The Stream Cipher Rabbit

Martin Boesgaard Mette Vesterager Thomas Christensen
Erik Zenner

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

Contents

1 Introduction

Rabbit is a synchronous stream cipher that was first presented at the Fast Software Encryption workshop in 2003 [6]. Since then, an IV-setup function has been designed [17], and additional security analysis has been completed. No cryptographical weaknesses have been revealed until now.

The Rabbit algorithm can briefly be described as follows. It takes a 128-bit secret key and a 64-bit IV (if desired) as input and generates for each iteration an output block of 128 pseudo-random bits from a combination of the internal state bits. Encryption/decryption is done 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 functions. The counters ensure a lower bound on the period length for the state variables.

Rabbit was designed to be faster than commonly used ciphers and to justify a key size of 128 bits for encrypting up to 2^{64} blocks of plaintext. This means that for an attacker who does not know the key, it should not be possible to distinguish up to 2^{64} blocks of cipher output from the output of a truly random generator, using less steps than would be required for an exhaustive key search over 2^{128} keys.

2 Stream Cipher Specification

2.1 Notation

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".

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 $c_{j,i}$ denotes the corresponding counter variable. 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.

2.2 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:

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

and

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

The system is iterated four times, according to the next-state function defined in section 2.4, to diminish correlations between bits in the key and bits in the internal state variables. Finally, the counter variables are re-initialized according to:

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

for all j, to prevent recovery of the key by inversion of the counter system.

2.3 IV Setup Scheme

Let the internal state after the key setup scheme be denoted the master state, and let a copy of this master state be modified according to the IV scheme. The IV setup scheme works by modifying the counter state as function of the IV. This is done by XORing the 64-bit IV on all the 256 bits of the counter state. The 64 bits of the IV are denoted $IV^{[63..0]}$. The counters are modified as:

$$\begin{aligned} c_{0,4} &= c_{0,4} \oplus IV^{[31..0]} & c_{1,4} &= c_{1,4} \oplus (IV^{[63..48]} \diamond IV^{[31..16]}) \\ c_{2,4} &= c_{2,4} \oplus IV^{[63..32]} & c_{3,4} &= c_{3,4} \oplus (IV^{[47..32]} \diamond IV^{[15..0]}) \\ c_{4,4} &= c_{4,4} \oplus IV^{[31..0]} & c_{5,4} &= c_{5,4} \oplus (IV^{[63..48]} \diamond IV^{[31..16]}) \\ c_{6,4} &= c_{6,4} \oplus IV^{[63..32]} & c_{7,4} &= c_{7,4} \oplus (IV^{[47..32]} \diamond IV^{[15..0]}). \end{aligned} \quad (4)$$

The system is then iterated four times to make all state bits non-linearly dependent on all IV bits. The modification of the counter by the IV guarantees that all 2^{64} different IVs will lead to unique keystreams.

2.4 Next-state Function

The core of the Rabbit algorithm is the iteration of the system defined by the following equations:

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

$$g_{j,i} = \big((x_{j,i} + c_{j,i+1})^2 \oplus ((x_{j,i} + c_{j,i+1})^2 \gg 32)\big) \bmod 2^{32} \tag{6}$$

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

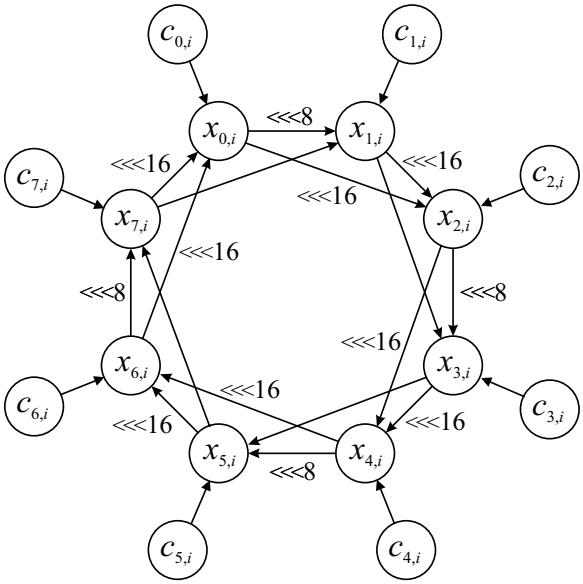

Figure 1: Graphical illustration of the system.

