Network Working Group
Request for Comments: 4503
Category: Informational

M. Boesgaard
M. Vesterager
E. Zenner
Cryptico A/S

May 2006

A Description of the Rabbit Stream Cipher Algorithm

Status of This Memo

This memo provides information for the Internet community. It does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

Copyright Notice

Copyright (C) The Internet Society (2006).

Abstract

This document describes the encryption algorithm Rabbit. It is a stream cipher algorithm with a 128-bit key and 64-bit initialization vector (IV). The method was published in 2003 and has been subject to public security and performance revision. Its high performance makes it particularly suited for the use with Internet protocols where large amounts of data have to be processed.

Table of Contents

1. Introduction

Rabbit is a stream cipher algorithm that has been designed for high performance in software implementations. Both key setup and encryption are very fast, making the algorithm particularly suited for all applications where large amounts of data or large numbers of data packages have to be encrypted. Examples include, but are not limited to, server-side encryption, multimedia encryption, hard-disk encryption, and encryption on limited-resource devices.

The cipher is based on ideas derived from the behavior of certain chaotic maps. These maps have been carefully discretized, resulting in a compact stream cipher. Rabbit has been openly published in 2003 [1] and has not displayed any weaknesses as of the time of this writing. To ensure ongoing security evaluation, it was also submitted to the ECRYPT eSTREAM project[2].

Technically, Rabbit consists of a pseudorandom bitstream generator that takes a 128-bit key and a 64-bit initialization vector (IV) as input and generates a stream of 128-bit blocks. Encryption is performed by combining this output with the message, using the exclusive-OR operation. Decryption is performed in exactly the same way as encryption.

Further information about Rabbit, including reference implementation, test vectors, performance figures, and security white papers, is available from http://www.cryptico.com/.

2. Algorithm Description

2.1. Notation

This document uses the following elementary operators:

```
 +     integer addition.
 *     integer multiplication.
div    integer division.
mod    integer modulus.
 ^     bitwise exclusive-OR operation.
<<<    left rotation operator.
 ||    concatenation operator.
```

When labeling bits of a variable, A, the least significant bit is denoted by A[0]. The notation A[h..g] represents bits h through g of variable A, where h is more significant than g. Similar variables are labeled by A0,A1,... with the notation A(0),A(1),... being used to denote those same variables if this improves readability.

Given a 64-bit word, the function MSW extracts the most significant 32 bits, whereas the function LSW extracts the least significant 32 bits.

Constants prefixed with 0x are in hexadecimal notation. In particular, the constant WORDSIZE is defined to be 0x100000000.

2.2. Inner State

The internal state of the stream cipher consists of 513 bits. 512 bits are divided between eight 32-bit state variables, X0,...,X7 and eight 32-bit counter variables, C0,...,C7. In addition, there is one counter carry bit, b.

2.3. Key Setup Scheme

The counter carry bit b is initialized to zero. The state and counter words are derived from the key K[127..0].

The key is divided into subkeys K0 = K[15..0], K1 = K[31..16], ... K7 = K[127..112]. The initial state is initialized as follows:

```
for j=0 to 7:
  if j is even:
    Xj = K(j+1 mod 8) || Kj
    Cj = K(j+4 mod 8) || K(j+5 mod 8)
  else:
    Xj = K(j+5 mod 8) || K(j+4 mod 8)
    Cj = Kj || K(j+1 mod 8)
```

The system is then iterated four times, each iteration consisting of counter update (Section 2.5) and next-state function (Section 2.6). After that, the counter variables are reinitialized to

```
for j=0 to 7:
  Cj = Cj ^ X(j+4 mod 8)
```

2.4. IV Setup Scheme

If an IV is used for encryption, the counter variables are modified after the key setup. Denoting the IV bits by IV[63..0], the setup proceeds as follows:

```
C0 = C0 ^ IV[31..0]          C1 = C1 ^ (IV[63..48] || IV[31..16])
C2 = C2 ^ IV[63..32]         C3 = C3 ^ (IV[47..32] || IV[15..0])
```

```
C4 = C4 ^ IV[31..0]         C5 = C5 ^ (IV[63..48] || IV[31..16])
C6 = C6 ^ IV[63..32]        C7 = C7 ^ (IV[47..32] || IV[15..0])
```

The system is then iterated another 4 times, each iteration consisting of counter update (Section 2.5) and next-state function (Section 2.6).

