can mount a G&D attack, but we do not even have to guess any blocks. Recall from Section 2.2 that the attacker can perform any computation, so the attacker can also compute combinations of keystream blocks. If we, for example, add z_0 and z_1 we get $z_0 + z_1 = g(a_1)$. In Section 2.1 we specified that we only inspect runaway generators with invertible block functions g. This means we can solve the equation for a_1, so $a_1 = g^{-1}(z_0 + z_1)$. Since we know a_1 and z_0, we can also determine $a_0 = z_0 - a_1$. Then we need to check for correctness by computing $a_2 + a_3$ and checking whether this equals z_2. Since we do not have to guess any blocks, the G&D-resistance is 0 bits.

Here, we used that we can solve for a_1 in the equation $z_0 + z_1$. Let's define when an element is solvable in a certain equation. This allows us to define when we can determine that element.

Definition 5 (Solvability and Determinability)**.** *An element a_x is* solvable *in a given equation z if there is only one occurrence of a_x in the equation. Solvability of a_x allows us write the equation with a_x on the LFS and at least z on the RHS. The RHS can contain other inner sequence elements, but it cannot contain a_x. a_x is* determinable *from z if and only if a_x is*

solvable in z, we don't have knowledge of a_t and all elements on the RHS of the equation solved for a_t are known.

Let's look at an example where we can apply this.

Example 3. *Consider the case with $n = 3$ and with $c = (0,0,1)$, $d = (0,1,0)$ and $e = (1,0,1)$. See the table below for the iterations of the feedback function and output filter.*

t	a_t	z_t
0	a_0	$a_0 + a_2$
1	a_1	$a_0 + a_1 + g(a_1)$
2	a_2	$a_1 + a_2 + g(a_2)$
3	$a_0 + g(a_1)$	$a_0 + a_2 + g(a_1) + g(a_0 + g(a_1))$
4	$a_1 + g(a_2)$	$a_0 + a_1 + g(a_1) + g(a_2) + g(a_1 + g(a_2))$

Looking at the output blocks, we see that z_0, z_1 and z_2 each depend on only two distinct blocks. Then we have two equations z_3 and z_4 that are dependent of 3 distinct blocks. We want to find the easiest and quickest way to obtain the initial state $S^0 = (a_0, a_1, a_2)$. It can be the case that a certain combination of keystream blocks results in an equation where an element of the initial state is determinable. In this example, the result of adding two or more keystream blocks together is dependent of at least 2 inner sequence elements. For example, $z_0 + z_1 + z_2 = g(a_1) + g(a_2)$ and $z_0 + z_1 = a_1 + a_2 + g(a_1)$. Combining keystream blocks does not lead to a determinable element, so let us look only at the keystream sequence. If we guess a_2, we can determine $a_0 = z_0 - a_2$, or vice versa. We still need to gain knowledge of a_1 to know S^0. We have to find an equation from which a_1 is determinable. Equation z_1 is not solvable for a_1 (there are two occurrences of a_1), so we cannot use that one. z_2 is solvable for a_1 and we have knowledge of all other elements in the equation, namely a_2. This means we can determine $a_1 = z_2 - a_2 - g(a_2)$. To obtain the initial state, we only needed to guess a_2, so we have a G&D-resistance of b bits. The attack can be written down as the following algorithm:

```
for all guesses a'_2 for a_2:
        determine a'_0 = z_0 - a'_2
        determine a'_1 = z_2 - a'_2 - g(a'_2)
        if z_3 == a'_0 + a'_2 + g(a'_1) + g(a'_0 + g(a'_1)):
                return S^0 = (a'_0, a'_1, a'_2)
```

We can specify an upper and lower bound of the G&D-resistance.

Theorem 2. *The G&D-resistance of the runaway generator is lower bounded by 0 bits and upper bound by $(n-1)b$ bits. There is one special case which is an exception to this rule, see Example 4.*

Proof. The lowest G&D-resistance is 0 bits. This happens when the KSG is defined in such a way that the initial state is revealed by the keystream sequence, either directly or by combining keystream elements. Or equivalently, some state S^t at time t is fully recoverable without making any guesses. We of course cannot have less than 0 bits G&D-resistance.

The highest resistance theoretically possible is nb bits, which would mean the best attack is an exhaustive search for the initial state. However, this is not achievable with the runaway generator. The output filter is simply a linear combination of state blocks. What blocks are present is defined by the vector e, as described in Section 2.1. The first keystream block z_0 is always dependent of $HW(e)$ blocks. So from guessing $HW(e)-1$ initial state blocks, you can always determine the last initial state block. Recall that vector e has length n. That means that we can never have $HW(e) > n$. Hence we can never have a G&D-resistance of more than $n-1$ blocks. □

Example 4 (Exception to Theorem 2)**.** *Consider the case with $n = 1$ and with $c = (0)$, $d = (1)$. Then, if we take as output filter $z_i = a_i + a_{i+1}$, we get the equations displayed in the table below. Note that we cannot express e in a vector of length $n = 1$, hence we give the recurrence relation directly.*

t	a_t	z_t
0	a_0	$a_0 + g(a_0)$
1	$g(a_0)$	$g(a_0) + g(g(a_0))$

All keystream blocks are of the form $z_t = g^x(a_0) + g^y(a_0)$, where $x < y$. Also all combinations of keystream blocks are of this form. This means that all equations are not solvable for a_0, hence we can never determine a_0. So we need to guess a_0. This runaway generator has a G&D-resistance of b bits, which is larger than $(n-1)b = 0$ bits as specified by Theorem 2.

4.1 Automated Attack

Now that we have seen how to perform the attack by hand, we can automate this process. So far, we have only been looking at simple examples that are good to get some intuition from, but are not that interesting. To find an interesting runaway generator, we need to assess its parallelizability and G&D-resistance. As we have seen in Chapter 3, assessing the parallelizability of a runaway generator is pretty straightforward. The G&D-resistance requires some trial and error. As the resistance is highly dependent on the cancellation of initial state blocks in (combinations of) keystream blocks, there is no theorem on what the G&D-resistance is for generic runaway generators. We can look at many generators and try to find some pattern, varying n, c, d and e. As it costs a lot of effort to assess all these variants for G&D-resistance by hand and mistakes are easily made, it is preferable to automate this process. More importantly, if we can try every attack possible, we have assurance that we indeed find the most efficient attack under our assumptions. In this Section, we will describe our G&D-attack program and show why it indeed gives the most efficient attack, keeping Remark 1 in mind.

As input of the program, we get the three vectors c, d and e that completely describe the runaway generator. We can also give a list of equations as input, this will come in handy in Chapter 5. We compute all the equations based on c, d and e. To display these equations as simple as possible, we strip them. Each equation z_t becomes a list $\texttt{stripped}_t$. In the program, we will keep a list of stripped equations, where the index t indicates that it concerns z_t. Each term in z_t is either an inner sequence element, or a block function call with certain inner sequence elements as input. The first option, for a_x that is present in z_t, we represent simply as an integer x, so we only take the index. For each block function call, we remove the function g and place its inputs into a list. This is a recursive process, as the input of the block function call can either be inner sequence elements or block function calls to the inner sequence. The following example will make this idea clearer. For equation $z_x = a_0 + a_1 + g(a_1)$, stripping leads to $\texttt{stripped}_x = [0, 1, [1]]$. Another example is $z_y = a_0 + g(a_1) + g(a_1 + g(a_2))$. Then $\texttt{stripped}_y = [0, [1], [1, [2]]]$.

A description of the attack in pseudocode is given in Listing 4.1. We have simplified the algorithm by removing some optimizations. We also leave out the stripping of equations, the

generation of equations and calling the program.