2.5 Counter System

The dynamics of the counters is defined as follows:

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

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

$$\phi_{j,i+1} = \begin{cases} 1 & \text{if } c_{0,i} + a_0 + \phi_{7,i} \geq 2^{32} \wedge j = 0 \\ 1 & \text{if } c_{j,i} + a_j + \phi_{j-1,i+1} \geq 2^{32} \wedge j > 0 \\ 0 & \text{otherwise.} \end{cases} \tag{8}$$

Furthermore, the a_j constants are defined as:

$$\begin{aligned}
a_0 &= \text{0x4D34D34D} & a_1 &= \text{0xD34D34D3} \\
a_2 &= \text{0x34D34D34} & a_3 &= \text{0x4D34D34D} \\
a_4 &= \text{0xD34D34D3} & a_5 &= \text{0x34D34D34} \\
a_6 &= \text{0x4D34D34D} & a_7 &= \text{0xD34D34D3.}
\end{aligned} \tag{9}$$

2.6 Extraction Scheme

After each iteration the output is extracted as follows:

$$\begin{aligned}
s_i^{[15..0]} &= x_{0,i}^{[15..0]} \oplus x_{5,i}^{[31..16]} & s_i^{[31..16]} &= x_{0,i}^{[31..16]} \oplus x_{3,i}^{[15..0]} \\
s_i^{[47..32]} &= x_{2,i}^{[15..0]} \oplus x_{7,i}^{[31..16]} & s_i^{[63..48]} &= x_{2,i}^{[31..16]} \oplus x_{5,i}^{[15..0]} \\
s_i^{[79..64]} &= x_{4,i}^{[15..0]} \oplus x_{1,i}^{[31..16]} & s_i^{[95..80]} &= x_{4,i}^{[31..16]} \oplus x_{7,i}^{[15..0]} \\
s_i^{[111..96]} &= x_{6,i}^{[15..0]} \oplus x_{3,i}^{[31..16]} & s_i^{[127..112]} &= x_{6,i}^{[31..16]} \oplus x_{1,i}^{[15..0]}
\end{aligned} \tag{10}$$

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

2.7 Encryption/decryption Scheme

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

$$c_i = p_i \oplus s_i, \tag{11}$$
$$p_i = c_i \oplus s_i, \tag{12}$$

where c_i and p_i denote the i^{th} 128-bit ciphertext and plaintext blocks, respectively.

3 Hidden Weaknesses

We, the designers of Rabbit, hereby state that no hidden weaknesses have been inserted by us in the Rabbit algorithm.

4 Security Properties

Extensive security evaluations have been conducted on the Rabbit design. A full description of the results is presented in [6, 17] and in a series of white papers, available from *www.cryptico.com*. The following security properties are claimed:

- The cipher provides 128-bit security, i.e. a successful attack has to be more efficient than 2^{128} Rabbit trial encryptions.
- If IV is used, security for up to 2^{64} different IVs is provided, i.e. by requesting 2^{64} different IVs, the attacker does not gain an advantage over using the same IV.
- For a successful attack, the attacker has up to 2^{64} matching pairs of plaintext and ciphertext blocks available.

The following gives a short survey of the results from security evaluations, for the full details, please refer to the white papers.

4.1 Key Setup Properties

Design Rationale: The key setup can be divided into three stages: Key expansion, system iteration, and counter modification.

- The *key expansion stage* guarantees a one-to-one correspondence between the key, the state and the counter, which prevents key redundancy. It also distributes the key bits in an optimal way to prepare for the the system iteration.

- The *system iteration* makes sure that after one iteration of the next-state function, each key bit has affected all eight state variables. It also 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.

- Even if the counters are presumed known to the 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. It also destroys the one-to-one correspondence between key and counter, however, this should not cause a problem in practice (see below).

Attacks on the Key Setup Function: After the key setup, 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 would make 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 will almost certainly lead to different keystreams[1]. Note that key expansion and system iteration were 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 after the four initial iterations, the inner state is essentially random and not correlated with the counter system, the probability for counter collisions is given by the birthday paradox, i.e. for all 2^{128} keys, one collision is expected in the 256-bit counter state. Thus, counter collisions should not cause a problem in practice.

[1]The reason is that when the periodic part of the functional graph has been reached, the next-state function, including the counter system, is one-to-one on the set of points in the period.

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 $a_j \neq a_{j+2}$.

4.2 IV Setup Properties

Design Rationale: The security goal of the IV scheme of Rabbit is to justify an IV length of 64 bits for encrypting up to 2^{64} plaintexts with the same 128-bit key, i.e. by requesting up to 2^{64} IV setups, no distinguishing from random should be possible. There are two stages: IV addition and system iteration.

