# The Rabbit Stream Cipher - Design and Security Analysis

Martin Boesgaard, Thomas Pedersen, Mette Vesterager, and Erik Zenner

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

**Abstract.** The stream cipher Rabbit was first presented at FSE 2003 [6]. In the paper at hand, a full security analysis of Rabbit is given, focusing on algebraic attacks, approximations and differential analysis. We determine the algebraic normal form of the main nonlinear parts of the cipher as part of a comprehensive algebraic analysis. In addition, both linear and nonlinear approximations of the next-state function are presented, as well as a differential analysis of the IV-setup function. None of the investigations have revealed any exploitable weaknesses. Rabbit is characterized by high performance in software with a measured encryption/decryption speed of 3.7 clock cycles per byte on a Pentium III processor.

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

## 1 Introduction

Rabbit was first presented at the Fast Software Encryption workshop in 2003 [6]. Since then, an IV-setup function has been designed, and additional security analysis has been completed. The main results are presented in this paper.

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 $2^{64}$ bytes of plaintext. This means that for an attacker who does not know the key, it should not be possible to distinguish up to $2^{64}$ bytes 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.

### 1.1 Organization and Notation

In section two, we describe the design of Rabbit in detail. We discuss the cryptanalysis 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.

We use the following notation: $\oplus$ denotes logical XOR, $\ll$ 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 Rabbit Stream Cipher

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.

### 2.1 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 in section 2.3, to diminish correlations between bits in the key and bits in the internal state variables. Finally, the counter variables are modified according to:

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

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

### 2.2 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:

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

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.

### 2.3 Next-state Function

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

$$x_{j,i+1} = \begin{cases} g_{j,i} + (g_{j-1 \bmod 8,i} \lll 16) + (g_{j-2 \bmod 8,i} \lll 16) & \text{for } j \text{ even} \\ g_{j,i} + (g_{j-1 \bmod 8,i} \lll 8) + g_{j-2 \bmod 8,i} & \text{for } j \text{ odd} \end{cases} \tag{5}$$

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

where all additions are modulo $2^{32}$. Before an iteration the counters are incremented as described below.

### 2.4 Counter System

The dynamics of the counters is defined as follows:

$$c_{0,i+1} = \begin{cases} c_{0,i} + a_0 + \phi_{7,i} \bmod 2^{32} & \text{for } j = 0 \\ c_{j,i} + a_j + \phi_{j-1,i+1} \bmod 2^{32} & \text{for } j > 0, \end{cases} \tag{7}$$

where the carry $\phi_{j,i+1}$ 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:

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

### 2.5 Extraction Scheme

After each iteration, four 32-bit words of pseudo-random data are generated as follows:

$$\begin{aligned} s_{j,i}^{[15..0]} &= x_{2j,i}^{[15..0]} \oplus x_{2j+5 \bmod 8,i}^{[31..16]}, \\ s_{j,i}^{[31..16]} &= x_{2j,i}^{[31..16]} \oplus x_{2j+3 \bmod 8,i}^{[15..0]}. \end{aligned} \tag{10}$$

where $s_{j,i}$ is word $j$ at iteration $i$. The four pseudorandom words are then XOR'ed with the plaintext/ciphertext to encrypt/decrypt.

## 3 Security Analysis

In this section we first discuss the key setup function, IV setup function, and periodic properties. We then present an algebraic analysis of the cipher, approximations of the next-state function, differential analysis, and the statistical properties.

### 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.

In the key expansion stage, we ensure two properties. The first one is 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 of $g_{j,0}$, $g_{(j+1 \bmod 8),0}$, $g_{(j+4 \bmod 8),0}$ and $g_{(j+5 \bmod 8),0}$. In each of the eight next-state subfunctions, at least one of those $g$-functions enter.

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.

Even if the counters are be 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. 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.

**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 will almost certainly lead to different keystreams. 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. 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 $a_j \neq a_{j+2}$.

### 3.2 IV setup Properties

This IV expansion is chosen in order to take the specific rotation scheme of the $g$-functions into account. Note that each IV bit will affect four different $g$-functions in the first iteration, which is the maximal possible influence when the IV is 64 bit. Also, this scheme insures that all eight state variables are potentially affected after one iteration.

The system is then iterated four times[1] in order to make all state bits non-linearly dependent on all IV bits. The counter modification by the IV is chosen since this implies that all $2^{64}$ possible different IVs will lead to unique keystreams.

**General Security Goals and Arguments**

The security goal of the IV Scheme of Rabbit is that it should justify an IV length of 64 bits for encrypting up to $2^{64}$ plaintexts with the same 128-bit key, e.g. by requesting up to $2^{64}$ IV setups, no distinguishing from random should be possible.