The general idea of the attack is as follows. We want to try every possible attack to recover the initial state. We can achieve this by trying out all orders of guessing. For runaway generator with state size n, we can try all variations of $(0,\ldots,n-1)$ of length `upper`. `upper` is the upper bound as specified in Theorem 2, so $n-1$, but one can also give some other value. First we guess the first integer in the permutation, then the second, etc., until we have knowledge of all n initial state blocks. Maybe we don't have to guess anything and we can determine S^0 right away. In the beginning, for some variation p, we have no knowledge of any of the initial state blocks (`known = []`). If there is something to determine, we add those state blocks to `known` and update the equations accordingly, using `remove`. Then we try to guess $p[0]$. Perhaps we can determine more initial state blocks after this guess has been made. If at any point we know the entire initial state (`known` $= [a_0,\ldots,a_{n-1}]$), we are done with the attack. We then count how many blocks we have guessed in total and this is the G&D-resistance for that variation. We then move on to the next variation. The function `determinable` checks for each equation whether there is a single element in that equation. If that is the case, we can use `determine` to recover that inner sequence element. Determining and guessing basically come down to removing a certain element from each equation. When we guess the first element in p, we remove it from p. As it makes no sense to guess an element that we already have knowledge of, we have to remove elements in p that are also in `known`. It can happen that for some variation, each element has either been guessed or determined, without `known` being equal to S^0. We will show this with an example in Section 4.1.1. This means p will be empty and the attack is incomplete. We stop with this attack and we continue to the next variation.

Listing 4.1: Pseudo-code for all possible G&D-attacks.

```
def attack(equations, n, upper)
  gd ← upper + 1
  vars ← all variations of [0, ..., n-1] of length upper
  for each p in vars
    known, guessed ← []
    eqs ← equations
    while known != [0, ..., n-1]
      while determinable(eqs)[0]
          eqs, determined ← determine(eqs)
          known ← known ++ determined
      if known = [0, ..., n-1]
        continue
      for each element in p that is in known
          remove element from p
      if p is empty
        this attack is incomplete, go to the next variation
      eqs, known ← guess_element(eqs, p[0], known)
      guessed ← guessed ++ p[0]
    if the length of guessed < gd
      gd ← length of guessed
  return gd

def determine(eqs)
  _, dets, z = determinable(eqs)
  copy = eqs
  for each element in dets
    for each index, eq in eqs
        copy[index] = remove(eq, element)
  return eqs, dets

def guess(eqs, i, known)
  copy = eqs
  for each index, eq in eqs
    copy[index] ← remove(eq, i)
  known ← known ++ i
  return copy, known

def remove(eq, i)
  new ← []
  for each elem in eq
    if type(elem) = int
      if elem ≠ i
        new ← new ++ [elem]
    else if type(elem) = list
      temp ← remove(elem, i)
        if temp ≠ []
          new ← new ++ [temp]
  return new

def determinable(eqs)
  for each index, eq in eqs
    deter, var ← determinable_eq(eq)
    if deter is True
      return True, [var], index
  return False, [], []

def determinable_eq(eq)
  if len(eq) = 1
    if type(eq[0]) = int
      return True, eq[0]
    else if type(eq[0]) = list and len(eq[0]) = 1
      return determinable_eq(eq[0])
  return False, []
```

4.1.1 Example Run

Let's use the program to assess and example runaway generator. Let's pick the following runaway generator, with $n = 3$, $c = (0,0,1)$, $d = (1,1,0)$ and $e = (1,0,1)$. We collect n keystream blocks and their combinations to get the following equations.

$$\begin{aligned} z_0 &= a_0 + a_2 \\ z_1 &= a_0 + a_1 + g(a_1 + a_2) \\ z_2 &= a_1 + a_2 + g(a_0 + a_2 + g(a_1 + a_2)) \\ z_0 + z_1 &= a_1 + a_2 + g(a_1 + a_2) \\ z_0 + z_2 &= a_0 + a_1 + g(a_0 + a_2 + g(a_1 + a_2)) \\ z_0 + z_1 + z_2 &= g(a_1 + a_2) + g(a_0 + a_2 + g(a_1 + a_2)) \end{aligned}$$

We strip these equations as described in Section 4.1. The result is in the first row of Table 4.1. As the G&D-resistance cannot be larger than $n-1$, we define `upper`$= 2$. There are $\frac{3!}{(3-2)!} = 6$ variations, i.e. 6 different orders of guessing. We will now show the attack using the first variation, $p = (0,1)$. Initially, `determinable(eqs)` returns `false`, as there is no equation that contains in total just one variable. Hence, we have to guess the first element in the variation, being 0 (a_0). As we saw before, this comes down to simply removing all occurrences of 0 from the equations. The result can be seen in the second row of Table 4.1. After this, we have `known`$= [0]$ and $p = (1)$. We do not know all initial state elements yet, so

	z_0	z_1	z_2	$z_0 + z_1$	$z_0 + z_2$	$z_0 + z_1 + z_2$
Stripped	$[0,2]$	$[0,1,[1,2]]$	$[1,2,[0,2,[1,2]]]$	$[1,2,[1,2]]$	$[0,1,[0,2,[1,2]]]$	$[[1,2],[0,2,[1,2]]]$
Guessed 0	$[2]$	$[1,[1,2]]$	$[1,2,[2,[1,2]]]$	$[1,2,[1,2]]$	$[1,[2,[1,2]]]$	$[[1,2],[2,[1,2]]]$
Determined 2	$[]$	$[1,[1]]$	$[1,[[1]]]$	$[1,[1]]$	$[1,[[1]]]$	$[[1],[[1]]]$
Guessed 1	$[]$	$[]$	$[]$	$[]$	$[]$	$[]$

Table 4.1: An overview of the equations during an attack on runaway generator defined by $n = 3$, $c = (0,0,1)$, $d = (1,1,0)$ and $e = (1,0,1)$. The order of guessing used is $(0,1)$.

we iterate the while loop again. Now, the equation for z_0 is of length 1, so `determinable` returns `true`. Hence we can determine variable 2 from equation z_0, so we remove all occurrences of 2 from the equations. See the third row of Table 4.1. Now we have `known`$= [0,2]$ and $p = (1)$, so we iterate the while loop again. All the remaining equations are not solvable for 1, so we cannot determine a_1. `determinable` for these equation returns false, as we

cannot determine a_1. Hence, we have to guess $p[0] = 1$. This leads to all empty equations, as can be seen in the fourth row of Table 4.1. `known`$= [0,1,2]$, so the attack for this variation is done. Summarizing, we have performed the following attack:

Guessing variable a_0
  Determining a_2 from z_0
  Guessing variable a_1

The G&D-resistance against this attack is $2b$ bits. We then continue with doing this for the other 5 variations. The attacks that are not incomplete will all give a G&D-resistance of $2b$ blocks. So we take this attack shown above as the most efficient attack. As promised above, let's look at one such variation that leads to an incomplete attack. Take for example $p = (0,2)$. At the beginning of the attack, we have no knowledge of any of the initial state blocks and we cannot determine any of them. So we have to make a guess, namely for $p[0] = 0$. From this, we can determine the value of $a_2 = z_0 - a_0$. We now update p, as it makes no sense to guess elements that we have knowledge of, we remove them from p. But then p becomes empty! In words, we don't have knowledge of the complete initial state, but we also cannot guess any elements any more. Since we can never determine a_1 with knowledge of just a_0 and a_2, all variations that do not contain 1 lead to incomplete attacks.

4.1.2 Correctness

We want to show correctness of the program, i.e. the correctness of the function `attack`. First, we have to define what correctness means in this context: using the program should result in the correct G&D-resistance.

To show this, we first need to show that the way that we are stripping equations is allowed. We do this by showing that the stripping process is invertible, since if that is the case, we don't lose any information by stripping an equation. As we are working in $\mathbb{F}_2$, applying plus and minus have the same result. We can then remove these operations entirely from the equation, resulting in a list of terms that occur in the original equation. This is invertible as we can simply add pluses between the terms again to obtain the original equation. Each

term can be either an inner sequence element or a block function call. For the first case, we simply take the index of the inner sequence element and place it in the list. In Section 4.1, this process is described as recursive, but it doesn't have to be. Another way to perform the stripping, which makes showing invertibility easier, is to say we remove the block function by replacing each "$g($" with "[" and each ")" with a "]". We can reconstruct the original equation from the stripped list. For each term in the list, if it contains a list, replace that list with a block function call. Then sum all terms together to get the original equation. Hence, stripping equations is invertible. You could see the stripping as an invertible encoding of an equation.

We will now show the correctness of the subroutines of the function `attack` to finally prove correctness of `attack`.

`remove`

To show that the program is correct, we start by showing that removing known elements from the equations is allowed. Known elements are just constants, so we use that terminology. Removing constants is allowed, if it results in exactly the same G&D-resistance as not removing them. We obtain the same G&D-resistance if we can determine exactly the same elements as without removing the known elements, since then we also have to guess exactly the same elements as without removing the constants.