- The *IV addition* modifies the counter values in such a way that it can be guaranteed that under an identical key, all 2^{64} possible different IVs will lead to unique keystreams. Note that each IV bit will affect the input of four different g-functions in the first iteration, which is the maximal possible influence for a 64-bit IV. The expansion of the bits also takes the specific rotation scheme of the g-functions into account, preparing for the system iteration.

- The *system iteration* guarantees that after just one iteration, each IV bit has affected all eight state variables. The system is iterated four times in total in order to make all state bits non-linearly dependent on all IV bits.

A full security analysis of the IV setup is given in [4]. It concludes that the good diffusion and non-linearity properties (see below) of the Rabbit next-state function seem to prevent all known attacks against the IV setup scheme.

4.3 Period Length

A central property of counter assisted stream ciphers [18] is that strict lower bounds on the period lengths can be provided. The counter system adopted in Rabbit has a period length of $2^{256} - 1$ [6]. Since it can be shown that the input to the g-functions has at least the same period, a very pessimistic

lower bound of 2^{215} can be guaranteed on the period of the state variables [17].

4.4 Partial Guessing

Guess-and-Verify Attack: Such attacks become possible if output bits can be predicted from a small set of inner state bits. The attacker will guess the relevant part of the state, predict the output bits and compare them with actually observed output bits, thus verifying whether his guess was correct.

In [6], it was shown that the attacker must guess at least $2 \cdot 12$ input bytes for the different g-functions in order to verify against one byte. This is equivalent to guessing 192 bits and is thus harder than exhaustive key search. It was also shown that even if the attacker verifies against less than one byte of output, the work required is still above 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 4.6). To conclude, it seems to be impossible to verify a guess of fewer than 128 bits against the output.

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. The system is then iterated a few times, producing output that can be compared with the actual cipher output, verifying the guess.

In the following, we sketch an attack based on guessing bytes, with the counters being considered as static for simplicity. The attacker tries to reconstruct 512 bit of inner state, i.e. he observes 4 consecutive 128-bit outputs of the cipher and proceeds as follows:

- Divide the 32-bit counter and state variables into 8-bit variables.
- Construct an equation system that models state transition and output. For each of the 4 outputs, he obtains $8 \cdot 2 = 16$ equations. For each of the 3 state transitions, he obtains $8 \cdot 4 = 32$ equations. Thus, he has an overall of 160 equations and 160 variables ($4 \cdot 32$ state and 32 counter variables).
- Solve this equation system by guessing as few variables as possible.

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

lower bounded by the 8-bit subsystem with the smallest number of input variables. Neglecting the counters, the results of [6] 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.

4.5 Algebraic Attacks

Known Algebraic Attacks: The algebraic attacks on stream ciphers discussed in the literature [1, 8, 9, 7, 10] target ciphers whose internal state is mainly updated in a linear way, with only a few memory bits having a non-linear update function. This, however, is not the case for Rabbit, where 256 inner state bits are updated in a strongly non-linear fashion. In the following, we will discuss in some detail the non-linearity properties of Rabbit, demonstrating why the known algebraic attacks are not applicable against the cipher.

The Algebraic Normal Form (ANF) of the g-function: A convenient way of representing Boolean functions is through its algebraic normal form (see, e.g., [16]). Given a Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$, the ANF is the representation of f as a multivariate polynomial (i.e., a sum of monomials in the input variables). Both a large number of monomials in the ANF and a good distribution of their degrees are important properties of non-linear building blocks in ciphers.

For a random Boolean function in 32 variables, the average total number of monomials is 2^{31}, and the average number of monomials including a given variable is 2^{30}. If we consider 32 such random functions, then the average number of monomials that are not present in any of the 32 functions is 1 and the corresponding variance is also 1. For more details, see [2].

For the g-function of Rabbit, the ANFs for the 32 Boolean subfunctions have an algebraic degree of at least 30. The number of monomials in the functions range from $2^{24.5}$ to $2^{30.9}$, where for a random function it should be 2^{31}. The distribution of monomials as function of degree is presented in Fig. 2. Ideally the bulk of the distribution should be within the dashed lines that illustrate the variance for ideal random functions. Some of the Boolean functions deviate significantly from the random case, however, they all have a large number of monomials of high degree.