There are several qualitative reason why we expect the IV-setup scheme of Rabbit to fulfill the above security goals. It was shown in [6] that the next-state function has good diffusion properties. For instance, after just two iterations every state bit depends on each key bit with a measured probability of one half. Furthermore, it was shown that each output byte (or bit) depends virtually on all input bytes (or bits). It was also demonstrated that the next-state function of Rabbit is highly non-linear. Due to the good diffusion and non-linearity properties and since the 64 IV bits are mixed into all the 256 state bits after the four IV setup iterations, we expect that differential attacks are impossible within the security goals.

In section 3.7, we provide some quantitative security arguments. In particular, we analyze the differential properties of this scheme. This is done in a way very similar to the usual differential cryptanalysis of block ciphers (see e.g. [11], chapter 5) but with emphasis on distinguishability properties and the uniqueness of keystreams for different IVs.

### 3.3 Period Length

The most important feature of counter assisted stream ciphers [21] is that strict lower bounds on the period lengths can be provided. The adopted counter system in Rabbit has a period length of $2^{256} - 1$ [6]. Since it can be shown that the input to the $g$-functions has at least the same period, a very pessimistic lower bound on the period of the state variables, $N_{\mathrm{x}} > 2^{215}$, can be guaranteed [19].

### 3.4 Algebraic Analysis

**Guess-and-Verify 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.

In [6], we found that 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 3.6. To conclude, it seems to be

[1] In principle, this amounts to five times before the extraction of the pseudorandom data, since the next-state function is applied once more after the IV setup has been completed as a part of the usual encryption scheme.

impossible 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 3.4 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.

**The Algebraic Normal Form (ANF) of Boolean Functions**

In this section, we describe the algebraic normal form of Boolean functions. Moreover, we discuss some properties of the algebraic normal form of random Boolean functions.

A convenient way of representing Boolean functions is through its algebraic normal form. Let a Boolean function $f \equiv f(x_0, x_1, ..., x_{n-1})$ from $\{0,1\}^n$ to $\{0,1\}$ be given, then its algebraic normal form is given by

$$f(x_0, x_1, ..., x_{n-1}) = \sum_{u \in \{0,1\}^n} a_u \prod_{i=0}^{n-1} x_i^{u_i}, \tag{11}$$

where $u \equiv (u_0, u_1, ..., u_i, ..., u_{n-1}) \in \{0,1\}^n$, and $a_u \in \{0,1\}$ is given by the Möbius transform of $f$:

$$a_u = \sum_{\{x : x \wedge \overline{u} = 0\}} f(x) \tag{12}$$

where $x \wedge \overline{u}$ means the component-wise logical AND operation between $x$ and the complement of $u$.

We say that the monomial $\prod_{i=0}^{n} x_i^{u_i}$ of degree $H(u)$ selected by $u$ is in the ANF for $f$ if $a_u = 1$. Here $H(u)$ denotes the Hamming weight of $u$. We can say that the ANF representation for $f$ is simply $f$ written as a multivariate polynomial over $GF(2)$.

The algebraic normal form of a Boolean function can be relatively easily constructed based on their respective truth table, by a method analogous to the Walsh-Hadamard Transform.

For a random Boolean function of $n$ variables, the number of monomials of degree $k$ is normally distributed with mean value and variance given by:

$$E[n_k] = \frac{1}{2}\binom{n}{k} \quad \text{and} \quad V[n_k] = \frac{1}{4}\binom{n}{k}. \tag{13}$$

The mean number $n_k^{x^i}$ of monomials of degree $k$ containing a given variable, $x^i$, and the corresponding variance is given by:

$$E[n_k^{x^i}] = \frac{1}{2}\binom{n-1}{k-1} \quad \text{and} \quad V[n_k^{x^i}] = \frac{1}{4}\binom{n-1}{k-1}. \tag{14}$$

Furthermore, we notice that the probability for a given monomial to be present in a random Boolean function is 1/2. Thus, for a random function mapping from $\{0,1\}^{32}$ to $\{0,1\}$, the average total number of monomials is $2^{31}$, obtained by summing eq. (13). The average number of monomials including a given variable is $2^{30}$, obtained by summing eq. (14). If we consider 32 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.

In the following we investigate the non-linear functions in Rabbit in detail. We do this by constructing the algebraic normal form of the output bits of the $g$-function as well as for a scaled-down version of the full cipher. We then compare them with results for random Boolean functions (In [13] the author also investigates random properties for various ciphers, however, here it is done in a different context.). In particular, we investigate properties that we believe to be relevant for an algebraic attack.

We have determined the ANF for the 32 Boolean functions of the $g$-function. All 32 Boolean functions 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 $2^{31}$ with a standard deviation of $2^{15}$.

