Rabbit: A New High-Performance Stream Cipher

Martin Boesgaard, Mette Vesterager,
Thomas Pedersen, Jesper Christiansen, and Ove Scavenius

CRYPTICO A/S
Fruebjergvej 3
2100 Copenhagen
Denmark
info@cryptico.com

Abstract. We present a new stream cipher, Rabbit, based on iterating a set of coupled non-linear functions. Rabbit is characterized by a high performance in software with a measured encryption/decryption speed of 3.7 clock cycles per byte on a Pentium III processor. We have performed detailed security analysis, in particular, correlation analysis and algebraic investigations. The cryptanalysis of Rabbit did not reveal an attack better than exhaustive key search.

Keywords: Stream cipher, fast, non-linear, coupled, counter, chaos

1 Introduction

Stream ciphers are an important class of symmetric encryption algorithms. Their basic design philosophy is inspired by the One-Time-Pad cipher, which encrypts by XOR'ing the plaintext with a random key. However, the need for a key of the same size as the plaintext makes the One-Time-Pad impractical for most applications. Instead, stream ciphers expand a given short random key into a pseudo-random keystream, which is then XOR'ed with the plaintext to generate the ciphertext. Consequently, the design goal for a stream cipher is to efficiently generate pseudo-random bits which are indistinguishable from truly random bits.

The aim of the present work is to design a secure stream cipher which is highly efficient in software.

1.1 The History Behind Rabbit

The design of Rabbit was inspired by the complex behavior of real-valued chaotic maps. Chaotic maps are primarily characterized by an exponential sensitivity to small perturbations causing iterates of such maps to seem random and long-time unpredictable. Those properties have also previously lead to suggestions that chaotic systems can be used for cryptographical purposes, see [1], [2] and references therein. However, even though chaotic systems exhibit random-like behavior, they are not necessarily cryptographically secure in their discretized form, see e.g. [3, 4]. The reason partly being that discretized chaotic functions do not automatically yield sufficiently complex behavior of the corresponding binary functions, which is a prerequisite for cryptographic security. It is therefore essential that the complexity of the binary functions is considered in the design phase such that necessary modifications can be made. Moreover, many suggested ciphers based on chaos suffer from reproducibility problems of the keystream due to the different handling of floating-point numbers on various processors, see e.g. [5].

The design goal of Rabbit was to take advantage of the random-like properties of real-valued chaotic maps and, at the same time, secure optimal cryptographic properties when discretizing them. More precisely, the design was initiated by constructing a chaotic system of coupled non-linear maps. This system was then restricted to be fixed-point valued[1]. This ensured reproducibility, and made the system analyzable from a binary point of view using well-known cryptographic techniques (see e.g. [7]). The analysis gave reason to some systematic improvements of the equation system, some of which were strictly binary in nature, e.g. adoption of rotations and the XOR operator. Those changes were advantageous for the complexity of the binary functions as well as the performance.

1.2 Rabbit in General

The Rabbit algorithm can briefly be described as follows. It takes a 128-bit secret key as input and generates for each iteration an output block of 128 pseudo-random bits from a combination of the internal state bits. The encryption/decryption is carried out by XOR'ing the pseudo-random data with the plaintext/ciphertext. The size of the internal state is 513 bits divided between eight 32-bit state variables, eight 32-bit counters and one counter carry bit. The eight state variables are updated by eight coupled non-linear integer valued functions. The counters secure a lower bound on the period length for the state variables.

The specific design goals of Rabbit were as follows:

- **Security:** The cipher should justify a key size of 128 bits for encrypting up to 2^{64} bytes of plaintext.
- **Speed:** It should be faster than commonly used ciphers.

1.3 Summary of Results

The cryptanalysis of Rabbit resulted in the following. To investigate the possibilities for Divide-and-Conquer and Guess-and-Determine types of attacks, an algebraic analysis was performed with special attention on the non-linear parts of the next-state function, as they are the main sources for mixing input bits.

[1] This means that each variable is represented by an integer type number, where a virtual decimal point is introduced manually, see [6] for details.

No such attacks better than exhaustive key search were found. To verify the resistance against correlation and distinguishing types of attacks, a correlation analysis was performed by calculating the Walsh-Hadamard spectra of the non-linear parts. Based on the correlation analysis we do not believe there exists a correlation-type attack, which requires less work than exhaustive key search for an output sequence shorter than 2^{64} bytes.

We measured an encryption/decryption speed of Rabbit of 3.7 clock cycles per byte on a Pentium III processor. For an ARM7 processor the measured performance was 10.5 clock cycles per byte.

1.4 Organization and Notation

In section two we describe the design of Rabbit in detail. We discuss the crypt-analysis of Rabbit in section three, and in section four the performance results are presented. We conclude and summarize in section five. Appendix A contains the ANSI C code for Rabbit. Note that the description below and the source code are specified for little-endian processors (e.g. most Intel processors). Appendix B contains test vectors. Appendix C discusses important properties of the counter system in detail.

We use the following notation: $\oplus$ denotes logical XOR, & denotes logical AND, $\ll$ and $\gg$ denote left and right logical bit-wise shift, $\lll$ and $\ggg$ denote left and right bit-wise rotation, and $\diamond$ denotes concatenation of two bit sequences. $A^{[g..h]}$ means bit number g through h of variable A. When numbering bits of variables, the least significant bit is denoted by 0. Hexadecimal numbers are prefixed by "0x". Finally, we use integer notation for all variables and constants.

2 The Design of Rabbit

In this section we provide a detailed description of the algorithm design.

2.1 The Cipher Algorithm

The internal state of the stream cipher consists of 513 bits. 512 bits are divided between eight 32-bit state variables $x_{j,i}$ and eight 32-bit counter variables $c_{j,i}$, where $x_{j,i}$ is the state variable of subsystem j at iteration i, and $c_{j,i}$ denote the corresponding counter variables. There is one counter carry bit, $\phi_{7,i}$, which needs to be stored between iterations. This counter carry bit is initialized to zero. The eight state variables and the eight counters are derived from the key at initialization.

Key Setup Scheme
The algorithm is initialized by expanding the 128-bit key into both the eight state variables and the eight counters such that there is a one-to-one correspondence between the key and the initial state variables, $x_{j,0}$, and the initial counters, $c_{j,0}$.

The key, $K^{[127..0]}$, is divided into eight subkeys: $k_0 = K^{[15..0]}$, $k_1 = K^{[31..16]}$, ..., $k_7 = K^{[127..112]}$. The state and counter variables are initialized from the subkeys as follows:

$$x_{j,0} = \begin{cases} k_{(j+1 \bmod 8)} \diamond k_j & \text{for } j \text{ even} \\ k_{(j+5 \bmod 8)} \diamond k_{(j+4 \bmod 8)} & \text{for } j \text{ odd} \end{cases} \tag{1}$$

and

$$c_{j,0} = \begin{cases} k_{(j+4 \bmod 8)} \diamond k_{(j+5 \bmod 8)} & \text{for } j \text{ even} \\ k_j \diamond k_{(j+1 \bmod 8)} & \text{for } j \text{ odd.} \end{cases} \tag{2}$$

The system is iterated four times, according to the next-state function defined below, to diminish correlations between bits in the key and bits in the internal state variables. Finally, the counter values are re-initialized according to:

$$c_{j,4} = c_{j,4} \oplus x_{(j+4 \bmod 8),4} \tag{3}$$

to prevent recovery of the key by inversion of the counter system.