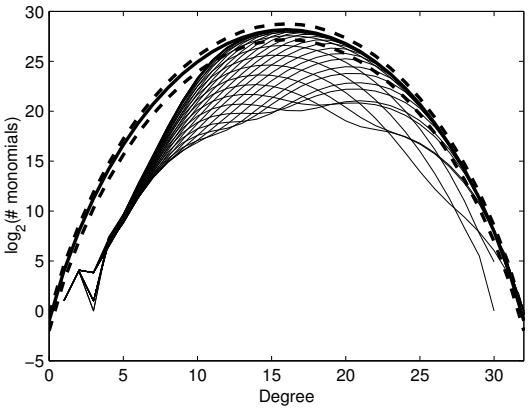

Figure 2: The number of monomials of each degree in each of the 32 Boolean functions of the g-function. The thick solid line and the two dashed lines denote the average and variance for an ideal random function.

Furthermore, the overlap between the 32 Boolean functions that constitute the g-function was investigated. The total number of monomials that only occur once in the g-function is $2^{26.03}$, whereas the number of monomials that do not occur at all is $2^{26.2}$. This should be compared to the random result which has a mean value of 1 and a variance of 1.

To conclude, the results for the g-function were easily distinguishable from random. However, the properties of the ANFs for the output bits of the g-function are highly complex, i.e. containig more than 2^{24} monomials per output bit, and with an algebraic degree of at least 30. Furthermore, no obvious exploitable structure seems present.

The Algebraic Normal Form (ANF) of the full cipher: It is clearly not feasible to calculate the full ANF of the output bits for the complete cipher. But reducing the word size from 32 bits to 8 bits makes it possible to study the 32 output Boolean functions as function of the 32-bit key.

For this scaled-down version of Rabbit, the setup function for different numbers of iterations was investigated. In the setup of Rabbit, four iterations of next-state are applied, plus one extra before extraction. We have determined the ANFs after 0+1, 1+1, 2+1, 3+1 and 4+1 iterations, where the +1 denotes the iteration in the extraction.

The results were much closer to random than in the case of the g-function. For 0+1 iterations, we found that the number of monomials is

very close to 2^{31} as expected for a random function. Already after two iterations the result seems to stabilize, i.e. the amount of fluctuations around 2^{31} does not change when increasing the number of iterations. We also made an investigation of the number of missing monomials for all 32 output bits. It turned out that for the 0+1, 1+1, 2+1, 3+1 and 4+1 iterations, the numbers were 0, 1, 2, 3 and 1, respectively. This seems in accordance with the mean value of 1 and variance of 1 for a random function. So after a few iterations, basically all possible monomials are present in the full cipher output functions.

Concluding, for the down-scaled version of the full cipher, no non-random properties were identified. For full details of the analysis, including statistical data, the reader may refer to [2].

Overdefined Equation Systems in the State: For simplicity, we ignore the counters and consider only the 256 inner state bits. Furthermore, we replace all arithmetical additions by XOR and omit the rotations. The use of XOR is a severe simplification as this will guarantee that the algebraic degree of the complete cipher will never exceed 32 for one iteration (but, of course, grow for more iterations).

With the inner state consisting of 256 bit, we need the output of at least two (ideally consecutive) iterations, giving us a non-linear system of 256 equations in 256 variables. Note that in the modified Rabbit design, everything is linear with the exception of the g-functions. Thus, we can calculate the number of monomials when expressing the output as a function of the state bits as follows:

- The output of the first iteration can be modelled as a linear function in the inner state. Thus, we obtain 128 very simple linear equations, containing all 256 monomials of degree 1.
- In order to generate the output of the next iteration, however, the inner state bits are run through the g-functions. Remember that $2^{32}-2^{26.2} \approx 2^{31.97}$ monomials (are contained in the output of each g-functions. Thus, the second set of equations contains approximately $8 \cdot 2^{31.97} = 2^{34.97}$ monomials.

In particular, this means that the non-linear system of equations is neither sparse, nor is it of low degree. Linearizing it increases the number of variables to about 2^{35}, and in order to solve it, an extra $2^{35}-2^8$ equations are required. These can not be obtained by using further iterations, because this way, the number of monomials increases beyond 2^{128}. Analysis conducted in

[2] indicates that they can not be obtained by using implicit equations, either. If, however, it would be possible to find such equations, the non-linear additions and the counter system would most likely destroy their benefit. Thus, we do not expect a algebraic attack using the inner state bits as variables to be feasible.

Overdefined Equation Systems in the Key: An algebraic attack targeting the key bits is even more difficult, since there are at least five rounds iterations of the non-linear layer before the first output bits can be observed (nine rounds if IV is used). Thus, the ANF of the full cipher has to be considered. Remembering that for the 8-bit version of the cipher, the ANF of the cipher is equivalent to a random function after just two iterations, it becomes obvious that the number of monomials in the equation system would be close to the maximum of 2^{128}. Solving such a system of equations would be well beyond a brute force search over the key space.