The relationship between key and IV setup is as follows:

- After the key setup, the resulting inner state is saved as a master state. Then the IV setup is run to obtain the first encryption starting state.

- Whenever re-initialization under a new IV is necessary, the IV setup is run on the master state again to derive the next encryption starting state.

2.5. Counter System

Before each execution of the next-state function (Section 2.6), the counter system has to be updated. This system uses constants A1,...,A7, as follows:

```
A0 = 0x4D34D34D        A1 = 0xD34D34D3
A2 = 0x34D34D34        A3 = 0x4D34D34D
A4 = 0xD34D34D3        A5 = 0x34D34D34
A6 = 0x4D34D34D        A7 = 0xD34D34D3
```

It also uses the counter carry bit b to update the counter system, as follows:

```
for j=0 to 7:
  temp = Cj + Aj + b
  b    = temp div WORDSIZE
  Cj   = temp mod WORDSIZE
```

Note that on exiting this loop, the variable b has to be preserved for the next iteration of the system.

2.6. Next-State Function

The core of the Rabbit algorithm is the next-state function. It is based on the function g, which transforms two 32-bit inputs into one 32-bit output, as follows:

```
g(u,v) = LSW(square(u+v)) ^ MSW(square(u+v))
```

where square(u+v) = ((u+v mod WORDSIZE) * (u+v mod WORDSIZE)).

Using this function, the algorithm updates the inner state as follows:

```
for j=0 to 7:
  Gj = g(Xj,Cj)

X0 = G0 + (G7 <<< 16) + (G6 <<< 16) mod WORDSIZE
X1 = G1 + (G0 <<<  8) +  G7         mod WORDSIZE
X2 = G2 + (G1 <<< 16) + (G0 <<< 16) mod WORDSIZE
X3 = G3 + (G2 <<<  8) +  G1         mod WORDSIZE
X4 = G4 + (G3 <<< 16) + (G2 <<< 16) mod WORDSIZE
X5 = G5 + (G4 <<<  8) +  G3         mod WORDSIZE
X6 = G6 + (G5 <<< 16) + (G4 <<< 16) mod WORDSIZE
X7 = G7 + (G6 <<<  8) +  G5         mod WORDSIZE
```

2.7. Extraction Scheme

After the key and IV setup are concluded, the algorithm is iterated in order to produce one 128-bit output block, S, per round. Each round consists of executing steps 2.5 and 2.6 and then extracting an output S[127..0] as follows:

```
S[15..0]     = X0[15..0]  ^ X5[31..16]
S[31..16]    = X0[31..16] ^ X3[15..0]
S[47..32]    = X2[15..0]  ^ X7[31..16]
S[63..48]    = X2[31..16] ^ X5[15..0]
S[79..64]    = X4[15..0]  ^ X1[31..16]
S[95..80]    = X4[31..16] ^ X7[15..0]
S[111..96]   = X6[15..0]  ^ X3[31..16]
S[127..112]  = X6[31..16] ^ X1[15..0]
```

2.8. Encryption/Decryption Scheme

Given a 128-bit message block, M, encryption E and decryption M' are computed via

```
E  = M ^ S    and
M' = E ^ S.
```

If S is the same in both operations (as it should be if the same key and IV are used), then M = M'.

The encryption/decryption scheme is repeated until all blocks in the message have been encrypted/decrypted. If the message size is not a multiple of 128 bits, only the needed amount of least significant bits from the last output block S is used for the last message block M.

If the application requires the encryption of smaller blocks (or even individual bits), a 128-bit buffer is used. The buffer is initialized by generating a new value, S, and copying it into the buffer. After that, all data blocks are encrypted using the least significant bits in this buffer. Whenever the buffer is empty, a new value S is generated and copied into the buffer.

3. Security Considerations

For an encryption algorithm, the security provided is, of course, the most important issue. No security weaknesses have been found to date, neither by the designers nor by independent cryptographers scrutinizing the algorithms after its publication in [1]. Note that a full discussion of Rabbit's security against known cryptanalytic techniques is provided in [3].

In the following, we restrict ourselves to some rules on how to use the Rabbit algorithm properly.