Let's say we want to determine initial state element a_x from equation z_t. We can only do this if a_x is determinable from z_t. This was described in Definition 5. To prove that we don't change the determinability of a_x in z_t, we have to show that we do not change the solvability of a_x in z_t and we do not change the unknown elements on the RHS of z_t solved for a_x.

We never remove an unknown element from the equation, so we never change the unknown elements of the RHS of z_t solved for a_x.
For solvability, we have to show that if a_x is solvable in z_t, we can still solve a_x from z_t after removing the constants from z_t. We also have to show that if a_x is not solvable in z_t, we can also not solve a_x from z_t after removing the constants from z_t. The first requirement is

fulfilled, as we can only make z_t unsolvable for a_x by adding a term a_x on the RHS. We are only removing, so this can never happen. The second requirement follows if we can show that we never remove a term a_x from the RHS. We only remove a term if we have knowledge of its value. So we can only remove a_x from the RHS if we have knowledge of a_x. We are trying to determine a_x, so we have no knowledge of a_x, hence we will never remove it from the equation.

If we do not change the determinability of some initial state element in some equation, it follows that we do not change the determinability of all elements in all equations. We can determine exactly the same elements as without removing the constants, then we also have to guess exactly the same elements as without removing the constants. Hence, the G&D-resistance is the same as without removing the constants.

The correctness of `determine` and `guess` follow from the correctness of `remove`. In both functions, we obtain knowledge of some initial state block and we have just shown that known elements can be removed from the equations.

`determinable`

Here we have to show that we only determine elements that are determinable according to Definition 5. In the program, we defined that an inner sequence element can be determined if we have a list of length 1. Let's call the only element in z_t again a_x. The LHS of the equation is a constant, since z_t is known. The t'th stripped equation [x] represents $c_2 = a_x + c_1$, so we can easily write $a_x = c_1 - c_2 = c_3$, where c_i is a constant. The equation is solvable for a_x, since there is only one variable and the rest is constant. We don't have knowledge of a_x, otherwise it would have been removed from the equation. We have no unknown elements in the equation besides a_x. The three requirements for determinability are fulfilled, so we can say that the function `determinable` works according to Definition 5.

attack

We have already shown that the subroutines of `attack` are correct. We still need to show that we indeed try all relevant orders of guessing and that we call the subroutines in the right order for some variation.

If we have the upper limit of the G&D-resistance to be `upper`, we only have to perform the attack for all orders of guessing of length `upper`. This is achieved by computing the variations of $[0,\ldots,n-1]$ of length $upper$.

For variation p, we want to continue the attack as long as `known` is not equal to S^0 and we can still guess and/or determine elements. We should always start to check `determinable` as we might not have to guess anything. Once `determinable` becomes `false`, we need to check whether we now know S^0. If that's the case, we are done for this variation. We might have guessed state elements that are present in p. We want to prevent that we guess a value that we already know, so we have to remove all known elements from p. Then it also might happen that p becomes empty. At that point, we cannot guess anything and we do not know the full initial state, so this order of guessing is not relevant. If p is not empty, we guess its first element. In the next iteration of the while-loop, the guessed element is removed from p.

4.2 Results of the Tool

We want a KSG with a security strength of at least 128 bits. We can achieve this with a KSG with 1 block of G&D-resistance if the block size $b = 128$. We can also achieve this with a KSG with 2 blocks of G&D-resistance, namely if $b = 64$, and so on. Depending on what block function one wants to use, n needs to be adapted accordingly to obtain a reasonable security strength. For some application, we might want 128 bits of security but parallelizability also needs to be high enough. Defining what a good runaway generator should look like is difficult as we do not know the application. Using the G&D-attack automation tool, we have generated an overview in Appendix A of some runaway generators and their

parallelizability and G&D-resistance. A remarkable result is the third to last generator in Table A.1. The generator is described by the relations $a_t = a_{t-n} + g(a_{t-n+1} + a_{t-n+2})$ and $z_t = a_t + a_{t+\lceil \frac{n}{2} \rceil} + a_{t+n-1}$, see Figure 4.1 for a graphical representation. Noticeable is that the internal state consists of $n = 4$ elements and this generator achieves a G&D-resistance of $n-1 = 3$ inner sequence elements, so $3b$ bits. This is the upper bound of G&D-resistance for runaway generators as was specified in Theorem 2. This means that if we want at least 128 bits of security, we could use a block function that operates on blocks of $b \geq 43$ bits. This means we could store the initial state in $4 \cdot 43 = 172$ bits, which is 21.5 bytes. However, when instantiating the block function with an actual function, the G&D-resistance might be lowered as opposed to the black box case. In Chapter 5, we show an example of this.

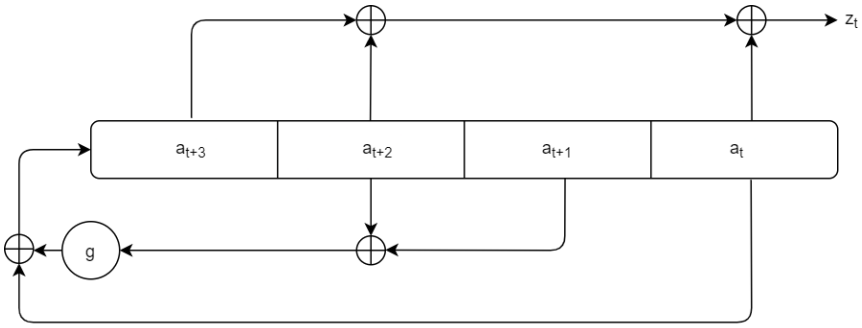

Figure 4.1: Graphical display of the runaway generator described by $c = (0,0,0,1)$, $d = (0,1,1,0)$ and $e = (1,0,1,1)$.

4.2.1 The Amount of Input Equations and Combinations

Another result of the tool was that we could test our intuitions. Throughout the research, our intuition was that we only needed combinations of the keystream blocks including z_0 to recover the initial state as efficiently as possible. The idea behind this was that for example $z_1 + z_2$ contains the same information as $z_0 + z_1$, but shifted. However, running the program on all combinations of keystream blocks occasionally resulted in a lower G&D-resistance. In hindsight, this result seems fair. Shifting an equation can make for blocks to cancel out. If we have x distinct blocks in equation $z_0 + z_1$, we might have y distinct blocks in equation $z_1 + z_2$. If $x > y$, we could have a better attack by including all combinations than by only having combinations with z_0. It could also occur that by shifting $z_0 + z_1$ the determinability of inner sequence elements in that equation changes.

Example 5. *Consider the runaway generator with* $n = 9$*,* $c = (0,\ldots,0,1)$*,* $d = (0,\ldots,0,1,0)$*,* $e = (1,0,0,1,0,0,0,0,1)$*. When we only consider combinations of n keystream blocks, we get the following attack:*

Guessing $variable\ a_0$
  Guessing $variable\ a_1$
    Determining $a_6\ from\ z_1$
    Determining $a_3\ from\ z_0+z_3+z_6$
    Guessing $variable\ a_2$
      Determining $a_7\ from\ z_2$
      Determining $a_8\ from\ z_3$
      Determining $a_5\ from\ z_0$
      Determining $a_4\ from\ z_5$

Hence, we have a G&D-resistance of $3b$ *bits. If we now consider all possible combinations of keystream blocks, we can improve on this attack. We get the following output from the program:*

Guessing $variable\ a_5$
  Guessing $variable\ a_7$
    Determining $a_2\ from\ z_1+z_4+z_5+z_7$
    Determining $a_1\ from\ z_2$
    Determining $a_4\ from\ z_5$
    Determining $a_3\ from\ z_0+z_3+z_4$
    Determining $a_8\ from\ z_3$
    Determining $a_0\ from\ z_0$
    Determining $a_6\ from\ z_1$

The interesting thing happens in determining a_2 *from* $z_1+z_4+z_5+z_7$*. Before we didn't have this combination of keystream blocks. Our assumption was that* $z_0+z_3+z_4+z_6$ *contains the same information as* $z_1+z_4+z_5+z_7$*. The first can be worked out as* $z_0+z_3+z_4+z_6 = a_4 + g(a_1)+g(a_4)+a_6+g(a_6)$*, which is only solvable for* a_1*. The second one is* $z_1+z_4+z_5+z_7 = a_5+g(a_2)+g(a_5)+a_7+g(a_7)$*. We can determine* a_2 *from this. So from including both these equations, we can determine* a_1 *if we know* a_4 *and* a_6*, and we can determine* a_2 *if we know* a_5 *and* a_7*. Including all combinations of keystream blocks gives us more information. Having*

more information could allow us to improve our attack.