Next-state Function
The core of the Rabbit algorithm is the iteration of the system defined by the following equations:

$$\begin{aligned}
x_{0,i+1} &= g_{0,i} + (g_{7,i} \lll 16) + (g_{6,i} \lll 16) \\
x_{1,i+1} &= g_{1,i} + (g_{0,i} \lll 8) + g_{7,i} \\
x_{2,i+1} &= g_{2,i} + (g_{1,i} \lll 16) + (g_{0,i} \lll 16) \\
x_{3,i+1} &= g_{3,i} + (g_{2,i} \lll 8) + g_{1,i} \\
x_{4,i+1} &= g_{4,i} + (g_{3,i} \lll 16) + (g_{2,i} \lll 16) \\
x_{5,i+1} &= g_{5,i} + (g_{4,i} \lll 8) + g_{3,i} \\
x_{6,i+1} &= g_{6,i} + (g_{5,i} \lll 16) + (g_{4,i} \lll 16) \\
x_{7,i+1} &= g_{7,i} + (g_{6,i} \lll 8) + g_{5,i}
\end{aligned} \tag{4}$$

$$g_{j,i} = \big((x_{j,i} + c_{j,i})^2 \oplus ((x_{j,i} + c_{j,i})^2 \gg 32)\big) \bmod 2^{32} \tag{5}$$

where all additions are modulo 2^{32}. This coupled system is schematically illustrated in Fig. 1. Before an iteration the counters are incremented as described below.

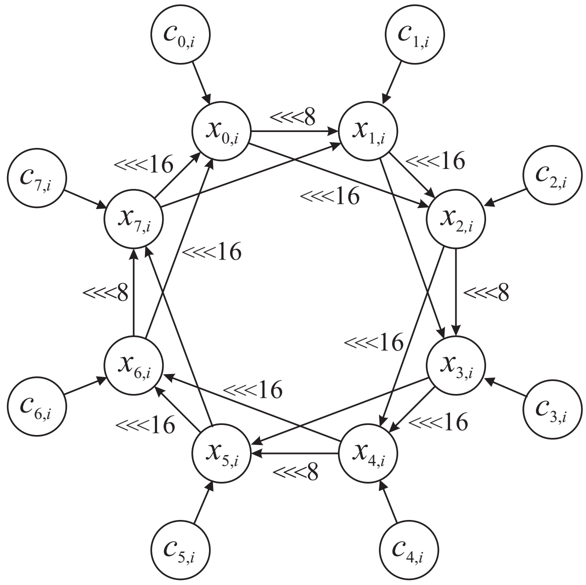

Fig. 1. Graphical illustration of the system.

Counter System
The dynamics of the counters is defined as follows:

$$\begin{aligned}
c_{0,i+1} &= c_{0,i} + a_0 + \phi_{7,i} \quad \bmod 2^{32} \\
c_{1,i+1} &= c_{1,i} + a_1 + \phi_{0,i+1} \bmod 2^{32} \\
c_{2,i+1} &= c_{2,i} + a_2 + \phi_{1,i+1} \bmod 2^{32} \\
c_{3,i+1} &= c_{3,i} + a_3 + \phi_{2,i+1} \bmod 2^{32} \\
c_{4,i+1} &= c_{4,i} + a_4 + \phi_{3,i+1} \bmod 2^{32} \\
c_{5,i+1} &= c_{5,i} + a_5 + \phi_{4,i+1} \bmod 2^{32} \\
c_{6,i+1} &= c_{6,i} + a_6 + \phi_{5,i+1} \bmod 2^{32} \\
c_{7,i+1} &= c_{7,i} + a_7 + \phi_{6,i+1} \bmod 2^{32}
\end{aligned} \tag{6}$$

where the counter carry bit, $\phi_{j,i+1}$, is given by

$$\phi_{j,i+1} = \begin{cases} 1 & \text{if } c_{0,i} + a_0 + \phi_{7,i} \geq 2^{32} \wedge j = 0 \\ 1 & \text{if } c_{j,i} + a_j + \phi_{j-1,i+1} \geq 2^{32} \wedge j > 0 \\ 0 & \text{otherwise.} \end{cases} \tag{7}$$

Furthermore, the a_j constants are defined as:

$$\begin{aligned} a_0 &= \text{0x4D34D34D} & a_1 &= \text{0xD34D34D3} \\ a_2 &= \text{0x34D34D34} & a_3 &= \text{0x4D34D34D} \\ a_4 &= \text{0xD34D34D3} & a_5 &= \text{0x34D34D34} \\ a_6 &= \text{0x4D34D34D} & a_7 &= \text{0xD34D34D3}. \end{aligned} \tag{8}$$

Extraction Scheme
After each iteration 128 bits of output are generated as follows:

$$\begin{aligned} s_i^{[15..0]} &= x_{0,i}^{[15..0]} \oplus x_{5,i}^{[31..16]} & s_i^{[31..16]} &= x_{0,i}^{[31..16]} \oplus x_{3,i}^{[15..0]} \\ s_i^{[47..32]} &= x_{2,i}^{[15..0]} \oplus x_{7,i}^{[31..16]} & s_i^{[63..48]} &= x_{2,i}^{[31..16]} \oplus x_{5,i}^{[15..0]} \\ s_i^{[79..64]} &= x_{4,i}^{[15..0]} \oplus x_{1,i}^{[31..16]} & s_i^{[95..80]} &= x_{4,i}^{[31..16]} \oplus x_{7,i}^{[15..0]} \\ s_i^{[111..96]} &= x_{6,i}^{[15..0]} \oplus x_{3,i}^{[31..16]} & s_i^{[127..112]} &= x_{6,i}^{[31..16]} \oplus x_{1,i}^{[15..0]} \end{aligned} \tag{9}$$

where s_i is the 128-bit keystream block at iteration i.

Encryption/decryption Scheme
The extracted bits are XOR'ed with the plaintext/ciphertext to encrypt/decrypt.

$$c_i = p_i \oplus s_i, \tag{10}$$

$$p_i = c_i \oplus s_i, \tag{11}$$

where c_i and p_i denote the i^{th} ciphertext and plaintext blocks, respectively.

3 Security Analysis

The security analysis is divided into six parts. First we discuss the key setup function and counter properties. We then perform an algebraic analysis of the next-state function, a correlation analysis of the binary functions and discuss statistical properties of Rabbit. In the last part the results of the investigations are used in specific types of attacks such as Guess-and-Determine, Divide-and-Conquer, Distinguishing and Correlation attacks.

3.1 Key Setup Properties

In this section we describe specific properties of the key setup scheme. The setup can be divided into three stages: Key expansion, system iteration and counter modification.

Key Expansion
In the key expansion stage we ensure two properties. The first one being a one-to-one correspondence between the key, the state and the counter, which prevents

key redundancy. The other property is that after one iteration of the next-state function, each key bit has affected all eight state variables. More precisely, for a given key bit there exists a j such that this key bit affects the output from $g_{j,0}$, $g_{(j+1 \bmod 8),0}$, $g_{(j+4 \bmod 8),0}$ and $g_{(j+5 \bmod 8),0}$. In each of the eight next-state subfunctions at least one of those g-functions enter.

System Iteration
The key expansion scheme ensures that after two iterations of the next-state function, all state bits are affected by all key bits with a measured probability of 0.5. A safety margin is provided by iterating the system four times.

Counter Modification
Even though the counters should be known to an attacker, the counter modification makes it hard to recover the key by inverting the counter system as this would require additional knowledge of the state variables. Due to the counter modification we cannot guarantee that every key results in unique counter values. However, we do not believe this to cause a problem as will be discussed later on.

3.2 Counter Properties

In this section we describe the dynamics of the counters, i.e. the period length and bit-flip probabilities of individual bit values.