4.6 Correlation Attacks

Linear Approximations: In [6], at thorough investigation of linear approximations by use of the Walsh-Hadamard Transform [16, 11] was made. The best linear approximation between bits in the input to the next-state function and the extracted output found in this investigation had a correlation coefficient of $2^{-57.8}$.

In 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} blocks of output cannot be applied using only the best linear approximation 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 [13].

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. [14]) 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 exist many such large and useable correlations, we do not believe such an attack to be feasible[2].

[2]Knowing the values of the counters may significantly improve both the Guess-and-Determine attack, the Guess-and-Verify 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.

Second Order Approximations: However, it was found that truncating the ANFs of the g-functions after second order terms proposes relatively good approximations under the right circumstances.

We denote by $f^{[j]}$ the functions that contain the terms of first and second order of the ANF of $g^{[j]}$. Measurements of the correlation between $f^{[j]}$ and $g^{[j]}$ revealed correlation coefficients of less than $2^{-9.5}$, which is relatively poor compared to the corresponding linear approximations. However, the XOR sum of two neighbor bits, i.e. $g^{[j]} \oplus g^{[j+1]}$ was found to be correlated with $f^{[j]} \oplus f^{[j+1]}$ with correlation coefficients as large as $2^{-2.72}$. This could indicate that some terms of higher degree vanish when two neighbor bits are XOR'ed.

These results can be applied to construct second order approximations of the cipher. The best one is correlated to the real function with a correlation coefficient of $2^{-26.4}$, and a number of approximations with correlation coefficients of similar size. Preliminary investigations were made with other XOR sums. In general, sums of two bits can be approximated significantly better than single bits. The sum of neighboring bits does, however, seem to be the best approximation. Preliminary investigations show that approximations of sums of more than two bits have relatively small correlation coefficients.

It is not trivial to use second-order relations in linear cryptanalysis, and even the improved correlation values are not high enough for an attack as we know it. In an attack it would be necessary to include the counter, and set up relations between two consecutive outputs. We expect this to seriously complicate such an attack and make it infeasible.

4.7 Differential Analysis

Difference scheme: Given two inputs x' and x'', and their corresponding outputs y' and y'' (all in $\{0,1\}^n$), the following difference schemes were used:

- The *subtraction modulus* input and output differences are defined by $\Delta x = x' - x'' \mod 2^n$ and $\Delta y = y' - y'' \mod 2^n$, respectively.
- The *XOR* difference scheme is defined by $\Delta x = x' \oplus x''$ and $\Delta y = y' \oplus y''$.

Other differences are in principle possible, however, none of them were found to be better than the above ones.

Differentials of the g-function: Differentials of the g-function are investigated in [3]. While in principle, it would be necessary to calculate the

probabilities of all 2^{64} possible differentials (which is not feasible given standard equipment), valuable insights can be gained by considering smaller versions of the g-functions. This way, 8-, 10-, 12-, 14-, 16- and 18-bit g-functions were considered.

For the XOR difference operator, the investigation of reduced g-functions revealed a simple structure of the most likely differential that persisted for all sizes. The input differences were characterized by a block of ones of size of approximately $\frac{3}{4}$ of the word length[3]. Making the reasonable assumption that these properties will be maintained in the 32-bit g-function, all input differences constituted by single blocks of ones were considered. The largest probability, and most likely the largest of all, found in this investigation was $2^{-11.57}$ for the differential (0x007FFFFE, 0xFF001FFF).

For the subtraction modulus difference, no such clear structure was observed, so the differentials with the largest probabilities could not be determined for the 32-bit g-function. However, the probabilities scale nicely with word length. Assuming that this scaling continues to 32-bit, the differential with the largest probability is expected to be of the order 2^{-17}. The probabilities are significantly lower compared those available for the XOR difference operator.

Higher order differentials were also briefly investigated, but due to the huge complexity, only g-functions with very small word length could be examined. This revealed that in order to obtain a differential with probability 1, the differential has to be of order equal to the word length, meaning that the non-linear order of the g-function is maximal, for the small word length g-functions examined.

Differentials of the full cipher: The differentials of the full cipher were extensively investigated in [2]. It was shown that any characteristic will involve at least 8 g-functions[4].

From analyzing the transition matrices for smaller word length g-functions it was found that after about four iterations of those, there resulted a steady state distribution of matrix elements close to uniform for both the XOR and subtraction modulus difference schemes. Using this and that the probability for the best characteristic, $P_{\max}$, satisfies $P_{\max} < 2^{-11.57 \cdot 8} \ll 2^{-64}$, we do not expect any exploitable differential.