Another assumption we made was that having n equations allowed us to perform the attack as efficiently as possible. That assumption also turned out to be incorrect. Having more equations is having more information to perform the attack. This can also lead to a lower G&D-resistance. In Appendix A under G&D-resistance there are two columns, one for having n equations and one for having $2n$ equations. You can see that the G&D-resistance is in some cases lower for $2n$ equation than for n equations. Having more equations can decrease the G&D-resistance, but it also takes much longer to perform the attacks. Maybe having $3n$ equations will lead to even better attacks, but this at this moment, the tool is too slow to check that.

In the following section, we will give some general recommendations and observations based on the results of the tool.

4.3 Recommendations

In this section we will make some recommendations. We found these recommendations by seeing a pattern in the results from the tool, or by confirming intuitions we already had.

4.3.1 Non-Linearity of the Block Function

In Section 2.1, besides being a permutation, we also stated that g should be non-linear. We saw there that no keystream generator should have a fully linear feedback function and output filter, as it allows the attacker to solve a linear system of equations, given more equations than unknowns. For the runaway generators, this property also ensures that less inner sequence elements cancel out in the (combinations of) keystream blocks. For larger n, the non-linearity aspect seems to be more important than for small n. The following example will illustrate this.

Example 6 (Non-Linear Block Function)**.** *We have the runaway generator defined by with* $n = 8$*,* $c = (0,0,0,0,0,0,0,1)$*,* $d = (0,0,0,0,0,1,1,0)$ *and* $e = (1,0,0,0,1,0,0,1)$*. This KSG has a G&D-resistance of* $4b$ *bits. The program outputs the following attack:*

Guessing $variable$ a_0
 Guessing $variable$ a_1
  Guessing $variable$ a_2
   Determining a_5 $from$ z_1
   Guessing $variable$ a_3
    Determining a_6 $from$ z_2
    Determining a_4 $from$ z_5
    Determining a_7 $from$ z_0

If we now change the program such that the block function is a linear function, i.e. $g(a+b) = g(a)+g(b)$*, we get the following attack:*

Guessing $variable$ a_0
  Guessing $variable$ a_1
   Guessing $variable$ a_2
    Determining a_5 $from$ z_1
    Determining a_3 $from$ $z_0+z_3+z_4$
    Determining a_6 $from$ z_2
    Determining a_4 $from$ z_5
    Determining a_7 $from$ z_0

The G&D-resistance has decreased from $4b$ *to* $3b$ *bits! Apparently, in equation* $z_0+z_3+z_4$ *blocks cancel out because of the linearity of g. Let's inspect that equation.*
$z_0+z_3+z_4 = a_2+g(a_1+a_2)+g(a_3+a_4)+g(a_4+a_5)$ *becomes*
$z_0+z_3+z_4 = a_2+g(a_1)+g(a_2)+g(a_3)+g(a_5)$*. State block* a_4 *has cancelled out, making this equation dependent of 1 less initial state block. The latter equation is only dependent of 4 distinct blocks, so the knowledge of* a_1*,* a_2 *and* a_5 *gives us knowledge of* a_3*.*

Linearity of the block function has decreased the G&D-resistance in this example. Hence, we want a non-linear block function.

4.3.2 Feedback Function

In order for the feedback function to be invertible, we need that $c \wedge d = (0,\ldots,0)$. In words, for each index $i \in [0,\ldots,n-1]$, at most one of $c[i]$ and $d[i]$ can be one, so not both ($c[i] = d[i] = 1$). This we already knew from Section 2.1.
From the results in Appendix A, we can see that it is beneficial for the G&D-resistance that we add blocks as input of the block function, so we increase $HW(d)$. However, there is a trade-off with parallelizability, as can also be seen in Appendix A.

4.3.3 Output Filter

We can make a recommendation about what the output filter, i.e. the vector e, should look like. Consider the following example.

Example 7. *Consider the case with $n = 3$ and with $c = (0,0,1)$, $d = (0,1,0)$ and $e = (0,1,1)$. This example differs from Example 3 in the output filter. See the table below for the iterations of the feedback function and output filter.*

t	a_t	z_t
0	a_0	$a_1 + a_2$
1	a_1	$a_2 + a_3 = a_0 + a_2 + g(a_1)$
2	a_2	$a_3 + a_4 = a_0 + a_1 + g(a_1) + g(a_2)$
3	$a_0 + g(a_1)$	$a_4 + a_5 = a_1 + a_2 + g(a_2) + g(a_0 + g(a_1))$

Some output blocks like z_1 now have more distinct state blocks than before. However, now if we look at the combination $z_0 + z_1 + z_2 = g(a_2)$ we see that we get a_2 for free. Then we can also compute $a_1 = z_0 - a_2$ and $a_0 = z_1 - a_2 - g(a_1)$. Hence the G&D-resistance has decreased from b bits to 0! The problem is that we are not using the first inner sequence element in the state block, i.e. $e[0] \neq 1$. Adding the first three keystream blocks results in $z_0 + z_1 + z_2 = a_1 + a_4$. Adding $a_4 = a_1 + g(a_2)$ and a_1 cancels out the a_1 term, as a_4 contains a terms a_1. We don't have this problem if we would have $e = (1,0,1)$ like in Example 3. It that example, adding the first three keystream blocks leads to $z_0 + z_1 + z_2 = a_0 + a_1 + a_3 + a_4 = g(a_1) + g(a_2)$.

A similar idea holds for the last block. If for z_t, we do not incorporate the a_{t-1} block, but

instead take term a_{t-2}, we get that z_0 and z_1 are both just dependent of 2 blocks. This makes it easier for an attacker to retrieve the initial state.

Our recommendation is to have $e[0] = e[n-1] = 1$, i.e. z_t should at least be dependent of a_{t-n} and a_{t-1}, to decrease the likelihood of inner sequence elements cancelling out. It can still happen that you obtain a G&D-resistance of 0 bits while following this recommendation. Or that you get a higher G&D-resistance without following this recommendation. But so far, we have reason to believe that this can improve the G&D-resistance.

Chapter 5

SKINNY as Block Function

In this chapter, we will investigate what happens to the G&D-resistance if we use a specific block function. In Section 2.1 we described some requirements for the block function, namely it should be non-linear and a permutation. We can use a block cipher such as AES as block function, as these are per definition a permutation when the key is fixed, however, we want the block function to be relatively lightweight. SKINNY [Beierle et al., 2016] is a family of lightweight block ciphers, where we can choose between a block size b of 64 or 128 bits. To make it even more lightweight, we alter the round function somewhat. Namely, we don't perform the AddConstants and AddRoundTweakey operations. The AddConstants transformation, as the name suggests, adds constants. We are performing our G&D-attacks abstractly, namely without concrete values for the keystream and the initial state. Adding constant does not make a difference in analysing the G&D-resistance. Leaving out the AddRoundTweakey transformation is the same as taking the tweakey words and subtweakeys as 0. We will denote this adapted round function as SKINNY'.

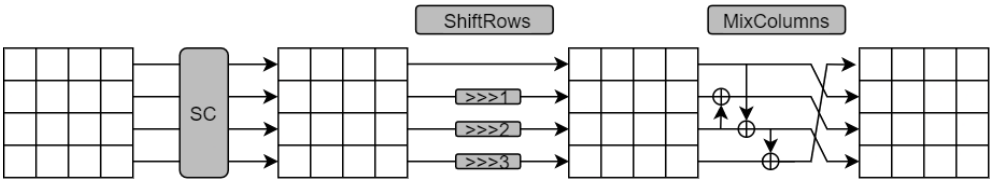

Figure 5.1: One round of SKINNY', an adaptation of the SKINNY round function. We only perform the SubCells, ShiftRows and MixColumns transformations, leaving out AddConstants and AddRoundTweakey.