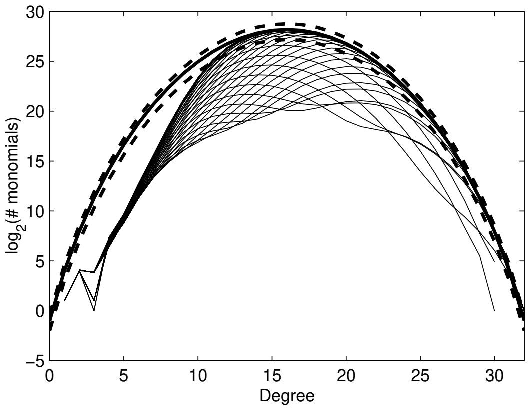


**Fig. 1.** The number of monomials of each degree in each of the 32 Boolean functions of the $g$-function. The thick solid line and the two dashed lines denote the average and variance for an ideal random function.

The distribution of monomials as function of degree is presented in Fig. 1. 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.

Furthermore, we investigated the overlap between the 32 Boolean functions that constitute the $g$-function. 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 one and a variance of one.

To conclude, we can say all the results for the $g$-function were easily distinguishable from random, but are still very acceptable, i.e. the monomials are of high degree and their number is large. Furthermore, no obvious exploitable structure seems present. In the above analysis, we did not investigate properties of implicit equations (i.e. equations containing monomials consisting of both input and output variables). This will be briefly discussed later.

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, we investigated the setup function for different numbers of iterations. 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.

In conclusion, we can see that the analysis showed that the properties of the ANFs for the output bits of the $g$-function were highly complex, i.e. larger than $2^{24}$ terms per output bit, and with an algebraic degree of at least 30. For the down-scaled version of the full cipher, the results were even better and no non-random properties were identified. For full details of the analysis, including statistical data, the reader may refer to [3].

### 3.5 Algebraic Attacks against Rabbit

In this section we will discuss whether it is possible to set up solvable equation systems both in the state variables as well as in the key. As illustrated both in section 2 and in the ANF analysis of the 8-bit version, each output depends on all its possible input, so a combination of guessing and analytical elimination of variables seems infeasible. Consequently, below we shall only consider algebraic attacks without combining with the guessing of some variables.

**Known Algebraic Attacks**

The known attacks on stream ciphers are mostly on ciphers based on linear feedback with more or less memoryless non-linear combiners/filters. Rabbit is based on iteration of non-linear functions so "everything is memorized" (disregarding the counter bits). One could argue that the counter system is (almost) linear (over GF($2^{32}$)), but it only constitutes half of the 513 bits.

In what sense can we generalize or use the concepts and ideas from the attacks described in [2, 8, 9, 7, 10]? First, we must understand how their attacks basically work. If the non-linear combiner is essentially memoryless and if it has low degree, we can capture enough output bits in order to get an overdefined equation system in the key bits. We can then linearize it, i.e. replace every monomial by a new variable and solve it using methods for linear equation systems. This can be done because we know how the binary equations look like as a function of the key bits, i.e. their degree is preserved over all iterations as the combiner is memoryless. This strategy can be refined. For instance, even though the combiner function should have large degree, it might be possible, by multiplying with another function, to obtain new equations with lower degree [10]. This is often possible, since the combiner function is simple, e.g. containing few monomials. Furthermore, if not too many, memory bits can also be handled (e.g. by elimination [2]).

How would this procedure work for Rabbit? Intuitively, overdefined equation systems can be found but will probably be very large when linearized as the $g$-function is 32-bit, i.e. the number of monomials is very large. Furthermore, the non-linear equations change for each iteration as function of the key, i.e. all the state bits are memory bits. Below, we will argue in some detail why we believe the above intuition to be true.

**Overdefined Equation Systems in the State**

This discussion will primarily be inspired by the algebraic analysis of Rijndael and Serpent performed by Courtois and Pieprzyk [10].

For simplicity we ignore the counters. Furthermore, we will 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).

According to the above ANF analysis of the 8-bit version, it seems that the Boolean functions for the complete cipher behave very close to random. The Boolean functions for the $g$-function seem to behave less randomly. We will in this subsection investigate whether this can be exploited.

Since everything is linear except the $g$-functions, we can easily calculate the number of monomials when expressing the output as a function of the state bits. 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 the output of the first iteration can be modelled as a linear function in the inner state, according to equation (10). 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. Note that each output bit for the second iteration depends on six output bits of the $g$-functions. Also remember from the last section that for each $g$-function, there are 32 output bits, and that the ANFs for these output bits contain almost all possible monomials of degree $\leq 32$. Thus, we have $2^{32} - 2^{26.2} \approx 2^{31.97}$ monomials that are contributed by one $g$-function, and approximately $8 \cdot 2^{31.97} = 2^{34.97}$ monomials overall.

