The Rabbit Stream Cipher

Martin Boesgaard, Mette Vesterager, Thomas Christensen, 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 [5], and no attacks against it have been published until now. With a measured encryption/decryption speed of 3.7 clock cycles per byte on a Pentium III processor, Rabbit does also provide very high performance. Thus, the Rabbit design is currently submitted to the Ecrypt call for stream cipher primitives. This paper gives a concise description of the Rabbit design (including the IV setup) and of the cryptanalytic results available.
Keywords: Stream cipher, fast, non-linear, coupled, counter

1 Introduction

Rabbit was first presented at the Fast Software Encryption workshop in 2003 [5]. Since then, an IV-setup function has been designed [16], and additional security analysis has been completed, but no cryptographical weaknesses have been revealed. The cipher is currently being submitted to the Ecrypt call for stream cipher primitives, negotiating further evaluation by cryptographic experts. Thus, this paper gives a short introduction to the prior work on Rabbit.

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.

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}. This coupled system is illustrated in Fig. 1. Before an iteration the counters are incremented as described below.

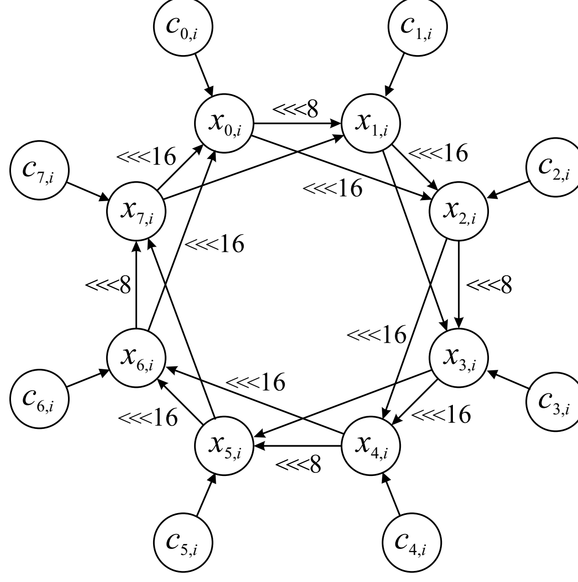

Fig. 1. Graphical illustration of the next-state function.

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

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.

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

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.

[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.

– 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.

3.3 Period Length

A central property of counter assisted stream ciphers [17] is that strict lower bounds on the period lengths can be provided. The counter system adopted in Rabbit has a period length of $2^{256} - 1$ [5]. 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 [16].

3.4 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 [5], 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 3.6). 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 [5] 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.

3.5 Algebraic Attacks

Known Algebraic Attacks: The algebraic attacks on stream ciphers discussed in the literature [1, 7, 8, 6, 9] target ciphers whose internal state is mainly updated in a linear way, with only a few memory bits having a nonlinear update function. This, however, is not the case for Rabbit, where 256 inner state bits are updated in a strongly nonlinear fashion. In the following, we will discuss in some detail the nonlinearity 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., [15]). 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 nonlinear 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.

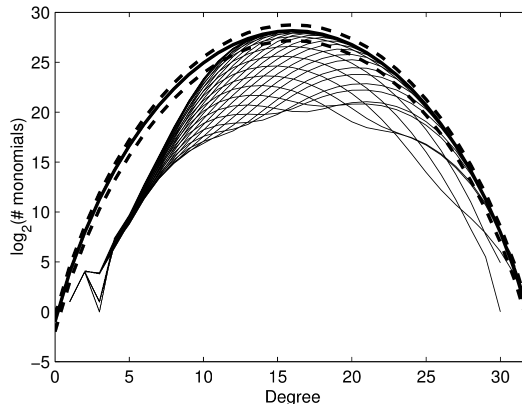

Fig. 2. The number of monomials of each degree in each of the 32 Boolean functions of the g-function. The thick solid line and the two dashed lines denote the average and variance for an ideal random function.

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, according to Equ. (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. 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.

3.6 Correlation Attacks

Linear Approximations: In [5], at thorough investigation of linear approximations by use of the Walsh-Hadamard Transform [15, 10] 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 [12].

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. [13]) 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].

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.

3.7 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.

[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.

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 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.

3.8 Statistical Tests

The statistical tests on Rabbit were performed using the NIST Test Suite [14], the DIEHARD battery of tests [11] and the ENT test [18]. 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

4.1 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

[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.

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.

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.

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.

4.2 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 and IV setup.

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

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.

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 (e.g., Xilinx Spartan 3 or Altera Cyclone II). 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 using 6 muliplier units, this will give us decryption performance of 8.9 Gbit/s and 5.3 Gbit/s respectively. If more multipliers are available, the number of pipelines can be increased, and throughputs of 17.8 Gbit/s and 10.7 Gbit/s will be achievable.

5 Conclusion

The stream cipher Rabbit was first presented at FSE 2003 [5], and no attacks against it have been published until now. With a measured encryption/decryption speed of 3.7 clock cycles per byte on a Pentium III processor, Rabbit does also provide very high performance. Thus, the Rabbit design is currently submitted to the Ecrypt call for stream cipher primitives. In this paper, we gave a concise description of the Rabbit design (including the IV setup) and of the cryptanalytic results available.

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.

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. 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.
6. 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.
7. 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.
8. 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.
9. 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.
10. J. Daemen. *Cipher and hash function design strategies based on linear and differential cryptanalysis.* PhD thesis, KU Leuven, March 1995.

11. G. Masaglia. A battery of tests for random number generators. http://stat.fsu.edu/~geo/diehard.html, 1996.
12. M. Matsui. Linear cryptanalysis method for DES cipher. In T. Helleseth, editor, *Proc. Eurocrypt '93*, volume 765 of *LNCS*, pages 386–397. Springer, 1993.
13. 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.
14. 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.
15. R. Rueppel. *Analysis and Design of Stream Ciphers.* Springer, 1986.
16. 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.
17. A. Shamir and B. Tsaban. Guaranteeing the diversity of number generators. *Information and Computation*, 171(2):350–363, 2001.
18. 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;
}
```

```c
// 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]
```