Period Length
The most important feature of counter assisted stream ciphers [8] is that strict lower bounds on the period lengths can be provided. The adopted counter system in Rabbit has a period length of $2^{256}-1$. Since it can be shown that the input to the g-functions has at least the same period, a highly conservative lower bound on the period of the state variables, $N_{\mathrm{x}} > 2^{158}$, can be secured (see Appendix C).

Probabilities for Bit-flips in the Counters
For a 256-bit counter incremented by one, the period length for bit position i is 2^{i+1}. This implies that the least significant bit has a bit-flip probability of 1 and the most significant bit has a bit-flip probability of 2^{-255}. Consequently, the value of the most significant bit will remain constant for 2^{255} iterations, thereby making it very predictable.

In contrast, all bits in the counter defined in Eqs. (6) and (7), will have equal period length, as each bit is indirectly influenced by all other bits, due to the feedback of the carry, $\phi_{i,7}$ into the counter $c_{i,0}$. This implies that all bits have the same period length as the full system.

In Appendix C we calculate the bit-flip probabilities in the counter system. The most important findings are as follows. For the chosen a_j constants, Eq. (8), bit-flip probabilities for the individual bit positions are all in the interval $[0.17; 0.91]$. Furthermore, the probabilities are unique for each bit position. Since all the counter bits have full period and unique bit-flip probabilities, it seems difficult to predict bit patterns of the counter variables.

3.3 Algebraic Analysis

In this section we analyze a given output byte's dependence on its input bytes. The counter is ignored in the following. We first analyze the g-function defined by

$$g(y) = (y^2 \oplus (y^2 \gg 32)) \bmod 2^{32}. \quad (12)$$

By dividing y into four bytes, (a, b, c, d), we can write y^2 as:

$$(a \ll 24 + b \ll 16 + c \ll 8 + d)^2 = (a^2 \ll 48 + ab \ll 41 + ac \ll 33 + b^2 \ll 32 + ad \ll 25 + bc \ll 25 + bd \ll 17 + c^2 \ll 16 + cd \ll 9 + d^2). \quad (13)$$

The similar form of $g(y)$ follows directly from above. By collecting terms corresponding to each of the four $g(y)$ output bytes their dependencies on the four input bytes can be obtained. These dependencies are summarized in Table 1.

Table 1. The influences of the input bytes on the output bytes of the g-function. The subscripts, H,L, denotes the eight most,least significant bits of the 16-bit result for the multiplication of the two 8-bit numbers. For simplicity, carries from additions are ignored and the shifts are changed to nearest multiple of eight.

	$g(y)^{[31..24]}$	$g(y)^{[23..16]}$	$g(y)^{[15..8]}$	$g(y)^{[7..0]}$
$a = y^{[31..24]}$	$(ad)_\mathrm{L} + (a^2)_\mathrm{H}$	$(a^2)_\mathrm{L} + (ab)_\mathrm{H}$	$(ab)_\mathrm{L} + (ac)_\mathrm{H}$	$(ac)_\mathrm{L} + (ad)_\mathrm{H}$
$b = y^{[23..16]}$	$(bc)_\mathrm{L} + (bd)_\mathrm{H}$	$(bd)_\mathrm{L} + (ab)_\mathrm{H}$	$(ab)_\mathrm{L} + (b^2)_\mathrm{H}$	$(b^2)_\mathrm{L} + (bc)_\mathrm{H}$
$c = y^{[15..8]}$	$(bc)_\mathrm{L} + (c^2)_\mathrm{H}$	$(c^2)_\mathrm{L} + (cd)_\mathrm{H}$	$(cd)_\mathrm{L} + (ac)_\mathrm{H}$	$(ac)_\mathrm{L} + (bc)_\mathrm{H}$
$d = y^{[7..0]}$	$(ad)_\mathrm{L} + (bd)_\mathrm{H}$	$(bd)_\mathrm{L} + (cd)_\mathrm{H}$	$(cd)_\mathrm{L} + (d^2)_\mathrm{H}$	$(d^2)_\mathrm{L} + (ad)_\mathrm{H}$

To quantitatively examine a given output byte's dependence on its input bytes, we define an input mask function, $M_\mathrm{I}(y) = y \& m_\mathrm{I}$ and a similar output mask function $M_\mathrm{O}(y) = y \& m_\mathrm{O}$ where m_I and m_O are masks selecting specific byte-patterns. For all input values, y, we calculate

$$z = M_\mathrm{O}\big(g(M_\mathrm{I}(y)) \oplus g(y)\big). \quad (14)$$

This function characterizes the error in the output byte based on the input bytes defined by the mask m_I. We can define a measure for this error by calculating its corresponding entropy.

The specific investigation consisted of calculating the 16 entropies obtained by using all combinations of four 8-bit rotations of m_I =0x00FFFFFF and four 8-bit rotations of m_O =0x000000FF. The results are shown in Table 2.

The table shows the entropy of z for the 16 different byte-wise combinations. We clearly observe the expected behavior from Table 1. Hence, we conclude that

Table 2. The entropy of the error, maximally 8 bits, for an estimated output byte when removing a given input byte.

	$g(y)^{[31..24]}$	$g(y)^{[23..16]}$	$g(y)^{[15..8]}$	$g(y)^{[7..0]}$
$a = y^{[31..24]}$	7.99	7.99	7.99	7.99
$b = y^{[23..16]}$	7.99	7.99	7.99	7.99
$c = y^{[15..8]}$	7.99	7.99	7.99	7.99
$d = y^{[7..0]}$	7.99	7.99	7.99	7.98

all four output bytes of the g-function each depend on four input bytes. Removing any of those input bytes will result in nearly maximal entropy of the error of the output bytes. We also performed a similar analysis based on individual bits instead of individual bytes leading to similar conclusions.

Using the above results, we analyze the next-state subfunctions given by

$$f_{\text{even}}(y_1, y_2, y_3) = g(y_1) + (g(y_2) \lll 16) + (g(y_3) \lll 16), \tag{15}$$

and

$$f_{\text{odd}}(y_1, y_2, y_3) = g(y_1) + (g(y_2) \lll 8) + g(y_3). \tag{16}$$

Each function depends on three independent g-functions of which one or two have been rotated. Therefore, we can easily construct a table similar to Table 1 and use the results shown in Table 2 to obtain the corresponding entropies of the errors for the next-state function. Clearly, all output bytes of the next-state function depend on the maximal 12 input bytes. Consequently, removing any of those input bytes will result in nearly maximal entropy of error of the output bytes.

3.4 Linear Correlation Analysis

The aim of the correlation analysis is to find the best linear approximations between bits in the input to the next-state function and the extracted output.

Each of the eight next-state functions takes three 32-bit state variables and three 32-bit counter values as input and returns the corresponding updated 32-bit state variable. Each bit position in $x_{j,i+1}$ defines a binary function from $\{0,1\}^{192}$ to $\{0,1\}$. Thus, assuming that all 192 input bits are independently and uniformly distributed random variables, all correlations from output bits to linear combinations of input bits can be found via the Walsh-Hadamard Transform (WHT) [10, 11]. Clearly, we cannot numerically perform such a complete WHT of a 192-bit binary function. However, from analyzing the basic building block of the next-state function, i.e. the g-function, we obtain linear approximations of the cipher and their corresponding correlations coefficients. Note that all correlation coefficients are represented as absolute values.