In particular, this means that the non-linear system of equations is neither sparse, nor is it of low degree. Thus, it seems hopeless to try and solve it directly. Instead, we could try to linearize the monomials, which increases the number of variables to about $2^{35}$. Assuming that we had enough equations for all those variables, the solving complexity would be about $(2^{35})^3 = 2^{105}$ (using Strassen's algorithm). So in order to be able to solve the system with this complexity we need to find some $2^{35} - 2^8$ extra equations describing the next-state function or use an XL algorithm [20] when fewer equations are available.

We do not believe that it is possible to find that many extra equations. The reason is the following. As illustrated in [10], there exist implicit representations of the S-boxes in Rijndael and Serpent [12, 1] for different reasons. In Serpent they exist because of the small S-box size and in Rijndael because of a special property of its S-boxes. We have tried for low resolution $g$-functions to see whether there were any obvious implicit relations, but we did not find anything useable. For small size $g$-functions (e.g. eight bit) it might be possible to perform a systematic search for implicit equations which hold with probability one[2]. However, we do not believe that there is any easy way to determine whether there exist any useful implicit equations of the 32-bit $g$-function. Moreover, even in the unlikely event that such equations could be found, the non-linear arithmetic additions and the rotations will most likely destroy the usefulness of such equations.

Furthermore, it does not really help to use output for more iterations. The reason is that even though we gain 128 extra equations for each new extraction, the number of monomials grows much more[3].

**Overdefined Equation Systems in the Key**

In this section we will investigate whether it is possible to construct overdefined equation systems in the key bits. The above approach (much like the analysis performed on block ciphers [10]) where it was tried to construct overdefined equation systems in the state variables, seemed non-promising. Basically, the reason for this is the high complexity of the involved Boolean functions; no simple description of those seems possible. In the case of overdefined equations systems in the key, non-linear memory effects will be difficult to handle. In the case of the simplified Rabbit as treated here, all state bits are non-linearly updated and therefore new monomials are generated for each iteration (See also footnote 3).

So the interesting question is whether we can use the specific equations in the 128-bit key. Since five iterations are performed before any bit is extracted, we now have to investigate Boolean functions of the complete cipher and not just those of the $g$-function (so now the counter system, the arithmetic additions as well as the rotations, are included again). It is currently impossible to directly construct the corresponding ANFs. However, the investigations done for the 8-bit version of the cipher clearly show that the relative complexity of the corresponding Boolean functions is greater than for the $g$-function, i.e. they look much more random. So we expect that the number of monomials after a few iterations[4] will be on the

[2] Suggested to us by Vincent Rijmen. A preliminary investigation showed that there are no second order implicit equations that hold with probability one.

[3] Note that, even if we had a linear updating of the state and the $g$-functions just served as non-linear filters (i.e. no memory) then the total number of possible monomials would be:

$$N_m^{\max} = \sum_{i=0}^{32} \binom{256}{i} \approx 2^{135}, \tag{15}$$

which rules out such an attack even on a memoryless type of cipher using the $g$-functions as non-linear filters.

[4] We believe that after about four iterations, the number of monomials will be very large since after two iterations an input bit will have influenced all output bits.

order of:

$$N_m^{\max} = \sum_{i=0}^{128} \binom{128}{i} = 2^{128}, \tag{16}$$

Therefore, we do not expect that it is possible to construct a solvable overdefined equation system.

To sum up, the analysis performed above clearly indicates that algebraic attacks on Rabbit are infeasible. The reasons for this are the large non-linearly updated state and the complexity of the $g$-function, i.e. the number of monomials, their distribution and so on. Furthermore, the other non-linear ingredient of Rabbit, namely the arithmetic additions, strongly increases the complexity and makes the number of monomials, their distribution and so on of the complete cipher very complex and random-like.

### 3.6 Approximations of Rabbit

**Linear Approximations**

In [6], we made a thorough investigation of linear approximations by use of the Walsh-Hadamard Transform (WHT) [18, 11]. 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 $2^{-57.8}$, see [6] for more details.

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 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 [15].

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. [16]) 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. 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**

We discovered 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 $g(y)^{[j]}$. Then, it can be shown that these approximations can be written as

$$f^{[j]} = y^{[j/2]} \oplus y^{[j/2+16]} \oplus \bigoplus_{i=0}^{15} y^{[j/2+i]} y^{[j/2-1-i]} \tag{17}$$

for even bit-positions, where $+$ and $-$ are modulus 32, and $\bigoplus$ denotes the XOR sum. For odd bit-positions and $j \neq 1$

$$f^{[j]} = y^{[(j-1)/2]} y^{[(j-1)/2-1]} \oplus y^{[(j-1)/2+16]} y^{[(j-1)/2+15]} \oplus \bigoplus_{i=0}^{14} y^{[(j+1)/2+i]} y^{[(j+1)/2-2-i]} \tag{18}$$

For $j = 1$,

$$f^{[1]} = y^{[16]}y^{[15]} \oplus \bigoplus_{i=0}^{14} y^{[1+i]}y^{[31-i]}. \tag{19}$$

By measurements we can determine the probability,

$$P(f^{[j]} = g^{[j]}) = \frac{1}{2} + \epsilon, \tag{20}$$

and define the correlation coefficient to be $|2\epsilon|$.