For a very simplified version of Rabbit, without rotations and with the XOR operation in the g-function replaced by an addition mod 2^{32}, higher

[3]Other structural properties are also present, they are described in [2] in more detail.

[4]probably it can be shown that 16 g-functions are the true minimum.

order differentials can be used to break the IV setup scheme even for a relatively large number of iterations. If we consider another simplified version, with rotations, third order differential still has a high probability for one round. However, for more iterations, the security increases very quickly. Finally, using the XOR in the g-function completely destroys the applicability of higher order differentials based on modular subtraction and XOR.

4.8 Statistical Tests

The statistical tests on Rabbit were performed using the NIST Test Suite [15], the DIEHARD battery of tests [12] and the ENT test [20]. 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 reduced to 8 bit. No weaknesses were found in any of these cases.

5 Strengths and Advantages

The design of Rabbit is a stream cipher with a new type of design. It provides a strong non-linear mixing of the inner state between two iterations. As opposed to almost all other designs currently available, it uses neither linear feedback shift registers nor S-boxes. These design decision have a number of important consequences.

5.1 Compact design

The design of Rabbit is very compact. All arithmetical operations used in the cipher are provided by modern processors, thus leading to very high speeds on a variety of platforms. As opposed to many stream cipher proposals, this speed advantage also holds for the key and IV setup.

If implemented in hardware, the gate count is low. Due to the lack of S-boxes and operations in GF(2^n), no lookup tables are required, keeping the memory requirements for both hardware and software base implementations very low (basically, only a copy of the inner state has to be stored). Also note that on most modern processors, the full inner state fits into the registers, eliminating (computationally expensive) memory access.

5.2 Security

The design of Rabbit makes the most wide-spread attacks against stream ciphers inapplicable. Both algebraic attacks ([10] and subsequent work) and correlation attacks ([19] and subsequent work) against stream ciphers are targeting designs with internal linear structures, which are not prevalent in Rabbit. Also the Time-Memory-Data tradeoffs ([5] and subsequent work) are not applicable due to the large internal state of 513 bits.

Nonetheless, Rabbit has been evaluated against all known attack techniques both from stream and block cipher cryptanalysis, and it has been carefully optimized in order to avoid any weaknesses towards them. We are thus optimistic that any attack against Rabbit would have to be based on a completely new attack technique.

6 Design Rationale

Rabbit was specifically designed to be very efficient in software implementations without sacrificing any security. It is meant to provide 128-bit security for up to 2^{64} blocks of plaintext. In the following, the decisions involved in designing the cipher are briefly sketched.

6.1 Key and IV setup

The design rationales for the key and IV setup functions were described in sections 4.1 and 4.2.

6.2 The g-function

Following initial ideas from chaos theory, the g-function was to be built on the basis of the arithmetical squaring function $\mathrm{sqr} : \{0,1\}^{32} \rightarrow \{0,1\}^{64}$. The question was how to reduce the 64-bit output of sqr to 32 bit. Not surprisingly, analysis showed that the correlation between input and output bits was particularly high for high and low order bits, while better results were obtained for the middle bits. The corresponding graph is roughly v-shaped.

One idea would have been to use only the 32 middle bits. However, combining the upper half of the output with the lower half using xor as done in Rabbit provides much better results; correlation coefficients are much lower for all output bits.

6.3 The counter system

Note that the inner state bits of Rabbit are updated in a highly non-linear fashion. Such systems are known to have unpredictable period lengths. Thus, it was decided to overcome this problem using counters, inspired by [18].

As opposed to the proposal in [18], the counters were added to the inner state bits before running them through the g-function. This was done in order to "hide" the counter values, i.e. to make sure that they can not easily be reconstructed from the output. On the other hand, this made the proof of the period length harder [17].

Note that a standard counter construction (using arithmetical addition of a constant to increase the counter value) would lead to very predictable behaviour for some bits. As an example, for all odd constants, the least significant bit would flip with every iteration. Thus, the carry feedback construction was introduced, making sure that all bit positions are equally strong.

A weak choice of the counter constant A would also result in a security problem. If the number contains too many consecutive zeroes or ones, the flip behaviour of certain bits could be predicted. Thus, the constant A was chosen to be the binary sequence 110100 repeated. This value makes sure that all counter bits have unique flip probabilities. Also note that all rotated versions of that constant are pairwise prime with $2^{256} - 1$.