The g-function

In the following we ignore the counter system and focus only on the correlation between the output of the g-function and its 32-bit input, $y \equiv x + c$. The WHTs of all single output bits of the g-function revealed that the largest correlation coefficients for all output bits of $g(y)$ are in the interval $[2^{-9.74}; 2^{-9.00}]$. Among those the best linear approximation is:

$$g^{[6]} \approx y^{[0]} \oplus y^{[3]} \oplus (y^{[5]} \oplus y^{[6]} \oplus \ldots \oplus y^{[16]} \oplus y^{[17]}) \oplus \\ (y^{[19]} \oplus y^{[20]} \oplus \ldots \oplus y^{[30]} \oplus y^{[31]}). \tag{17}$$

In general, linear approximations for linear combinations of binary functions can be obtained by a convolution of the involved WHT spectra, [11]. An exhaustive investigation of all 2^{32} possible convolutions of WHT spectra from the individual output bits in the g-function is not feasible. However, investigations of all convolutions of 16-, 18- and 20-bit g-functions show that the largest resulting correlation coefficients are of similar magnitude as the non-combined output bits and we expect the 32-bit g-function to behave similarly.

Non-Combined Output Bits

To determine linear approximations between the input to the next-state subfunctions and single output bits of the next-state subfunctions, we applied the following strategy. The next-state function includes the addition of three g-functions. To take those additions into account, we determined the best linear approximations for the function, $f(a, b, c) = a + b + c$, for each bit position. For instance, for each bit position $j \geqq 3$ the best linear approximations are:

$$f^{[j]} \approx a^{[j]} \oplus b^{[j]} \oplus c^{[j]} \tag{18}$$

$$f^{[j]} \approx a^{[j]} \oplus b^{[j]} \oplus c^{[j]} \oplus a^{[j-1]} \oplus b^{[j-1]} \tag{19}$$

$$f^{[j]} \approx a^{[j]} \oplus b^{[j]} \oplus c^{[j]} \oplus b^{[j-1]} \oplus c^{[j-1]} \tag{20}$$

$$f^{[j]} \approx a^{[j]} \oplus b^{[j]} \oplus c^{[j]} \oplus a^{[j-1]} \oplus c^{[j-1]}. \tag{21}$$

Their corresponding correlation coefficients are given by: $\sum_{n=1}^{j-1} 2^{-2n}$.

The next step is to substitute a, b and c by the corresponding binary functions of the g-function, i.e. $a^{[j]} \oplus a^{[j-1]} \rightarrow g^{[j]} \oplus g^{[j-1]}$. By determining the WHT spectra of the independent parts, we obtain linear approximations for the output bits of the next-state function. Each corresponding correlation coefficient is found by multiplying the product of the correlation coefficients for the independent parts with each correlation coefficient for the addition approximations. This results in a largest correlation coefficient of $2^{-28.6}$ for $f_{\text{even}}^{[1]}$. In the extraction function two bits from independent subsystems are XOR'ed, and the largest correlation coefficient can therefore be determined by the product of their largest individual correlation coefficients, yielding a largest correlation coefficient of $2^{-57.8}$ for $f_{\text{even}}^{[1]} \oplus f_{\text{odd}}^{[17]}$.

Linearly Combined Output-Bits

We assume that the best linear approximations for combined output bits are those that depend on the least number of g-functions. At the same time we assume that for a given number of g-function dependencies, the best approximations are those which include the fewest number of combinations of extracted output bits. To find these g-function dependencies, additions were replaced by XORs and the output from each g-function were divided into 8-bit blocks. Then an exhaustive search among all combinations of extracted output bytes were performed to find those with the least g-function dependencies. It was found that all combinations of output bytes depend on at least four different g-functions which can only be obtained by combining at least five extracted output bits. On the other hand, it was found that by combining two extracted output bits, the least number of g-function dependencies is five.

For instance, combining the extracted output $s^{[7..0]}$ and $s^{[127..120]}$ yields:

$$\begin{aligned} s^{[7..0]} \oplus s^{[127..120]} = &(g_0 + (g_7 \lll 16) + (g_6 \lll 16))^{[7..0]} \oplus \\ &(g_5 + (g_4 \lll 8) + g_3)^{[23..16]} \oplus \\ &(g_1 + (g_0 \lll 8) + g_7)^{[15..8]} \oplus \\ &(g_6 + (g_4 \lll 16) + (g_5 \lll 16))^{[31..24]} \end{aligned} \tag{22}$$

and using Eq. (18) for each parenthesis, i.e. replacing addition with XOR we obtain:

$$\begin{aligned} s^{[7..0]} \oplus s^{[127..120]} \approx\ &g_1^{[15..8]} \oplus g_3^{[23..16]} \oplus g_5^{[15..8]} \oplus g_5^{[23..16]} \oplus \\ &g_6^{[23..16]} \oplus g_6^{[31..24]} \oplus g_7^{[15..8]} \oplus g_7^{[23..16]} \end{aligned} \tag{23}$$

which depends on five different g-functions. The largest corresponding correlation coefficient is $2^{-59.8}$. All other combinations of two output bits depending on five g-functions have smaller correlation coefficients.

An example of a linear approximation that only depends on four g-functions is:

$$\begin{aligned} s^{[7..0]} \oplus s^{[23..16]} \oplus s^{[79..72]} \oplus s^{[55..48]} \oplus s^{[111..104]} \approx\ &g_3^{[23..16]} \oplus g_5^{[7..0]} \oplus g_5^{[23..16]} \oplus \\ &g_5^{[31..24]} \oplus g_6^{[7..0]} \oplus g_6^{[15..8]} \oplus \\ &g_6^{[23..16]} \oplus g_7^{[7..0]} \oplus g_7^{[31..24]} \oplus \\ &g_7^{[23..16]}. \end{aligned} \tag{24}$$

with a largest correlation coefficient of $2^{-59.2}$. All other byte-wise combinations of five output bits depending on four g-functions have smaller correlation coefficients.

3.5 Statistical Tests

The statistical tests on Rabbit were performed using the NIST Test Suite [12], the DIEHARD battery of tests [13] and the ENT test [14]. Tests were performed on the internal state as well as on the extracted output. Furthermore, we also conducted various statistical tests on the key setup function. Finally, we performed the same tests on a version of Rabbit where each state variable and counter variable was only 8 bit wide. No weaknesses were found in any case.

3.6 Resulting Attacks

This subsection discusses relevant attacks based on the above analysis.

Attacks on the Key Setup Function
Due to the four iterations after key expansion and the final counter modification, both the counter bits and the state bits depend strongly and highly non-linearly on the key bits. This makes attacks based on guessing parts of the key difficult. Furthermore, even if the counter bits were known after the counter modification, it is still hard to recover the key. Of course, knowing the counters makes other types of attacks easier.

As the non-linear map in Rabbit is many-to-one, different keys could potentially result in the same keystream. This concern can basically be reduced to the question whether different keys result in the same counter values, since different counter values must necessarily lead to different keystreams. The reason is that when the periodic solution has been reached, the next-state function, including the counter system, is one-to-one on the set of points in the period. The key expansion scheme was designed such that each key leads to unique counter values. However, the counter modification might result in equal counter values for two different keys. Assuming that the output after the four initial iterations is essentially random and not correlated with the counter system, the probability for counter collisions is essentially given by the birthday paradox, i.e. for all 2^{128} keys one collision is expected in the 256-bit counter state. Thus, we do not believe counter collisions to cause a problem. Another possibility for related key attacks is to exploit the symmetries of the next-state and key setup functions. For instance, consider two keys, K and $\tilde{K}$ related by $K^{[i]} = \tilde{K}^{[i+32]}$ for all i. This leads to the relation, $x_{j,0} = \tilde{x}_{j+2,0}$ and $c_{j,0} = \tilde{c}_{j+2,0}$. If the a_j constants were related in the same way, the next-state function would preserve this property. In the same way this symmetry could lead to a set of bad keys, i.e. if $K^{[i]} = K^{[i+32]}$ for all i, then $x_{j,0} = x_{j+2,0}$ and $c_{j,0} = c_{j+2,0}$. However, the next-state function does not preserve this property due to the counter system as $a_j \neq a_{j+2}$.