Measurements of the correlation between the approximation $f^{[j]}$ and the actual function $g^{[j]}$ gave rather poor correlation coefficients 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 a much higher correlation coefficient. Approximations of single bits have correlation coefficients less than $2^{-9.5}$, whereas approximations of the XOR sums have 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. By using linear approximations of the additions of the $g$-function, the best second order approximation is:

$$s_0^{[26]} \oplus s_0^{[25]} \approx f_0^{[26]} \oplus f_0^{[25]} \oplus f_1^{[10]} \oplus f_1^{[9]} \oplus f_2^{[2]} \oplus f_2^{[1]} \oplus f_3^{[10]} \oplus f_3^{[9]} \oplus f_6^{[10]} \oplus f_6^{[9]} \oplus f_7^{[10]} \oplus f_7^{[9]}, \tag{21}$$

where the functions $f^{[j]}$ are given by eqs. 17, 18 and 19. The approximation is correlated to the real function with a correlation coefficient of $2^{-26.4}$.

A number of approximations can be constructed in this way with correlation coefficients of similar size. We made preliminary investigations with other XOR sums. In general, sums of two bits can be approximated significantly better than single bits. The sum of neighbor bits does, however, seem to be the best approximations. Preliminary investigations show that approximations of sums of more than two bits have relatively small correlation coefficients.

For comparison, the best linear approximation we have found has a correlation coefficient of $2^{-57.8}$. This correlation was also between the input to the next-state and the output.

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.

### 3.7 Differential Analysis

In this analysis we have used two difference schemes which are defined as follows: Take two inputs, $x'$ and $x''$, and their corresponding outputs $y'$ and $y''$, then 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 word length of the input and output variables is denoted by $n$. The XOR difference scheme is defined by $\Delta x = x' \oplus x''$ and $\Delta y = y' \oplus y''$. Since the subtraction modulus scheme causes the counters to be linear, we shall in many cases refer to that. XOR difference is also discussed in certain cases. We have not found any other difference schemes to be better.

**Differentials of the $g$-function**

Differentials of the $g$-function are investigated in [4], but we give a short description of the findings here.

In principle, it would be necessary to calculate the probabilities of all $2^{64}$ differentials. However, in terms of XOR as difference operator, the investigation of smaller word length $g$-functions has revealed that the structure of the differentials with the largest probabilities remain equivalent for the different word lengths. We have determined the probabilities for all differentials in 8-, 10-, 12-, 14-, 16- and 18-bit $g$-functions. The structure is characterized by a block of ones of size of approximately $\frac{3}{4}$ of the word length. Furthermore, the block starts at bit position one. For larger word lengths, i.e. 14-, 16- and 18-bit, all entries in the top 32 list have input differences build by one block of consecutive ones, but of various size and starting at different bit positions. Finding an analytical explanation of these properties remains an open research problem.

We make the reasonable assumption that these properties will be maintained in the 32-bit $g$-function, and investigate all input differences constituted by single blocks of ones. The largest probability, and most likely the largest of all, found in this investigation was $2^{-11.57}$ for the differential (0x007FFFFE, 0xFF001FFF).

The $g$-function is non-injective, which means that there are input differences besides 0 that map to an output difference of 0. The input difference resulting in an output difference of 0 with the largest probability is 0xFFFFFFFF, i.e. it corresponds to flipping all bits in the input to the function. For the smaller word length $g$-functions it is also this type of input difference, i.e. flipping all bits, that results in an output difference of zero with the largest probability. The probability for this in the 32-bit $g$-function is $2^{-14.41}$.

For the subtraction modulus difference we have determined the probabilities for all differentials in 8-, 10-, 12-, 14-, 16- and 18-bit $g$-functions.

No clear structures are observed, so the differentials with the largest probabilities cannot be determined for the 32-bit $g$-function. 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 to the case with XOR as differential operator.

We have also briefly investigated higher order differentials, but due to the huge complexity, only $g$-functions with very small word length have been examined. This revealed that 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 Rabbit**

The differentials are extensively investigated in [5], and here we will only present the results.

The largest probability ratios for the $g$-function were found to be at least $2^{-11.57}$ for XOR differences and probably around $2^{-17}$ for subtraction modulus differences. For first order differentials it was illustrated that any characteristic will involve at least 8 $g$-functions[5].

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. Us-

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

ing 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 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.