6.4 Symmetry and Mixing

The system was designed to be as symmetric as possible in order to facilitate analysis. However, in order to prevent the attacker to decompose the system into subsystems, next-state function, rotations, counter modification in setup, and extraction were constructed in such a way that a thorough mixing of states is achieved.

In particular, all possible rotations were tested in order to identify the ones that would provide a maximal mixing of internal state bits and thus avoid system decomposition. Amongst the most secure rotations, the most efficient ones were chose for Rabbit.

7 Computational Efficiency

7.1 Software Performance

Encryption speeds for the specific processors were obtained by encrypting 8 kilobytes of data stored in RAM and measuring the number of clock cycles passed. 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 related to the calling convention (function arguments, return address and saved registers) and for temporary data. Memory for storing the key, instance, ciphertext and plaintext has not been included. All performance results, code size and memory requirements are listed in Table 1 below.

Intel Pentium Architecture: The performance was measured on a 1.0 GHz Pentium III processor and on a 1.7 GHz Pentium 4 processor. The speed-optimized version of Rabbit was programmed in assembly language (using MMX instructions) inlined in C and compiled using the Intel C++ 7.1 compiler. A memory-optimized version can eliminate the need for memory, since the entire instance structure and temporary data can fit into the CPU registers.

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.

MIPS 4Kc Architecture: An assembly language version of Rabbit has been written for the MIPS 4Kc processor[5]. Development was done using The Embedded Linux Development Kit (ELDK), which includes GNU cross-development tools. Performance was measured on a 150 MHz processor running a Linux operating system.

8-bit Processors: The simplicity and small size of Rabbit makes it suitable for implementations on processors with limited resources such as 8-bit microcontrollers. Multiplying 32-bit integers is rather resource demanding using plain 32-bit arithmetics. However, squaring involves only ten 8-bit multiplications which reduces the workload by approximately a factor of

[5]The MIPS 4Kc processor has a reduced instruction set compared to other MIPS 4K series processors, which decreases performance.

Processor	Performance	Code size	Memory
Pentium III	3.7/278/253	440/617/720	40/36/44
Pentium 4	5.1/486/648	698/516/762	16/36/28
ARM7	9.6/610/624	368/436/408	48/80/80
MIPS 4Kc	10.9/749/749	892/856/816	40/32/32

Table 1: Performance (in clock cycles or clock cycles per byte), code size and memory requirements (in bytes) for encryption / key setup / IV setup.

two. Finally, the rotations in the algorithm have been chosen to correspond to simple byte-swapping.

7.2 Hardware Estimates

ASIC Performance: The toughest operation from a hardware point of view is the 32-bit squaring. If no separate squaring unit is available, the nature of squaring allows for some simplification over an ordinary 32×32 multiplication. It can be implemented as three 16×16 multiplications followed by addition. Being the most complex part of the algorithm, it determines the overall speed and contributes significantly to the gate count.

The 8 internal state and counter words can be computed using between 1 and 8 parallel pipelines. Estimates for different versions are given in table 2, giving gate count, die area and performance on a .18 micron technology. If greater speed is needed and if the gate count is of less importance, more advanced multiplication methods can be used. The gate count and die area numbers include key setup and IV setup.

Pipelines	Gate count	Die area	Performance
1	28K	0.32 mm^2	3.7 GBit/s
2	35K	0.40 mm^2	6.2 GBit/s
4	57K	0.66 mm^2	9.3 GBit/s
8	100K	1.16 mm^2	12.4 GBit/s

Table 2: Hardware estimates for Rabbit on .18 micron technology.

The performance numbers for 4 parallel pipelines can be doubled if two instances of Rabbit are executed in an interleaving manner (will require approx. 10K gates extra). Performance for 8 pipelines can be tripled in the same way if three instances are executed (approx. 20K gates extra).

FPGA Performance: When implementing Rabbit in an FPGA, the challenges will be similar to those in an ASIC implementation. Again the squaring operation will be the most complex element. Several FPGA families have dedicated multiplication units available. An example of this would be the Xilinx Spartan 3 or Altera Cyclone II families. In these architectures the latencies of the multiplier units are given to be 2.4 and 4.0 ns respectively. Based on a 2-pipeline design similar to that discussed in the ASIC section this will give us decryption performance of 8.9 Gbit/s and 5.3 Gbit/s respectively if the multiplication is the bottleneck. These implementations will fit on any of the Altera Cyclone II family members and from Xilinx XC3S200 and upwards.