Divide-and-Conquer Attack
This type of attack is feasible if only a part of the state needs to be known in order to predict a significant fraction of the output bits. An attacker will guess a part of the state, predict the output bits and compare them with actually observed output bits. Our strategy is to accurately predict one extracted output byte based on guessing as few input bytes as possible.

According to section 3.3 the attacker must guess $2 \cdot 12$ input bytes for the different g-functions. Thus, 192 bits in total must be guessed. Furthermore, we have verified that calculating less extracted bits than a byte still results in more work than exhaustive key search. Finally, when replacing all additions by XORs, all byte-wise combinations of the extracted output still depend on at least four different g-functions, see section 3.4. To conclude, it is not possible to verify a guess on fewer bits than the key size.

Guess-and-Determine Attack

The strategy for this attack is to guess a few of the unknown variables of the cipher and from those deduce the remaining unknowns. For simplicity, we assume that the counters are static.

A simple attack of this type consists of guessing the remaining 128 bits of the internal state from the extracted 128 bits for each of two consecutive iterations. This amounts to guessing the remaining $128 + 128$ bits and derive the counter values. Each of the resulting systems must then be iterated a couple of times to verify the output.

However, in the above attack it is assumed that no advantage is gained by dividing the counters and state variables into smaller blocks. An attack exploiting this possibility can be formulated as follows. Divide the 32-bit state variables and counters into 8-bit variables. Construct an equation system consisting of the $8 \cdot 4$ 8-bit subsystems for N iterations together with the corresponding $(N+1) \cdot 8$ extraction functions which are split into $(N + 1) \cdot 16$ 8-bit functions. In order to obtain a closed system of equations, output from $4 \cdot 8$ extraction functions is needed, i.e. $N = 3$. Thus, the equation system consists of 160 coupled equations with $8 \cdot 4$ unknown counter bytes and $(3 + 1) \cdot 8 \cdot 4$ unknown state bytes, i.e. a total of 160 unknowns.

A strategy for solving this equation system must be found by guessing as few input bytes as possible and determining the remaining unknown bytes. The efficiency of such a strategy depends on the amount of variables that must be guessed before the determining process can begin. This amount is given by the 8-bit subsystem with the fewest number of input variables. Neglecting the counters, the results of section 3.3 illustrate that each byte of the next-state function depends on 12 input bytes. When the counters are included, each output byte of a subsystem depends on 24 input bytes. Consequently, the attacker must guess more than 128 bits before the determining process can begin, thus, making the attack infeasible. Dividing the system into smaller blocks than bytes results in the same conclusion.

Distinguishing and Correlation Attacks

In case of a distinguishing attack the attacker tries to distinguish a sequence generated by the cipher from a sequence of truly random numbers. A distinguishing attack using less than 2^{64} bytes of output cannot be applied using only the best linear approximation found in section 3.4 because the corresponding correlation coefficient is $2^{-57.8}$. This implies that in order to observe this particular correlation, output from 2^{114} iterations must be generated [9].

The independent counters have very simple and almost linear dynamics. Therefore, large correlations to the counter bits may cause a possibility for a correlation attack (see e.g. [15]) for recovering the counters. It is not feasible to exploit only the best linear approximation in order to recover a counter value. However, more correlations to the counters could be exploited. As this requires that there exists many such large and useable correlations, we do not believe such an attack to be feasible. Knowing the values of the counters may significantly improve both the Guess-and-Determine attack, the Divide-and-Conquer attack as well as a Distinguishing attack even though obtaining the key from the counter values is prevented by the counter modification in the setup function.

4 Performance

In this section we provide performance results from implementations of Rabbit on 32-bit processors and discuss 8-bit implementation aspects.

4.1 32-bit Processors

Encryption speeds for the specific processors were obtained by encrypting 200 kilobytes of data stored in RAM and measuring the number of clock cycles passed. Memory requirements and performance results are listed in the tables below. For convenience, all 513 bits of the internal state are stored in an instance structure, occupying a total of 68 bytes. The presented memory requirements show the amount of memory allocated on the stack for calling conventions (function arguments, return address and preserved registers) and temporary data. No memory requirements for storing the key, the instance, the ciphertext and the plaintext have been included.

Intel Pentium III Architecture
The performance was measured on a 1000 MHz Pentium III processor. The speed-optimized version of Rabbit was programmed in assembly language inlined in C using MMX instructions and compiled using the Intel C++ 7.0 compiler. The results are listed in Table 3 below. A memory-optimized version (where calling conventions are ignored) eliminates the need for memory, including the instance structure, since the entire instance structure and temporary data can fit into the CPU registers.

Table 3. Code size, memory requirements and performance for Pentium III.

Function	Code size	Memory	Performance
Key Setup	617 bytes	32 bytes	350 cycles
Encryption/Decryption	440 bytes	36 bytes	3.7 cycles/byte

ARM7 Architecture
A speed optimized ARM implementation was compiled and tested using ARM Developer Suite version 1.2 for ARM7TDMI. Performance was measured using the integrated ARMulator. The performance results, code size and memory requirements are listed in Table 4 below:

Table 4. Code size, memory requirements and performance for ARM7.

Function	Code size	Memory	Performance
Key Setup	516 bytes	44 bytes	679 cycles
Encryption/Decryption	424 bytes	52 bytes	10.5 cycles/byte

4.2 8-bit Processors

The simplicity and small size of Rabbit makes it suitable for implementation for processors with limited resources such as 8-bit microcontrollers. Multiplying 32-bit integers is rather resource demanding using plain 32-bit arithmetics. However, as seen in Eq. (13) in section 3.3, squaring involves only ten 8-bit multiplications which reduces the workload by approximately a factor of two. Finally, the rotations in the algorithm have been chosen to correspond to simple byte-swapping.

5 Conclusion

In this paper we presented a new stream cipher called Rabbit. A complete description of the algorithm, an evaluation of its security properties, performance and implementation aspects were given. Our most important findings include the following: In terms of security, Guess-and-Determine attacks, Divide-and-Conquer attacks as well as Distinguishing and Correlation attacks were considered, but no attack better than exhaustive key search was found. The measured encryption/decryption performance was 3.7 clock cycles per byte on a Pentium III processor and 10.5 clock cycles per byte on an ARM7 processor.

Acknowledgements

The authors would like to thank Vincent Rijmen for several ideas and suggestions. Furthermore, we thank Ivan Damgaard and Tomas Bohr for many helpful inputs. Finally, we would like to thank Anne Canteaut and the anonymous reviewers for helpful feedback.

References

1. S. Wolfram, *Cryptography with Cellular Automata*, Proceedings of Crypto '85, pp. 429-432 (1985)
2. G. Jakimoski and L. Kocarev: *Chaos and Cryptography: Block Encryption Ciphers Based on Chaotic Maps*, IEEE Transactions on Circuits and Systems-1: Fundamental Theory and Applications, Vol. 48, NO. 2, pp 163-169 (2001)
3. T. Habutso, Y. Nishio, I. Sasase and S. Mori: *A secret key cryptosystem by iterating a chaotic map*, Proceedings of the EUROCRYPT '91, Springer, Berlin, pp. 127-140 (1991)