We will show that exploiting specific properties of SKINNY' allows us to perform a better G&D-attack than in the black box case, so performing a generic attack. In Chapter 4, we used an abstract block function g. We will show that performing a single round of SKINNY'

allows us the reduce the G&D-resistance of some examples we saw before. We will then also show how many rounds of SKINNY' are needed to obtain the same G&D-resistance as in the black box case.

We start by showing what the result is of applying SKINNY' to some message m. For the initial state, we split m into 16 nibbles (or bytes) as follows:

$$\begin{pmatrix} m_0 & m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 & m_7 \\ m_8 & m_9 & m_{10} & m_{11} \\ m_{12} & m_{13} & m_{14} & m_{15} \end{pmatrix} \tag{5.1}$$

For this initial state, we can write out the result of applying 1 round of the adapted SKINNY round function.

The 4-bit S-box is defined in Table 2 in [Beierle et al., 2016]. Instead of using a lookup table, the authors also describe how to compute each bit of the output of the S-box. It can be computed by applying the transformation shown just below Table 2 four times and three left rotations. For a nibble m_i consisting of four bits $(m_{i,3}, m_{i,2}, m_{i,1}, m_{i,0})$, where $m_{i,3}$ is the most significant bit, the S-box(m_i) is:

$$\begin{cases} m'_{i,3} = m_{i,0} \oplus \overline{(m_{i,2} \vee m_{i,3})} \\ m'_{i,2} = m_{i,3} \oplus \overline{(m_{i,1} \vee m_{i,2})} \\ m'_{i,1} = m_{i,2} \oplus \overline{(m_{i,1} \vee (m_{i,0} \oplus \overline{(m_{i,2} \vee m_{i,3})}))} \\ m'_{i,0} = m_{i,1} \oplus \overline{((m_{i,0} \oplus \overline{(m_{i,2} \vee m_{i,3})}) \vee (m_{i,3} \oplus \overline{(m_{i,1} \vee m_{i,2})}))} \end{cases}$$

The result of the S-box is solely based on its input, so there is no diffusion between nibbles m_i. We can see that each equation for $m'_{i,j}$ is dependent of $m_{i,j}$. We can also see that learning some $m'_{i,j}$ has no influence on the other equations. This means we can never guess some bits and determine some others for free. So at least for G&D-attacks, working on bit-level does not yield more information that we would improve the G&D-attack. Hence, let's write the

result of the SubCells transformation, so applying the S-box to each nibble as:

$$\begin{pmatrix} m'_0 & m'_1 & m'_2 & m'_3 \\ m'_4 & m'_5 & m'_6 & m'_7 \\ m'_8 & m'_9 & m'_{10} & m'_{11} \\ m'_{12} & m'_{13} & m'_{14} & m'_{15} \end{pmatrix} \tag{5.2}$$

Then ShiftRows transformation rotates row j right with j positions, where $0 \leq j \leq 3$. This leads to

$$\begin{pmatrix} m'_0 & m'_1 & m'_2 & m'_3 \\ m'_7 & m'_4 & m'_5 & m'_6 \\ m'_{10} & m'_{11} & m'_8 & m'_9 \\ m'_{13} & m'_{14} & m'_{15} & m'_{12} \end{pmatrix} \tag{5.3}$$

And finally, applying MixColumns yields:

$$\texttt{SKINNY}(m) = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} m'_0 & m'_1 & m'_2 & m'_3 \\ m'_7 & m'_4 & m'_5 & m'_6 \\ m'_{10} & m'_{11} & m'_8 & m'_9 \\ m'_{13} & m'_{14} & m'_{15} & m'_{12} \end{pmatrix} = \tag{5.4}$$

$$\begin{pmatrix} m'_0 + m'_{10} + m'_{13} & m'_1 + m'_{11} + m'_{14} & m'_2 + m'_8 + m'_{15} & m'_3 + m'_9 + m'_{12} \\ m'_0 & m'_1 & m'_2 & m'_3 \\ m'_7 + m'_{10} & m'_4 + m'_{11} & m'_5 + m'_8 & m'_6 + m'_9 \\ m'_0 + m'_{10} & m'_1 + m'_{11} & m'_2 + m'_8 & m'_3 + m'_9 \end{pmatrix}$$

5.1 Application to Non-Determinable Case

We saw that $z = m + g(m)$ is not determinable for m, among others in Example 4. In the black box model, this means we can never determine m for free. However, if we instead have one round of SKINNY′ as block function, we can greatly reduce the G&D-resistance. Using Equation 5.4, we can write out $z = m + \texttt{SKINNY}'(m)$ as follows. Note that we also write out

the keystream block z in nibbles.

$$\begin{pmatrix} z_0 & z_1 & z_2 & z_3 \\ z_4 & z_5 & z_6 & z_7 \\ z_8 & z_9 & z_{10} & z_{11} \\ z_{12} & z_{13} & z_{14} & z_{15} \end{pmatrix} = \begin{pmatrix} m_0 & m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 & m_7 \\ m_8 & m_9 & m_{10} & m_{11} \\ m_{12} & m_{13} & m_{14} & m_{15} \end{pmatrix} + \tag{5.5}$$

$$\begin{pmatrix} m'_0+m'_{10}+m'_{13} & m'_1+m'_{11}+m'_{14} & m'_2+m'_8+m'_{15} & m'_3+m'_9+m'_{12} \\ m'_0 & m'_1 & m'_2 & m'_3 \\ m'_7+m'_{10} & m'_4+m'_{11} & m'_5+m'_8 & m'_6+m'_9 \\ m'_0+m'_{10} & m'_1+m'_{11} & m'_2+m'_8 & m'_3+m'_9 \end{pmatrix} =$$

$$\begin{pmatrix} m_0+m'_0+m'_{10}+m'_{13} & m_1+m'_1+m'_{11}+m'_{14} & m_2+m'_2+m'_8+m'_{15} & m_3+m'_3+m'_9+m'_{12} \\ m_4+m'_0 & m_5+m'_1 & m_6+m'_2 & m_7+m'_3 \\ m_8+m'_7+m'_{10} & m_9+m'_4+m'_{11} & m_{10}+m'_5+m'_8 & m_{11}+m'_6+m'_9 \\ m_{12}+m'_0+m'_{10} & m_{13}+m'_1+m'_{11} & m_{14}+m'_2+m'_8 & m_{15}+m'_3+m'_9 \end{pmatrix}$$

If we know the value of m_i we also know the value of m'_i and vice versa, by just applying the (inverse) of the S-box. If we guess m'_0, we can determine $m_4 = z_4 - m'_0$. If we then also guess m_1 and m_8, we can determine all other nibbles. See below the full attack. So we only have to guess 3 nibbles. This comes down to just 12 bits of G&D-resistance. Hence, with a single SKINNY′ round as block function instead of an abstract one, the G&D-resistance has decreased from 64 to 12 bits.

Guessing variable m_0
- **Determining** m_4 from z_4
- **Guessing** variable m_1
  - **Determining** m_5 from z_5
  - **Guessing** variable m_8
    - **Determining** m_{10} from z_{10}
    - **Determining** m_{13} from z_0
    - **Determining** m_7 from z_8
    - **Determining** m_3 from z_7
    - **Determining** m_{12} from z_{12}
    - **Determining** m_9 from z_3

Determining m_{11} from z_9

Determining m_{14} from z_1

Determining m_6 from z_{11}

Determining m_2 from z_6

Determining m_{15} from z_2

5.1.1 Performing More Rounds

Ideally, the G&D-resistance of $z = m + \texttt{SKINNY'}(m)$ is equal to the G&D-resistance of $z = m + g(m)$, namely without knowledge of the block function. We can perform more rounds of `SKINNY'` until we get 64 bits of resistance, at the expense of efficiency. The findings can be summarized in the following overview:

Number of rounds	G&D-resistance
1	12 bits
2	20 bits
3	32 bits
4	40 bits
5	52 bits
6	56 bits
7	60 bits
8	64 bits

Table 5.1: The G&D-resistance in bits for the case $z = m +$ `SKINNY'-x`(m), where the block function used is x-rounds of `SKINNY'`.