Depending on the number of multipliers available in the chosen FPGA, greater parallelism can be exploited for better performance by increasing the number of pipelines. With 24 multipliers available, throughputs of 17.8 Gbit/s and 10.7 Gbit/s will be achievable. This number of multipliers is present, e.g., in chips from Altera EP2C20 and Xilinx XC3S1000 upwards.

8 Advice for Implementers

Note that all of Rabbits elementary operations are readily available on a standard processor. Thus, the reference implementation is very similar to the specification and shows how to obtain an elegant basic implementation of Rabbit. Since no S-boxes or computations over GF(2^n) are required, table lookups for increased performance are not required to implement the cipher. In order to obtain fast implementations, it is recommended to implement at least the g-function in assembly language and inline it into the code. Assembly code can also help improve the performance by making use of the carry flag (in order to implement the counter system).

References

[1] F. Armknecht and M. Krause. Algebraic attacks on combiners with memory. In D. Boneh, editor, *Proc. Crypto 2003*, volume 2729 of *LNCS*, pages 162–175. Springer, 2003.

[2] Cryptico A/S. Algebraic analysis of Rabbit. http://www.cryptico.com, 2003. white paper.

[3] Cryptico A/S. Differential properties of the g-function. http://www.cryptico.com, 2003. white paper.

[4] Cryptico A/S. Security analysis of the IV-setup for Rabbit. http://www.cryptico.com, 2003. white paper.

[5] S. Babbage. A space/time tradeoff in exhaustive search attacks on stream ciphers. In *European Convention on Security and Detection*, volume 408 of *IEE Conference Publication*, May 1995.

[6] M. Boesgaard, M. Vesterager, T. Pedersen, J. Christiansen, and O. Scavenius. Rabbit: A new high-performance stream cipher. In T. Johansson, editor, *Proc. Fast Software Encryption 2003*, volume 2887 of *LNCS*, pages 307–329. Springer, 2003.

[7] N. Courtois. Fast algebraic attacks on stream ciphers with linear feedback. In D. Boneh, editor, *Proc. Crypto 2003*, volume 2729 of *LNCS*, pages 176–194. Springer, 2003.

[8] N. Courtois. Higher order correlation attacks, XL algorithm and cryptoanalysis of toyocrypt. In P.J. Lee and C.H. Lim, editors, *Proc. Information Security and Cryptology 2002*, volume 2587 of *LNCS*, pages 182–199. Springer, 2003.

[9] N. Courtois and W. Meier. Algebraic attacks on stream ciphers with linear feedback. In E. Biham, editor, *Proc. of Eurocrypt 2003*, volume 2656 of *LNCS*, pages 345–359. Springer, 2003.

[10] N. Courtois and J. Pieprzyk. Cryptanalysis of block ciphers with overdefined systems of equations. In Y. Zheng, editor, *Proc. Asiacrypt 2002*, volume 2501 of *LNCS*, pages 267–287. Springer, 2003.

[11] J. Daemen. *Cipher and hash function design strategies based on linear and differential cryptanalysis.* PhD thesis, KU Leuven, March 1995.

[12] G. Masaglia. A battery of tests for random number generators. http://stat.fsu.edu/~geo/diehard.html, 1996.

[13] M. Matsui. Linear cryptanalysis method for DES cipher. In T. Helleseth, editor, *Proc. Eurocrypt '93*, volume 765 of *LNCS*, pages 386–397. Springer, 1993.

[14] W. Meier and O. Staffelbach. Fast correlation attacks on stream ciphers. In C. Günther, editor, *Proc. Eurocrypt '88*, volume 330 of *LNCS*, pages 301–314. Springer, 1988.

[15] National Institute of Standards and Technology. A statistical test suite for the validation of random number generators and pseudo random number generators for cryptographic applications. NIST Special Publication 800-22, http://csrc.nist.gov/rng, 2001.

[16] R. Rueppel. *Analysis and Design of Stream Ciphers*. Springer, 1986.

[17] O. Scavenius, M. Boesgaard, T. Pedersen, J. Christiansen, and V. Rijmen. Periodic properties of counter assisted stream cipher. In T. Okamoto, editor, *Proc. CT-RSA 2004*, volume 2964 of *LNCS*, pages 39–53. Springer, 2004.

[18] A. Shamir and B. Tsaban. Guaranteeing the diversity of number generators. *Information and Computation*, 171(2):350–363, 2001.

[19] T. Siegenthaler. Decrypting a class of stream ciphers using ciphertext only. *IEEE Transactions on Information Theory*, C-34(1):81–85, January 1985.

[20] J. Walker. A pseudorandom number sequence test program. http://www.fourmilab.ch/random, 1998.