4. E. Biham: *Cryptoanalysis of the chaotic-map cryptosystem suggested at EUROCRYPT '91*, Proceedings of the EUROCRYPT '91, Springer, Berlin, pp. 532-534 (1991)
5. R. Matthews: *On the Derivation of a "Chaotic" Encryption Algorithm*, Cryptology Vol. XIII NO. 1, pp. 29-41 (1989)
6. R. Yates, *Fixed-Point Arithmetic: An Introduction*, http://personal.mia.bellsouth.net/lig/y/a/yatesc/fp.pdf
7. A. Menezes, P. Oorschot and S. Vanstone: *Handbook of Applied Cryptography*, CRC Press LLC (1997).
8. A. Shamir and B. Tsaban: *Guaranteeing the Diversity of Number Generators*, Information and Computation Vol. 171, No. 2, pp. 350-363 (2001)
9. M. Matsui, *Linear Cryptanalysis Method for DES Cipher*, Advances in Cryptology – EUROCRYPT '93. pp. 386-397 (1993).
10. R. A. Rueppel: *Analysis and Design of Stream Ciphers*, Springer, Berlin (1986).
11. J. Daemen, *Chapter 5: Propagation and Correlation*, Annex to AES Proposal (1998).
12. *A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications*, NIST Special Publication 800-22, National Institute of Standards and Technology, 2001, http://csrc.nist.gov/rng
13. G. Masaglia, *A battery of tests for random number generators*, Florida State University, http://stat.fsu.edu/ geo/diehard.html
14. J. Walker, *A Pseudorandom Number Sequence Test Program*, http://www.fourmilab.ch/random
15. W. Meier and O. Staffelbach, *Fast Correlation Attacks on Stream Ciphers*, Advances in Cryptology-EUROCRYPT (LNCS 330), pp. 301-314, (1988).

A ANSI C Source Code

This appendix presents the ANSI C source code for Rabbit.

rabbit.h

Below the rabbit.h header file is listed:

```
#ifndef _RABBIT_H
#define _RABBIT_H

#include <stddef.h>

// Type declarations of 32-bit and 8-bit unsigned integers
typedef unsigned int uint32;
typedef unsigned char byte;

// Structure to store the instance data (internal state)
typedef struct
{
   uint32 x[8];
   uint32 c[8];
   uint32 carry;
} t_instance;

void key_setup(t_instance *p_instance, const byte *p_key);
void cipher(t_instance *p_instance, const byte *p_src,
            byte *p_dest, size_t data_size);

#endif
```

rabbit.c

In the C file, rabbit.c, the logical rotation function, `_rotl`, is used, however, for some compilers it may not be defined. In such case, the logical rotation function can be defined as:

```
uint32 _rotl(uint32 x, int rot) { return (x<<rot) | (x>>(32-rot)); }
```

Below the rabbit.c file is listed:

```
#include <stdlib.h>
#include "rabbit.h"

// Square a 32-bit number to obtain the 64-bit result and return
// the upper 32 bit XOR the lower 32 bit
uint32 g_func(uint32 x)
{
   // Construct high and low argument for squaring
   uint32 a = x&0xFFFF;
```

```
   uint32 b = x>>16;

   // Calculate high and low result of squaring
   uint32 h = ((((a*a)>>17) + (a*b))>>15) + b*b;
   uint32 l = x*x;

   // Return high XOR low;
   return h^l;
}


// Calculate the next internal state
void next_state(t_instance *p_instance)
{
   // Temporary data
   uint32 g[8], c_old[8], i;

   // Save old counter values
   for (i=0; i<8; i++)
      c_old[i] = p_instance->c[i];

   // Calculate new counter values
   p_instance->c[0] += 0x4D34D34D + p_instance->carry;
   p_instance->c[1] += 0xD34D34D3 + (p_instance->c[0] < c_old[0]);
   p_instance->c[2] += 0x34D34D34 + (p_instance->c[1] < c_old[1]);
   p_instance->c[3] += 0x4D34D34D + (p_instance->c[2] < c_old[2]);
   p_instance->c[4] += 0xD34D34D3 + (p_instance->c[3] < c_old[3]);
   p_instance->c[5] += 0x34D34D34 + (p_instance->c[4] < c_old[4]);
   p_instance->c[6] += 0x4D34D34D + (p_instance->c[5] < c_old[5]);
   p_instance->c[7] += 0xD34D34D3 + (p_instance->c[6] < c_old[6]);
   p_instance->carry = (p_instance->c[7] < c_old[7]);

   // Calculate the g-functions
   for (i=0;i<8;i++)
      g[i] = g_func(p_instance->x[i] + p_instance->c[i]);

   // Calculate new state values
   p_instance->x[0] = g[0] + _rotl(g[7],16) + _rotl(g[6],16);
   p_instance->x[1] = g[1] + _rotl(g[0], 8) + g[7];
   p_instance->x[2] = g[2] + _rotl(g[1],16) + _rotl(g[0],16);
   p_instance->x[3] = g[3] + _rotl(g[2], 8) + g[1];
   p_instance->x[4] = g[4] + _rotl(g[3],16) + _rotl(g[2],16);
   p_instance->x[5] = g[5] + _rotl(g[4], 8) + g[3];
   p_instance->x[6] = g[6] + _rotl(g[5],16) + _rotl(g[4],16);
   p_instance->x[7] = g[7] + _rotl(g[6], 8) + g[5];
```

```
}


// key_setup
void key_setup(t_instance *p_instance, const byte *p_key)
{
   // Temporary data
   uint32 k0, k1, k2, k3, i;

   // Generate four subkeys
   k0 = *(uint32*)(p_key+ 0);
   k1 = *(uint32*)(p_key+ 4);
   k2 = *(uint32*)(p_key+ 8);
   k3 = *(uint32*)(p_key+12);

   // Generate initial state variables
   p_instance->x[0] = k0;
   p_instance->x[2] = k1;
   p_instance->x[4] = k2;
   p_instance->x[6] = k3;
   p_instance->x[1] = (k3<<16) | (k2>>16);
   p_instance->x[3] = (k0<<16) | (k3>>16);
   p_instance->x[5] = (k1<<16) | (k0>>16);
   p_instance->x[7] = (k2<<16) | (k1>>16);

   // Generate initial counter values
   p_instance->c[0] = _rotl(k2,16);
   p_instance->c[2] = _rotl(k3,16);
   p_instance->c[4] = _rotl(k0,16);
   p_instance->c[6] = _rotl(k1,16);
   p_instance->c[1] = (k0&0xFFFF0000) | (k1&0xFFFF);
   p_instance->c[3] = (k1&0xFFFF0000) | (k2&0xFFFF);
   p_instance->c[5] = (k2&0xFFFF0000) | (k3&0xFFFF);
   p_instance->c[7] = (k3&0xFFFF0000) | (k0&0xFFFF);

   // Reset carry flag
   p_instance->carry = 0;

   // Iterate the system four times
   for (i=0;i<4;i++)
      next_state(p_instance);

   // Modify the counters
   for (i=0;i<8;i++)
      p_instance->c[(i+4)&0x7] ^= p_instance->x[i];
```

```
}


// Encrypt or decrypt a block of data
void cipher(t_instance *p_instance, const byte *p_src,
            byte *p_dest, size_t data_size)
{
   uint32 i;

   for (i=0; i<data_size; i+=16)
   {
      // Iterate the system
      next_state(p_instance);

      // Encrypt 16 bytes of data
      *(uint32*)(p_dest+ 0) = *(uint32*)(p_src+ 0) ^
                              p_instance->x[0] ^
                              (p_instance->x[5]>>16) ^
                              (p_instance->x[3]<<16);
      *(uint32*)(p_dest+ 4) = *(uint32*)(p_src+ 4) ^
                              p_instance->x[2] ^
                              (p_instance->x[7]>>16) ^
                              (p_instance->x[5]<<16);
      *(uint32*)(p_dest+ 8) = *(uint32*)(p_src+ 8) ^
                              p_instance->x[4] ^
                              (p_instance->x[1]>>16) ^
                              (p_instance->x[7]<<16);
      *(uint32*)(p_dest+12) = *(uint32*)(p_src+12) ^
                              p_instance->x[6] ^
                              (p_instance->x[3]>>16) ^
                              (p_instance->x[1]<<16);

      // Increment pointers to source and destination data
      p_src += 16;
      p_dest += 16;
   }
}
```