These results have been generated using the G&D-attack tool. We had to define the round function of `SKINNY'`. For any number of rounds r, we can define the matrix for the keystream block in nibbles (or bytes) like `z = m + skinny_n(m, r)`. We obtain 16 equations from this. By looking at the structure of the equations in Equation 5.5, we see we can perform Gaussian elimination to get simpler equations. For example, we can add the last row of the matrix to the first row of the matrix. We can add the second row to the last row. Note that this doesn't decrease the amount of distinct blocks in the last row, but it changes what blocks are present. In the attack for `SKINNY'-3` and `SKINNY'-4`, this sum of rows was used to

improve the attack. We obtain the following matrix:

$$\begin{pmatrix} m_0+m'_0+m'_{10}+m'_{13} & m_1+m'_1+m'_{11}+m'_{14} & m_2+m'_2+m'_8+m'_{15} & m_3+m'_3+m'_9+m'_{12} \\ m_4+m'_0 & m_5+m'_1 & m_6+m'_2 & m_7+m'_3 \\ m_8+m'_7+m'_{10} & m_9+m'_4+m'_{11} & m_{10}+m'_5+m'_8 & m_{11}+m'_6+m'_9 \\ m_{12}+m'_0+m'_{10} & m_{13}+m'_1+m'_{11} & m_{14}+m'_2+m'_8 & m_{15}+m'_3+m'_9 \end{pmatrix}$$

$$\xrightarrow[R_3=R_2+R_3]{R_0=R_0+R_3} \begin{pmatrix} m_0+m_{12}+m'_{13} & m_1+m_{13}+m'_{14} & m_2+m_{14}+m'_{15} & m_3+m_{15}+m'_{12} \\ m_4+m'_0 & m_5+m'_1 & m_6+m'_2 & m_7+m'_3 \\ m_8+m'_7+m'_{10} & m_9+m'_4+m'_{11} & m_{10}+m'_5+m'_8 & m_{11}+m'_6+m'_9 \\ m_4+m_{12}+m'_{10} & m_5+m_{13}+m'_{11} & m_6+m_{14}+m'_8 & m_7+m_{15}+m'_9 \end{pmatrix}$$

For the attack, we use the original equations, so the top matrix, and we use the first and last row of the lower matrix. We could have added more or different rows to each other, which would perhaps lead to a more efficient attack. Having all the $\sum_{i=2}^{16} C(16,i) = 65519$ equations gives more information, hence having all of these would possibly give a better attack. We feed the equations to the G&D-automation tool with $n = 16$, as we need to recover 16 nibbles.

The program shows us that we need $z = m + \texttt{SKINNY'-8}(m)$ to have the same G&D-resistance as the generic block function case. It is possible that these attacks can be improved upon. Like described in Remark 1, we only guess inner sequence elements (or nibbles of inner sequence elements), and not combinations of inner sequence elements. Also, having more information, so more equations could lower the G&D-resistances listed in Table 5.1. Ideally, we would also use all combinations of these equations, of any length between 2 and 16. This would result in $\sum_{i=2}^{16} C(16,i) = C(16,2)+C(16,3)+\cdots+C(16,16) = 65519$ equations. As the G&D-tool has to perform each order of guessing for 16 nibbles, this becomes way too slow. Even though the attacks could be improved on, our intuition is that the G&D-resistance for taking as block function 8-round SKINNY' will remain 16 nibbles.

5.2 More Complex Example

Let's again look at Example 3, where $c = (0,0,1)$, $d = (0,1,0)$ and $e = (1,0,1)$. We want to use our knowledge of SKINNY′ to improve on the attack described before. The G&D-resistance in the original attack in Example 3 was 1 block. We again start by writing out the equations z_0, z_1 and z_2 in nibbles. So we will get $3 \cdot 16 = 48$ equations. Since $z_0 = a_0 + a_2$ contains no block function call, we will simply get $z_{0,i} = a_{0,i} + a_{2,i}$ for $0 \leq i < 16$. From these equations we learn $a_{0,i}$ if we know $a_{2,i}$ or vice versa. Writing out $z_1 = a_0 + a_1 + \texttt{SKINNY}'(a_1)$ using Equation 5.4 becomes:

$$\begin{pmatrix} z_{1,0} & z_{1,1} & z_{1,2} & z_{1,3} \\ z_{1,4} & z_{1,5} & z_{1,6} & z_{1,7} \\ z_{1,8} & z_{1,9} & z_{1,10} & z_{1,11} \\ z_{1,12} & z_{1,13} & z_{1,14} & z_{1,15} \end{pmatrix} = \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{0,4} & a_{0,5} & a_{0,6} & a_{0,7} \\ a_{0,8} & a_{0,9} & a_{0,10} & a_{0,11} \\ a_{0,12} & a_{0,13} & a_{0,14} & a_{0,15} \end{pmatrix} + \begin{pmatrix} a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{1,4} & a_{1,5} & a_{1,6} & a_{1,7} \\ a_{1,8} & a_{1,9} & a_{1,10} & a_{1,11} \\ a_{1,12} & a_{1,13} & a_{1,14} & a_{1,15} \end{pmatrix} +$$

$$\begin{pmatrix} a'_{1,0} + a'_{1,10} + a'_{1,13} & a'_{1,1} + a'_{1,11} + a'_{1,14} & a'_{1,2} + a'_{1,8} + a'_{1,15} & a'_{1,3} + a'_{1,9} + a'_{1,12} \\ a'_{1,0} & a'_{1,1} & a'_{1,2} & a'_{1,3} \\ a'_{1,7} + a'_{1,10} & a'_{1,4} + a'_{1,11} & a'_{1,5} + a'_{1,8} & a'_{1,6} + a'_{1,9} \\ a'_{1,0} + a'_{1,10} & a'_{1,1} + a'_{1,11} & a'_{1,2} + a'_{1,8} & a'_{1,3} + a'_{1,9} \end{pmatrix}$$

And very similar for $z_2 = a_1 + a_2 + \texttt{SKINNY}'(a_2)$:

$$\begin{pmatrix} z_{2,0} & z_{2,1} & z_{2,2} & z_{2,3} \\ z_{2,4} & z_{2,5} & z_{2,6} & z_{2,7} \\ z_{2,8} & z_{2,9} & z_{2,10} & z_{2,11} \\ z_{2,12} & z_{2,13} & z_{2,14} & z_{2,15} \end{pmatrix} = \begin{pmatrix} a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{1,4} & a_{1,5} & a_{1,6} & a_{1,7} \\ a_{1,8} & a_{1,9} & a_{1,10} & a_{1,11} \\ a_{1,12} & a_{1,13} & a_{1,14} & a_{1,15} \end{pmatrix} + \begin{pmatrix} a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{2,4} & a_{2,5} & a_{2,6} & a_{2,7} \\ a_{2,8} & a_{2,9} & a_{2,10} & a_{2,11} \\ a_{2,12} & a_{2,13} & a_{2,14} & a_{2,15} \end{pmatrix} +$$

$$\begin{pmatrix} a'_{2,0} + a'_{2,10} + a'_{2,13} & a'_{2,1} + a'_{2,11} + a'_{2,14} & a'_{2,2} + a'_{2,8} + a'_{2,15} & a'_{2,3} + a'_{2,9} + a'_{2,12} \\ a'_{2,0} & a'_{2,1} & a'_{2,2} & a'_{2,3} \\ a'_{2,7} + a'_{2,10} & a'_{2,4} + a'_{2,11} & a'_{2,5} + a'_{2,8} & a'_{2,6} + a'_{2,9} \\ a'_{2,0} + a'_{2,10} & a'_{2,1} + a'_{2,11} & a'_{2,2} + a'_{2,8} & a'_{2,3} + a'_{2,9} \end{pmatrix}$$