In conclusion, we have not found any differential weaknesses of the IV-setup scheme of Rabbit.

### 3.8 Statistical Tests

The statistical tests on Rabbit were performed using the NIST Test Suite [17], the DIEHARD battery of tests [14] and the ENT test [22]. 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.

## 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 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 III Architecture**
The performance was measured on a 1000 MHz Pentium III 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 (where calling conventions are ignored) can eliminate the need for memory, including the instance structure, 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[6]. 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.

| Processor | Performance | Code size | Memory |
|---|---|---|---|
| Pentium III | 3.7/278/253 | 440/617/720 | 36/32/40 |
| 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.

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

## 5 Conclusion

In this paper, we described the stream cipher Rabbit previously presented at the Fast Software Encryption workshop in 2003. In this version, we also included a specification of an IV-setup function. We performed a comprehensive algebraic analysis of the cipher, described the best approximations of the next-state function found and performed a thorough differential analysis. None of the investigations have revealed any weaknesses.

The measured encryption/decryption performance was 3.7 clock cycles per byte on a Pentium III processor, 9.6 clock cycles per byte on an ARM7 processor, and 10.9 clock cycles per byte on a MIPS 4Kc 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.

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

## References


1. R. Anderson, E. Biham, and L. Knudsen. A proposal for the Advanced Encryption Standard. http://www.cl.cam.ac.uk/~rja14/serpent.html, 1999.
2. 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.
3. Cryptico A/S. Algebraic analysis of Rabbit. http://www.cryptico.com, 2003. white paper.
4. Cryptico A/S. Differential properties of the g-function. http://www.cryptico.com, 2003. white paper.
5. Cryptico A/S. Security analysis of the IV-setup for Rabbit. http://www.cryptico.com, 2003. white paper.
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. J. Daemen and V. Rijmen. AES proposal: Rijndael. http://csrc.nist.gov/encryption/aes/rijndael/Rijndael.pdf, 1999.
13. E. Filiol. A new statistical testing for symmetric ciphers and hash functions. In R. Deng, S. Qing, F. Bao, and J. Zhou, editors, *Proc. Information and Communications Security 2002*, volume 2513 of *LNCS*, pages 342–353. Springer, 2002.
14. G. Masaglia. A battery of tests for random number generators. http://stat.fsu.edu/~geo/diehard.html, 1996.
15. M. Matsui. Linear cryptanalysis method for DES cipher. In T. Helleseth, editor, *Proc. Eurocrypt '93*, volume 765 of *LNCS*, pages 386–397. Springer, 1993.
16. 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.
17. 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.
18. R. Rueppel. *Analysis and Design of Stream Ciphers*. Springer, 1986.
19. 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.
20. A. Shamir, J. Patarin, N. Courtois, and A. Klimov. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In B. Preneel, editor, *Proc. Eurocrypt 2000*, volume 1807 of *LNCS*, pages 392–407. Springer, 2000.
21. A. Shamir and B. Tsaban. Guaranteeing the diversity of number generators. *Information and Computation*, 171(2):350–363, 2001.
22. J. Walker. A pseudorandom number sequence test program. http://www.fourmilab.ch/random, 1998.

## 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:

```
/******************************************************************************/
/* File name: rabbit.h                                                        */
/*----------------------------------------------------------------------------*/
/* Header file for reference C version of the Rabbit stream cipher.          */
/*----------------------------------------------------------------------------*/
/* Copyright (C) Cryptico A/S. All rights reserved.                           */
/*                                                                            */
/* YOU SHOULD CAREFULLY READ THIS LEGAL NOTICE BEFORE USING THIS SOFTWARE.    */
/*                                                                            */
/* This software is developed by Cryptico A/S and/or its suppliers.          */
/* All title and intellectual property rights in and to the software,        */
/* including but not limited to patent rights and copyrights, are owned by    */
/* Cryptico A/S and/or its suppliers.                                         */
/*                                                                            */
/* The software may be used solely for non-commercial purposes                */
/* without the prior written consent of Cryptico A/S. For further             */
/* information on licensing terms and conditions please contact Cryptico A/S  */
/* at info@cryptico.com                                                       */
/*                                                                            */
/* Cryptico, CryptiCore, the Cryptico logo and "Re-thinking encryption" are   */
/* either trademarks or registered trademarks of Cryptico A/S.                */
/*                                                                            */
/* Cryptico A/S shall not in any way be liable for any use of this software.  */
/* The software is provided "as is" without any express or implied warranty.  */
/*                                                                            */
/******************************************************************************/

#ifndef _RABBIT_H
#define _RABBIT_H

#include <stddef.h>

// Type declarations of 32-bit and 8-bit unsigned integers
typedef unsigned int rabbit_uint32;
typedef unsigned char rabbit_byte;

// Structure to store the instance data (internal state)
typedef struct
{
   rabbit_uint32 x[8];
   rabbit_uint32 c[8];
   rabbit_uint32 carry;
} rabbit_instance;

#ifdef __cplusplus
extern "C" {
#endif
```

```
// All function calls returns zero on success
int rabbit_key_setup(rabbit_instance *p_instance, const rabbit_byte *p_key, size_t key_size);
int rabbit_iv_setup(const rabbit_instance *p_master_instance,
               rabbit_instance *p_instance, const rabbit_byte *p_iv, size_t iv_size);
int rabbit_cipher(rabbit_instance *p_instance, const rabbit_byte *p_src,
               rabbit_byte *p_dest, size_t data_size);

#ifdef __cplusplus
}
#endif

#endif
```