B Test Vectors

The keys and outputs are presented byte-wise. The leftmost byte of `key` is $K^{[7..0]}$.

```
key   =  [00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00]

s[0]  =  [02 F7 4A 1C 26 45 6B F5 EC D6 A5 36 F0 54 57 B1]
s[1]  =  [A7 8A C6 89 47 6C 69 7B 39 0C 9C C5 15 D8 E8 88]
```

```
s[31] =  [EF 9A 69 71 8B 82 49 A1 A7 3C 5A 6E 5B 90 45 95]


key   =  [C2 1F CF 38 81 CD 5E E8 62 8A CC B0 A9 89 0D F8]

s[0]  =  [3D 02 E0 C7 30 55 91 12 B4 73 B7 90 DE E0 18 DF]
s[1]  =  [CD 6D 73 0C E5 4E 19 F0 C3 5E C4 79 0E B6 C7 4A]
s[31] =  [9F B4 92 E1 B5 40 36 3A E3 83 C0 1F 9F A2 26 1A]


key   =  [1D 27 2C 6A 2D 8E 3D FC AC 14 05 6B 78 D6 33 A0]

s[0]  =  [A3 A9 7A BB 80 39 38 20 B7 E5 0C 4A BB 53 82 3D]
s[1]  =  [C4 42 37 99 C2 EF C9 FF B3 A4 12 5F 1F 4C 99 A8]
s[31] =  [97 C0 73 3F F1 F1 8D 25 6A 59 E2 BA AB C1 F4 F1]
```

C Counter Properties

In this appendix we discuss important properties of the counter system.

Counter Period

The period of the counter system is $N_\mathrm{c} = 2^{256} - 1$. This can be shown as follows. The counter system defined in Eqs. (6) and (7) can equivalently be described by the following recurrence relation:

$$C_{i+1} = \big(C_i + A + \Phi_i\big) \bmod 2^{256}, \tag{25}$$

where Φ_{i+1} is defined as:

$$\Phi_{i+1} = \begin{cases} 1 & \text{if } C_i + A + \Phi_i \geq 2^{256} \\ 0 & \text{otherwise.} \end{cases} \tag{26}$$

C_i is a 256-bit integer obtained by concatenating all eight individual counters, i.e. $C_i = c_{7,i} \diamond c_{6,i} \diamond c_{5,i} \diamond c_{4,i} \diamond c_{3,i} \diamond c_{2,i} \diamond c_{1,i} \diamond c_{0,i}$, and A is likewise obtained by concatenating the eight a_j constants.

The above recurrence relation is equivalent to the following linear congruential generator:

$$Z_{i+1} = (Z_i + A) \bmod (2^{256} - 1), \tag{27}$$

which has a period length of $N_\mathrm{c} = 2^{256} - 1$, since A has been chosen such that $\gcd(A, 2^{256} - 1) = 1$.

To show that Z is equivalent to C, we consider an initial value $C_0 = Z_0$ for $Z_0 > A$. The recurrence relation for C_i can be defined in terms of Z_i:

$$C_i = \begin{cases} Z_i & \text{if } (Z_{i-1} + A) < 2^{256} - 1 \ \wedge \ Z_{i-1} \neq 0 \\ 2^{256} - 1 & \text{if } (Z_{i-1} + A) = 2^{256} - 1 \\ Z_i - 1 & \text{if } (Z_{i-1} + A) > 2^{256} - 1 \ \vee \ Z_{i-1} = 0. \end{cases} \tag{28}$$

Therefore, C_i will run through the same set of numbers as Z_i except that C_i will attain the value $2^{256}-1$ but not the value A. Thus, the period of the recurrence relation, C, is the same as for the linear congruential generator, Z. In particular, $C_i \neq C_j$ if $i-j \bmod N_\mathrm{c} \neq 0$.

Internal State Period

For convenience, we write the next-state function in the following way

$$\vec{x}_{i+1} = \overrightarrow{F(\vec{y_i})} \bmod 2^{32}, \tag{29}$$

where

$$\vec{y_i} = (\vec{c}_i + \vec{x}_i) \bmod 2^{32}, \tag{30}$$

such that $\vec{x}_i$ is the internal state variable and $\vec{c}_i$ is the counter state.

According to a generalized version of lemma 4.1 in [8], $\vec{y}_i$ will have at least the period of the counter system, N_c:

Proof. Given that $\vec{y}_i = \vec{y}_j$ for $i-j \bmod N_c \neq 0$, then $\vec{y}_{i+1} = \overrightarrow{F(\vec{y}_i)} + \vec{c}_i$ and $\vec{y}_{j+1} = \overrightarrow{F(\vec{y}_j)} + \vec{c}_j$. Moreover, we have: $\vec{c}_i \neq \vec{c}_j$, therefore, $\vec{y}_{i+1} \neq \vec{y}_{j+1}$. Finally, if $\vec{y}_{i-1} = \vec{y}_{j-1}$ this would imply that $\vec{y}_i \neq \vec{y}_j$ which is a contradiction. Thus, also $\vec{y}_{i-1} \neq \vec{y}_{j-1}$ ■

However, a combination of the internal state, $\vec{x}_i$, is extracted as output. It is not evident that $\vec{x}_i$ will have the same period as the counter system, but a lower bound for that period is obtained in the following.

First, we note that there are relations between the counter period, N_c, the internal state period, N_x and the period of the y variables, N_y:

$$N_\mathrm{y} = aN_\mathrm{x} = bN_\mathrm{c} \tag{31}$$

where a and b are integers greater than zero with $\gcd(a,b)=1$.

Proof. Since $\vec{x}_{i+1} = \overrightarrow{F(\vec{y_i})}$, we have $N_\mathrm{x} \leqq N_\mathrm{y}$. In particular, N_x divides N_y, because, if we assume that this is not the case, then there would exist an i such that $\overrightarrow{F(\vec{y_i})} = \vec{x}_{i+1} \neq \vec{x}_{i+1+\frac{N_\mathrm{y}}{N_\mathrm{x}}N_\mathrm{x}} = \overrightarrow{F(\vec{y}_{i+\frac{N_\mathrm{y}}{N_\mathrm{x}}N_\mathrm{x}})}$ which contradicts the N_y periodicity. Thus, there exists an integer, $a>0$, such that $N_\mathrm{y} = aN_\mathrm{x}$. We also have that N_c divides N_y because if this was not the case then $\vec{c}_i \neq \vec{c}_{i+\frac{N_\mathrm{y}}{N_\mathrm{c}}N_\mathrm{c}}$. We just showed that $\vec{x}_i = \vec{x}_{i+N_\mathrm{y}}$ for all i, but $\vec{y}_i = \vec{x}_i + \vec{c}_i \neq \vec{x}_{i+\frac{N_\mathrm{y}}{N_\mathrm{c}}N_\mathrm{c}} + \vec{c}_{i+\frac{N_\mathrm{y}}{N_\mathrm{c}}N_\mathrm{c}} = \vec{y}_{i+N_\mathrm{y}}$ which again contradicts the N_y periodicity. Therefore, there exists an integer, $b>0$ such that $N_\mathrm{y} = bN_\mathrm{c}$ and consequently, $N_\mathrm{y} = aN_\mathrm{x} = bN_\mathrm{c}$ ■

We have the relation: $N_\mathrm{x} = \frac{b}{a}N_\mathrm{c}$. Thus, we want to find an upper bound on the ratio, a/b. This can be done as follows. Define the degeneracy d to be the maximal number of pre-images $\vec{x}_{i+1}$ can have, i.e. d is the maximal number of different $\vec{y}_i$ which give the same $\vec{x}_{i+1}$ and similarly, define d_g to be the analogue for each g function. Then we can obtain the following rather conservative lower bound for the period:

Let $(\vec{x}_0, \vec{x}_1, \vec{x}_2, ..., \vec{x}_{N_\mathrm{x}-1})$ be a periodic sequence with period N_x, then the upper bound on a/b is the degeneracy d, i.e.:

$$N_\mathrm{x} \geqq \frac{N_\mathrm{c}}{d}, \tag{32}$$

where N_c is the counter period.