To improve on the generic case with abstract g, we should be able obtain all 48 nibbles (a_0, a_1 and a_2) with less than 16 guessed nibbles. We have the equations z_0, z_1 and z_2, but the

combination of the first three keystream blocks is also interesting. We already saw in Chapter 4 Example 3 that $z_0 + z_1 + z_2 = g(a_1) + g(a_2)$. If we use the adapted SKINNY function, this becomes:

$$\begin{pmatrix} z_{0,0} & z_{0,1} & z_{0,2} & z_{0,3} \\ z_{0,4} & z_{0,5} & z_{0,6} & z_{0,7} \\ z_{0,8} & z_{0,9} & z_{0,10} & z_{0,11} \\ z_{0,12} & z_{0,13} & z_{0,14} & z_{0,15} \end{pmatrix} + \begin{pmatrix} z_{1,0} & z_{1,1} & z_{1,2} & z_{1,3} \\ z_{1,4} & z_{1,5} & z_{1,6} & z_{1,7} \\ z_{1,8} & z_{1,9} & z_{1,10} & z_{1,11} \\ z_{1,12} & z_{1,13} & z_{1,14} & z_{1,15} \end{pmatrix} + \begin{pmatrix} z_{2,0} & z_{2,1} & z_{2,2} & z_{2,3} \\ z_{2,4} & z_{2,5} & z_{2,6} & z_{2,7} \\ z_{2,8} & z_{2,9} & z_{2,10} & z_{2,11} \\ z_{2,12} & z_{2,13} & z_{2,14} & z_{2,15} \end{pmatrix} =$$

$$\begin{pmatrix} a'_{1,0} + a'_{1,10} + a'_{1,13} & a'_{1,1} + a'_{1,11} + a'_{1,14} & a'_{1,2} + a'_{1,8} + a'_{1,15} & a'_{1,3} + a'_{1,9} + a'_{1,12} \\ a'_{1,0} & a'_{1,1} & a'_{1,2} & a'_{1,3} \\ a'_{1,7} + a'_{1,10} & a'_{1,4} + a'_{1,11} & a'_{1,5} + a'_{1,8} & a'_{1,6} + a'_{1,9} \\ a'_{1,0} + a'_{1,10} & a'_{1,1} + a'_{1,11} & a'_{1,2} + a'_{1,8} & a'_{1,3} + a'_{1,9} \end{pmatrix} +$$

$$\begin{pmatrix} a'_{2,0} + a'_{2,10} + a'_{2,13} & a'_{2,1} + a'_{2,11} + a'_{2,14} & a'_{2,2} + a'_{2,8} + a'_{2,15} & a'_{2,3} + a'_{2,9} + a'_{2,12} \\ a'_{2,0} & a'_{2,1} & a'_{2,2} & a'_{2,3} \\ a'_{2,7} + a'_{2,10} & a'_{2,4} + a'_{2,11} & a'_{2,5} + a'_{2,8} & a'_{2,6} + a'_{2,9} \\ a'_{2,0} + a'_{2,10} & a'_{2,1} + a'_{2,11} & a'_{2,2} + a'_{2,8} & a'_{2,3} + a'_{2,9} \end{pmatrix} \quad (5.6)$$

We are especially interested in the second row of Equation 5.6. If we guess $a_{2,0}, a_{2,1}, a_{2,2}$ and $a_{2,3}$, we can determine $a_{1,0}, a_{1,1}, a_{1,2}$ and $a_{1,3}$. Before we also saw that with knowledge of some $a_{2,i}$ we can also determine $a_{0,i}$. We can then actually determine all 48 nibbles by guessing 5 nibbles. However, feeding these equations to the G&D-attack tool, we find an even more efficient attack. If we guess $a_{0,0}$, $a_{0,1}$, $a_{0,2}$ and $a_{0,10}$, we can determine all other nibbles. Thus, with one round SKINNY', we can reduce the G&D-resistance from 64 bits to 16 bits!

5.2.1 Performing More Rounds

Here also, we can perform more rounds SKINNY'. However, already when using SKINNY'-2, the equations become so fuzzy that it is really hard performing an attack by hand. The following results were generated by the tool:

Number of rounds	G&D-resistance
1	16 bits
2	40 bits
3	56 bits
4	64 bits

Table 5.2: The G&D-resistance in bits for Example 3, where the block function used is x-rounds of SKINNY'.

We perform the same Gaussian elimination as described in Section 5.1.1, namely $R_0 = R_0 + R_3$ and $R_3 = R_2 + R_3$. We perform this Gaussian elimination on z_1, z_2 and $z_0 + z_1 + z_2$. We use the G&D-tool in the same manner as was described in Section 5.1.1, namely we feed the equations to the tool, now with $n = 3 \cdot 16 = 48$. Checking the result is also difficult, as we have 60 equations and 48 unknowns. What is noticeable is that the Gaussian elimination is only used to recover 1 nibble in total, for all number of rounds $1 \leq r < 5$.

It is noticable that for this example we only need 4 rounds of SKINNY', whereas for the case described in Section 5.1.1 we needed 8 rounds to obtain 64 bits of security.

Chapter 6

Related Work

We already described in Chapter 1 that LFSRs are not suited as a cryptographic KSG. The main characteristics of runaway generators are non-linearity of the update function and that no part of the initial state is kept during generation of the keystream. In addition, they are low-cost, word-oriented and synchronous. To make a stream cipher non-linear, non-linearity can be introduced in the update function, the output filter, or both. Runaway generators only have non-linearity in the feedback function. Our idea behind taking this first approach is that making a function non-linear introduces computational overhead, making the generator less efficient, as was described in Section 2.1. In this chapter, we will describe some of the related work, i.e. stream ciphers that have similar characteristics.

RC4 is similar to the runaway generators in the fact that it has a simple structure and it has a non-linear update function. However, multiple vulnerabilities in RC4 have been shown, such as in [Fluhrer et al., 2001], where the authors performed a related key attack. HC-128 [Wu, 2008] is a modernised version of RC4. The authors claim that the most efficient attack for recovering the initial key is exhaustive key search, as the key is 128-bit, it offers 128-bit security. Apart from having a non-linear update function, it also has a non-linear output filter, which adds overhead. The internal state consists of two tables of 512 registers containing 32-bit words, in total taking up 4096 bytes. Even though runaway generators could have any internal state size, we aim for it to be smaller. We saw for example in Section 4.2 a runaway generator that has an internal state of 21.5 bytes, while having 128-bit security against G&D-attacks.

RC4 and HC-128 are both software-efficient. For the runaway generators, the block function

decides whether it is software- or hardware-oriented, but it would be nice to have both. Trivium [Cannière, 2006] is both software- and hardware-efficient. However, they only claim 80 bits of security, whereas we strive to have at least 128-bit security.

The Snow stream ciphers [Ekdahl and Johansson, 2003] are also software-oriented. A difference between Snow and the runaway generators is that Snow has non-linearity in the output filter. As mentioned in the beginning of this chapter, this increases the computational complexity.

Two ciphers that are also similar to the runaway generators are `Tiaoxin-346` [Nikolić, 2016] and AEGIS [Wu and Preneel, 2013]. These are not stream ciphers but authenticated encryption algorithms. Authenticated encryption provides both confidentiality and integrity of the data. One can achieve this by using an encryption scheme and a message authentication code (MAC) for message authentication. However, it is more efficient to make a scheme that does both of this, an authenticated encryption algorithm. These schemes do not output a keystream, but a ciphertext and a tag. The plaintext is used in the state update function, such that AEGIS and `Tiaoxin` are asynchronous. Both schemes use AES in the update function to make it non-linear. In this sense they are similar to the runaway generators: the update function is a simple non-linear function where XORs and a block function (AES) is used. They both are also runaway, namely no part of the initial state is kept throughout generation of the keystream.

Chapter 7

Conclusions and Future Research

In this thesis, we have seen that we can make keystream generators from simple building blocks. In particular, we have seen that we can even achieve the upper bound of $n-1$ blocks (of b bits) of G&D-resistance for some specific configurations. In Section 4.3, we have made some recommendation on what the vectors c, d and e should look like to have a high probability of getting a good G&D-resistance. For example, adding more inner sequence elements together as input of the block function generally leads to a better G&D-resistance, i.e. having some more non-zero values in d. This comes at the expense of parallelizability. We have seen in Chapter 3 that it is easy to assess the parallelizability of a runaway generator. It simply comes down to finding the lowest index in c and d which has a non-zero value.