**rabbit.c**

Below the rabbit.c file is listed:

```
/******************************************************************************/
/* File name: rabbit.c                                                        */
/*----------------------------------------------------------------------------*/
/* Source file for reference C version of the Rabbit stream cipher           */
/*----------------------------------------------------------------------------*/
/* Copyright (C) Cryptico A/S. All rights reserved.                           */
/*                                                                            */
/* YOU SHOULD CAREFULLY READ THIS LEGAL NOTICE BEFORE USING THIS SOFTWARE.    */
/*                                                                            */
/* This software is developed by Cryptico A/S and/or its suppliers.           */
/* All title and intellectual property rights in and to the software,         */
/* including but not limited to patent rights and copyrights, are owned by    */
/* Cryptico A/S and/or its suppliers.                                         */
/*                                                                            */
/* The software may be used solely for non-commercial purposes                */
/* without the prior written consent of Cryptico A/S. For further             */
/* information on licensing terms and conditions please contact Cryptico A/S  */
/* at info@cryptico.com                                                       */
/*                                                                            */
/* Cryptico, CryptiCore, the Cryptico logo and "Re-thinking encryption" are   */
/* either trademarks or registered trademarks of Cryptico A/S.                */
/*                                                                            */
/* Cryptico A/S shall not in any way be liable for any use of this software.  */
/* The software is provided "as is" without any express or implied warranty.  */
/*                                                                            */
/******************************************************************************/


#include "rabbit.h"

// Left rotation of a 32-bit unsigned integer
static rabbit_uint32 rabbit_rotl(rabbit_uint32 x, int rot)
{
   return (x<<rot) | (x>>(32-rot));
}
```

```
// Square a 32-bit unsigned integer to obtain the 64-bit result and return
// the 32 high bits XOR the 32 low bits
static rabbit_uint32 rabbit_g_func(rabbit_uint32 x)
{
   // Construct high and low argument for squaring
   rabbit_uint32 a = x&0xFFFF;
   rabbit_uint32 b = x>>16;

   // Calculate high and low result of squaring
   rabbit_uint32 h = ((((a*a)>>17) + (a*b))>>15) + b*b;
   rabbit_uint32 l = x*x;

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


// Calculate the next internal state
static void rabbit_next_state(rabbit_instance *p_instance)
{
   // Temporary data
   rabbit_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] = rabbit_g_func(p_instance->x[i] + p_instance->c[i]);

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

```
// Initialize the cipher instance (*p_instance) as function of the key (*p_key)
int rabbit_key_setup(rabbit_instance *p_instance, const rabbit_byte *p_key, size_t key_size)
{
   // Temporary data
   rabbit_uint32 k0, k1, k2, k3, i;

   // Return error if the key size is not 16 bytes
   if (key_size != 16)
      return -1;

   // Generate four subkeys
   k0 = *(rabbit_uint32*)(p_key+ 0);
   k1 = *(rabbit_uint32*)(p_key+ 4);
   k2 = *(rabbit_uint32*)(p_key+ 8);
   k3 = *(rabbit_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] = rabbit_rotl(k2, 16);
   p_instance->c[2] = rabbit_rotl(k3, 16);
   p_instance->c[4] = rabbit_rotl(k0, 16);
   p_instance->c[6] = rabbit_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 bit
   p_instance->carry = 0;

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

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

   // Return success
   return 0;
}
```

```
// Initialize the cipher instance (*p_instance) as function of the IV (*p_iv)
// and the master instance (*p_master_instance)
int rabbit_iv_setup(const rabbit_instance *p_master_instance,
               rabbit_instance *p_instance, const rabbit_byte *p_iv, size_t iv_size)
{
   // Temporary data
   rabbit_uint32 i0, i1, i2, i3, i;

   // Return error if the IV size is not 8 bytes
   if (iv_size != 8)
      return -1;

   // Generate four subvectors
   i0 = *(rabbit_uint32*)(p_iv+0);
   i2 = *(rabbit_uint32*)(p_iv+4);
   i1 = (i0>>16) | (i2&0xFFFF0000);
   i3 = (i2<<16) | (i0&0x0000FFFF);