Proof. We want to show that $k \equiv \frac{a}{b} = \frac{N_\mathrm{c}}{N_\mathrm{x}} \leqq d$. The periodicity gives: $\vec{x}_i = \vec{x}_{i+N_\mathrm{x}} = \vec{x}_{i+2N_\mathrm{x}} = ... = \vec{x}_{i+(k-1)N_\mathrm{x}}$. On the other hand, the corresponding counter values are non-equal: $\vec{c}_i \neq \vec{c}_{i+N_\mathrm{x}} \neq \vec{c}_{i+2N_\mathrm{x}} \neq ... \neq \vec{c}_{i+(k-1)N_\mathrm{x}}$. Therefore, it follows: $\vec{x}_i + \vec{c}_i \neq \vec{x}_{i+N_\mathrm{x}} + \vec{c}_{i+N_\mathrm{x}} \neq \vec{x}_{i+2N_\mathrm{x}} + \vec{c}_{i+2N_\mathrm{x}} \neq ... \neq \vec{x}_{i+(k-1)N_\mathrm{x}} + \vec{c}_{i+(k-1)N_\mathrm{x}}$ or equivalently: $\vec{y}_i \neq \vec{y}_{i+N_\mathrm{x}} \neq \vec{y}_{i+2N_\mathrm{x}} \neq ... \neq \vec{y}_{i+(k-1)N_\mathrm{x}}$. Because of the periodicity we have $\overrightarrow{F(\vec{y}_i)} = \overrightarrow{F(\vec{y}_{i+N_\mathrm{x}})} = \overrightarrow{F(\vec{y}_{i+2N_\mathrm{x}})} = ... = \overrightarrow{F(\vec{y}_{i+(k-1)N_\mathrm{x}})}$. Since each $\vec{x}_{i+1}$ maximally can have d pre-images, we see that $k = \frac{a}{b} = \frac{N_\mathrm{c}}{N_\mathrm{x}} \leqq d$ ■

To illustrate that the period length is sufficiently large, consider the equation system, $\vec{x}_{i+1} = \overrightarrow{F_\mathrm{I}(\vec{x}_i)}$ arising by replacing all the g-functions by identity functions, but keeping the rotations. Fixing any two of the 32-bit input variables, the resulting equation system has a unique output for the remaining six input variables. Therefore, $\overrightarrow{F_\mathrm{I}(\vec{x})}$ is maximally 2^{64}-to-one. This bound can be combined with the measured degeneracy for the g-function, $d_\mathrm{g} = 18$, to obtain $d < 2^{64} \cdot 18^8 < 2^{98}$ which shows that the period length of the state variables is sufficiently large, i.e. $N_\mathrm{x} \geqq (2^{256} - 1)/d > 2^{158}$.

This bound is, of course, highly underestimated. For instance, the $\vec{F_\mathrm{I}}$ map will probably have degeneracy close to one. Furthermore, all points in the periodic solution should have the maximal degeneracy, d, and they should appear in exact synchronization with the counter. So if the output of $\vec{F}$ is not correlated strongly with the counter sequence, the probability for actually realizing this lower bound is vanishing. Furthermore, for the specific g-function only one point have a maximal degeneracy of 18 and about half of the points have degeneracy one. It also follows from above that if a point with a degeneracy one belongs to the periodic solution then the period cannot be shorter than the counter period.

Bit-flip Probabilities

Below we calculate the bit-flip probabilities for the counter bits.

Let the bit-wise carry $\Phi^{[j\boxplus 1]}$ from bit position j to bit position $j \boxplus 1$ be defined as:

$$\Phi^{[j\boxplus 1]} = \begin{cases} 1 & \text{if } C^{[j]} + A^{[j]} + \Phi^{[j]} \geq 2 \\ 0 & \text{otherwise} \end{cases} \tag{33}$$

where $x \boxplus y \equiv x + y \bmod 256$ and C and A are defined above. The value of $C^{[j]}$ only changes when either $\Phi^{[j]} = 1$ and $A^{[j]} = 0$ or $\Phi^{[j]} = 0$ and $A^{[j]} = 1$. The probability of the carry can be found by solving a system of recursive equations for carry probability as is shown in the following.

The probability for carry from bit position j is given by:

$$\mathrm{P}\big(\Phi^{[j]} = 1\big) = \frac{A^{[j]} + \mathrm{P}\big(\Phi^{[j\boxminus 1]} = 1\big)}{2} \tag{34}$$

where $x \boxminus y \equiv x - y \bmod 256$. Inserting the same expression for $\mathrm{P}\big(\Phi^{[j\boxminus 1]} = 1\big)$ into this equation we obtain

$$\mathrm{P}\big(\Phi^{[j]} = 1\big) = \frac{A^{[j]}}{2^1} + \frac{A^{[j\boxminus 1]} + \mathrm{P}\big(\Phi^{[j\boxminus 2]} = 1\big)}{2^2}. \tag{35}$$

Continuing like this we get

$$\mathrm{P}\big(\Phi^{[j]} = 1\big) = \frac{A^{[j\boxminus 1]}}{2^1} + \frac{A^{[j\boxminus 2]}}{2^2} + \ldots + \frac{A^{[j\boxminus 255]}}{2^{255}} + \frac{A^{[j]} + \mathrm{P}\big(\Phi^{[j]}\big) = 1\big)}{2^{256}} \tag{36}$$

which can be rearranged into

$$(2^{256} - 1)\mathrm{P}\big(\Phi^{[j]} = 1\big) = 2^{255} A^{[j\boxminus 1]} + 2^{254} A^{[j\boxminus 2]} + \ldots + 2^1 A^{[j\boxminus 255]} + 2^0 A^{[j]}. \tag{37}$$

This can equivalently be written as

$$\mathrm{P}\big(\Phi^{[j]} = 1\big) = \frac{A \ggg j}{2^{256} - 1}. \tag{38}$$

Inserting this expression into:

$$\begin{aligned}\mathrm{P}\big(\Phi^{[j]} \neq A^{[j]}\big) &= \begin{cases} \mathrm{P}\big(\Phi^{[j]} = 0\big) = 1 - \mathrm{P}\big(\Phi^{[j]} = 1\big) & \text{if } A^{[j]} = 1 \\ \mathrm{P}\big(\Phi^{[j]} = 1\big) & \text{if } A^{[j]} = 0 \end{cases} \\ &= \Big| A^{[j]} - \mathrm{P}\big(\Phi^{[j]} = 1\big) \Big| \end{aligned} \tag{39}$$

leads to the following equation describing the probability for a bit-flip at position j.

$$\mathrm{P}\big(\Phi^{[j]} \neq A^{[j]}\big) = \left| A^{[j]} - \frac{A \ggg j}{2^{256} - 1} \right|. \tag{40}$$

The probabilities will be unique for each bit position, as A is formed by repeating the 6-bit block 110100, which fits unevenly into a 256-bit integer. Consequently, $A \ggg i \neq A$ for all $i \bmod 256 \neq 0$, thereby making $\mathrm{P}\big(\Phi^{[j]} \neq A^{[j]}\big)$ unique for each j.