We have also shown that G&D-attacks can be automated. This comes in particularly handy for larger n or when assessing generators with a specific block function. The main goal of the tool is to find the most efficient G&D-attack possible under the assumption made in Remark 1, by checking every attack possible. It is hard to verify the result, as we cannot check every attack by hand. We have shown our algorithm finds the correct G&D-resistance in Section 4.1.2. However, more information could lead to a better attack, such as having more equations and combinations of those equations. In specific cases, so far only for some $n > 7$ cases, having $2n$ equations leads to a better attack than having n equations. Maybe having $3n$ equations will further reduce the G&D-resistance of some runaway generators. Unfortunately, this is computationally too hard to do with the current program. For $2n$ equations we already had problems with computing the G&D-resistance for large n, as can be seen in the overview in Appendix A. A dash "-" shows that it took too long to compute. In future research, it would be good to allow the attacker to query the runaway generator any amount of keystream

blocks. The attaker should also be allowed to guess combinations of inner sequence elements.

We have only looked at the security of the runaway generators in terms of the G&D-resistance, but of course, other attacks are possible. When we take a concrete block function, the attacker might be able to perform a better attack than in the black box case, like we have seen in Chapter 5 using the round function `SKINNY'`. The attacker could also distinguish the runaway generator from a $\mathscr{RO}$ by looking at statistical properties of the generated keystream. The block function could have imbalance in its output, namely a bias towards specific values. We have also assumed so far that the initial state is a given uniform secret, whereas in a real application, we would have to generate this, leading to an imbalanced initial state. An attacker could exploit these statistical properties, perhaps leading to a better attack than a G&D-attack.

The research done for this thesis is useful to construct specific lightweight keystream generators. We have seen some configurations of runaway generators that look promising. Further research could be done in picking a specific block function and assessing how it changes the G&D-resistance as opposed to the generic block function case. G&D-attacks are just one way to distinguish the runaway generator from a $\mathscr{RO}$. Implementing a specific block function can give rise to other attacks. The block function used might be statistically biased, i.e. the keystream will have imbalance. An attacker can then also try to distinguish the runaway generator from a $\mathscr{RO}$ by looking at statistical properties of the keystream. So to get to a specific efficient and low-cost stream cipher, other properties such as imbalance should be investigated in future work.

An application of a specific runaway generator would be that we can use the structure of the generator to do compression, instead of expansion like a stream cipher does. These two functionalities, being expansion and compression, is what doubly extendable cryptographic keyed (deck) functions such as Xoofff [Daemen et al., 2018] are able to do. These functions take both variable length input and output, with which one could build stream ciphers, MAC functions and authenticated encryption schemes.

References

[Barkan and Biham, 2006] Barkan, E. and Biham, E. (2006). Conditional estimators: An effective attack on a5/1. In Preneel, B. and Tavares, S., editors, *Selected Areas in Cryptography*, pages 1–19, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Beierle et al., 2016] Beierle, C., Jean, J., Kölbl, S., Leander, G., Moradi, A., Peyrin, T., Sasaki, Y., Sasdrich, P., and Sim, S. M. (2016). The `SKINNY` family of block ciphers and its low-latency variant mantis. In Robshaw, M. and Katz, J., editors, *Advances in Cryptology – CRYPTO 2016*, pages 123–153, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Cannière, 2006] Cannière, C. D. (2006). Trivium: A stream cipher construction inspired by block cipher design principles. In Katsikas, S. K., López, J., Backes, M., Gritzalis, S., and Preneel, B., editors, *Information Security, 9th International Conference, ISC 2006, Samos Island, Greece, August 30 - September 2, 2006, Proceedings*, volume 4176 of *Lecture Notes in Computer Science*, pages 171–186. Springer.

[Daemen et al., 2018] Daemen, J., Hoffert, S., Van Assche, G., and Van Keer, R. (2018). The design of xoodoo and xoofff. *IACR Transactions on Symmetric Cryptology*, 2018(4):1–38.

[Ekdahl and Johansson, 2003] Ekdahl, P. and Johansson, T. (2003). A new version of the stream cipher snow. In Nyberg, K. and Heys, H., editors, *Selected Areas in Cryptography*, pages 47–61, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Fluhrer et al., 2001] Fluhrer, S., Mantin, I., and Shamir, A. (2001). Weaknesses in the key scheduling algorithm of rc4. In Vaudenay, S. and Youssef, A. M., editors, *Selected Areas in Cryptography*, pages 1–24, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Nikolić, 2016] Nikolić, I. (2016). `Tiaoxin-346`. `https://competitions.cr.yp.to/round3/tiaoxinv21.pdf`.

[Shah and Mahalanobis, 2012] Shah, J. and Mahalanobis, A. (2012). A new guess-and-determine attack on the a5/1 stream cipher. Cryptology ePrint Archive, Report 2012/208. https://eprint.iacr.org/2012/208.

[Wu, 2008] Wu, H. (2008). *The Stream Cipher HC-128*, pages 39–47. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Wu and Preneel, 2013] Wu, H. and Preneel, B. (2013). Aegis: A fast authenticated encryption algorithm. Cryptology ePrint Archive, Report 2013/695. https://eprint.iacr.org/2013/695.

Appendix A

G&D-Automation Tool Results

$a_t =$	$z_t =$	n	G&D-resistance		Parallel.
			n eqs.	$2n$ eqs.	
$a_{t-n} + g(a_{t-n+1})$	$\sum_{k=0}^{n-1} a_{t+k}$	3	0		2
		4	0		3
		5	0		4
		6	0		5
		7	0		6
		8	0		7
		9	0		8
		10	0		9
$a_{t-n} + g(a_{t-n+1})$	$a_t + a_{t+\lceil \frac{n}{2} \rceil} + a_{t+n-1}$	3	0	0	2
		4	1	1	3
		5	1	1	4
		6	2	2	5
		7	2	1	6
		8	2	2	7
		9	2	-	8
$a_{t-n} + g(a_{t-n+2})$	$a_t + a_{t+\lceil \frac{n}{2} \rceil} + a_{t+n-1}$	3	0	0	1
		4	1	1	2
		5	1	1	3
		6	2	2	4
		7	2	1	5
		8	3	2	6
		9	4	-	7
$a_{t-n} + g(a_{t-n+1})$	$a_t + a_{t+\lceil \frac{n}{4} \rceil} + a_{t+\lceil \frac{3n}{4} \rceil} + a_{t+n-1}$	4	0	0	3
		5	1	1	4
		6	0	0	5
		7	1	1	6
		8	2	2	7
		9	3	-	8
		10	3	-	9

$a_t =$	$z_t =$	n	G&D-resistance		Parallel.
			n eqs.	$2n$ eqs.	
$a_{t-n} + g(a_{t-n+2})$	$a_t + a_{t+\lceil \frac{n}{4} \rceil} + a_{t+\lceil \frac{3n}{4} \rceil} + a_{t+n-1}$	4	0	0	2
		5	1	1	3
		6	0	0	4
		7	1	1	5
		8	2	2	6
		9	4	-	7
		10	3	-	8
$a_{t-n} + g(a_{t-n+1} + a_{t-n+2})$	$a_t + a_{t+n-1}$	3	2	2	1
		4	2	2	2
		5	2	2	3
		6	2	2	4
		7	2	2	5
		8	2	2	6
		9	2	-	7
$a_{t-n} + a_{t-n+1} + g(a_{t-n+2})$	$a_t + a_{t+n-1}$	3	2	2	1
		4	1	1	2
		5	2	2	3
		6	1	1	4
		7	2	2	5
		8	1	1	6
		9	2	-	7
$a_{t-n} + g(a_{t-n+1} + a_{t-n+2})$	$a_t + a_{t+\lceil \frac{n}{2} \rceil} + a_{t+n-1}$	3	0	0	1
		4	3	3	2
		5	2	2	3
		6	3	3	4
		7	3	3	5
		8	4	4	6
		9	3	-	7
$a_{t-n} + g(\sum_{k=1}^{n-1} a_{t-k})$	$a_t + a_{t+n-1}$	3	2	2	1
		4	2	2	1
		5	3	3	1
		6	4	-	1
		7	4	-	1
		8	4	-	1
$a_{t-n} + g(\sum_{k=1}^{n-1} a_{t-k})$	$\sum_{k=0}^{n-1} a_{t+k}$	3	1	1	1
		4	2	2	1
		5	3	3	1
		6	4	-	1
		7	4	-	1
		8	4	-	1

Table A.1: The results of the G&D-automation tool. The first two columns define the runaway generator. We then compute the G&D-resistance with the tool, using n and $2n$ equations. The dashes "-" indicate that we were not able to use the G&D-tool, due to it taking a long time (> 4 hours). We list the generator's parallelizability in the last column.