   // Modify counter values
   p_instance->c[0] = p_master_instance->c[0] ^ i0;
   p_instance->c[1] = p_master_instance->c[1] ^ i1;
   p_instance->c[2] = p_master_instance->c[2] ^ i2;
   p_instance->c[3] = p_master_instance->c[3] ^ i3;
   p_instance->c[4] = p_master_instance->c[4] ^ i0;
   p_instance->c[5] = p_master_instance->c[5] ^ i1;
   p_instance->c[6] = p_master_instance->c[6] ^ i2;
   p_instance->c[7] = p_master_instance->c[7] ^ i3;

   // Copy internal state values
   for (i=0; i<8; i++)
      p_instance->x[i] = p_master_instance->x[i];
   p_instance->carry = p_master_instance->carry;

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

   // Return success
   return 0;
}
```

```
// Encrypt or decrypt a block of data
int rabbit_cipher(rabbit_instance *p_instance, const rabbit_byte *p_src,
               rabbit_byte *p_dest, size_t data_size)
{
   // Temporary data
   rabbit_uint32 i;

   // Return error if the size of the data to encrypt is
   // not a multiple of 16
   if (data_size%16)
      return -1;

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

      // Encrypt 16 bytes of data
      *(rabbit_uint32*)(p_dest+ 0) = *(rabbit_uint32*)(p_src+ 0) ^ p_instance->x[0] ^
                          (p_instance->x[5]>>16) ^ (p_instance->x[3]<<16);
      *(rabbit_uint32*)(p_dest+ 4) = *(rabbit_uint32*)(p_src+ 4) ^ p_instance->x[2] ^
                          (p_instance->x[7]>>16) ^ (p_instance->x[5]<<16);
      *(rabbit_uint32*)(p_dest+ 8) = *(rabbit_uint32*)(p_src+ 8) ^ p_instance->x[4] ^
                          (p_instance->x[1]>>16) ^ (p_instance->x[7]<<16);
      *(rabbit_uint32*)(p_dest+12) = *(rabbit_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;
   }

   // Return success
   return 0;
}
```

## B Testing

Below some test vectors are presented. A program testing the keys, IV's and the corresponding output is included in the zip-file. The keys and outputs are presented byte-wise. The leftmost byte of `key` is $K^{[7..0]}$.

### B.1 Test Vectors

```
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[2] =  [96 D6 73 16 88 D1 68 DA 51 D4 0C 70 C3 A1 16 F4]


key  =  [AC C3 51 DC F1 62 FC 3B FE 36 3D 2E 29 13 28 91]

s[0] =  [9C 51 E2 87 84 C3 7F E9 A1 27 F6 3E C8 F3 2D 3D]
s[1] =  [19 FC 54 85 AA 53 BF 96 88 5B 40 F4 61 CD 76 F5]
s[2] =  [5E 4C 4D 20 20 3B E5 8A 50 43 DB FB 73 74 54 E5]


key  =  [43 00 9B C0 01 AB E9 E9 33 C7 E0 87 15 74 95 83]

s[0] =  [9B 60 D0 02 FD 5C EB 32 AC CD 41 A0 CD 0D B1 0C]
s[1] =  [AD 3E FF 4C 11 92 70 7B 5A 01 17 0F CA 9F FC 95]
s[2] =  [28 74 94 3A AD 47 41 92 3F 7F FC 8B DE E5 49 96]
```

### B.2 IV Test Vectors

```
mkey =  [00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00]

iv   =  [00 00 00 00 00 00 00 00]

s[0] =  [ED B7 05 67 37 5D CD 7C D8 95 54 F8 5E 27 A7 C6]
s[1] =  [8D 4A DC 70 32 29 8F 7B D4 EF F5 04 AC A6 29 5F]
s[2] =  [66 8F BF 47 8A DB 2B E5 1E 6C DE 29 2B 82 DE 2A]


iv   =  [59 7E 26 C1 75 F5 73 C3]

s[0] =  [6D 7D 01 22 92 CC DC E0 E2 12 00 58 B9 4E CD 1F]
s[1] =  [2E 6F 93 ED FF 99 24 7B 01 25 21 D1 10 4E 5F A7]
s[2] =  [A7 9B 02 12 D0 BD 56 23 39 38 E7 93 C3 12 C1 EB]


iv   =  [27 17 F4 D2 1A 56 EB A6]

s[0] =  [4D 10 51 A1 23 AF B6 70 BF 8D 85 05 C8 D8 5A 44]
s[1] =  [03 5B C3 AC C6 67 AE AE 5B 2C F4 47 79 F2 C8 96]
s[2] =  [CB 51 15 F0 34 F0 3D 31 17 1C A7 5F 89 FC CB 9F]
```