3.1. Message Length

Rabbit was designed to encrypt up to 2 to the power of 64 128-bit message blocks under the same the key. Should this amount of data ever be exceeded, the key has to be replaced. It is recommended to follow this rule even when the IV is changed on a regular basis.

3.2. Initialization Vector

It is possible to run Rabbit without the IV setup. However, in this case, the generator must never be reset under the same key, since this would destroy its security (for a recent example, see [4]). However, in order to guarantee synchronization between sender and receiver, ciphers are frequently reset in practice. This means that both sender and receiver set the inner state of the cipher back to a known value and then derive the new encryption state using an IV. If this is done, it is important to make sure that no IV is ever reused under the same key.

4. Informative References

[1] M. Boesgaard, M. Vesterager, T. Pedersen, J. Christiansen, O. Scavenius. "Rabbit: A New High-Performance Stream Cipher". Proc. Fast Software Encryption 2003, Lecture Notes in Computer Science 2887, p. 307-329. Springer, 2003.

[2] ECRYPT eSTREAM project, available from http://www.ecrypt.eu.org/stream/

[3] M. Boesgaard, T. Pedersen, M. Vesterager, E. Zenner. "The Rabbit Stream Cipher - Design and Security Analysis". Proc. SASC Workshop 2004, available from http://www.isg.rhul.ac.uk/research/projects/ecrypt/stvl/sasc.html.

[4] H. Wu. "The Misuse of RC4 in Microsoft Word and Excel". IACR eprint archive 2005/007, available from http://eprint.iacr.org/2005/007.pdf.

[5] Jonsson, J. and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, February 2003.

Appendix A: Test Vectors

This is a set of test vectors for conformance testing, given in octet form. For use with Rabbit, they have to be transformed into integers by the conversion primitives OS2IP and I2OSP, as described in [5].

A.1. Testing without IV Setup

```
key  = [00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00]
S[0] = [B1 57 54 F0 36 A5 D6 EC F5 6B 45 26 1C 4A F7 02]
S[1] = [88 E8 D8 15 C5 9C 0C 39 7B 69 6C 47 89 C6 8A A7]
S[2] = [F4 16 A1 C3 70 0C D4 51 DA 68 D1 88 16 73 D6 96]

key  = [91 28 13 29 2E 3D 36 FE 3B FC 62 F1 DC 51 C3 AC]
S[0] = [3D 2D F3 C8 3E F6 27 A1 E9 7F C3 84 87 E2 51 9C]
S[1] = [F5 76 CD 61 F4 40 5B 88 96 BF 53 AA 85 54 FC 19]
S[2] = [E5 54 74 73 FB DB 43 50 8A E5 3B 20 20 4D 4C 5E]

key  = [83 95 74 15 87 E0 C7 33 E9 E9 AB 01 C0 9B 00 43]
S[0] = [0C B1 0D CD A0 41 CD AC 32 EB 5C FD 02 D0 60 9B]
S[1] = [95 FC 9F CA 0F 17 01 5A 7B 70 92 11 4C FF 3E AD]
S[2] = [96 49 E5 DE 8B FC 7F 3F 92 41 47 AD 3A 94 74 28]
```

A.2. Testing with IV Setup

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

iv   = [C3 73 F5 75 C1 26 7E 59]
S[0] = [1F CD 4E B9 58 00 12 E2 E0 DC CC 92 22 01 7D 6D]
S[1] = [A7 5F 4E 10 D1 21 25 01 7B 24 99 FF ED 93 6F 2E]
S[2] = [EB C1 12 C3 93 E7 38 39 23 56 BD D0 12 02 9B A7]

iv   = [A6 EB 56 1A D2 F4 17 27]
S[0] = [44 5A D8 C8 05 85 8D BF 70 B6 AF 23 A1 51 10 4D]
S[1] = [96 C8 F2 79 47 F4 2C 5B AE AE 67 C6 AC C3 5B 03]
S[2] = [9F CB FC 89 5F A7 1C 17 31 3D F0 34 F0 15 51 CB]
```

Appendix B: Debugging Vectors

The following set of vectors describes the inner state of Rabbit during key and iv setup. It is meant mainly for debugging purposes. Octet strings are written according to I2OSP conventions.

B.1. Testing Round Function and Key Setup

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

Inner state after key expansion:
b  = 0
X0 = 0xDC51C3AC, X1 = 0x13292E3D, X2 = 0x3BFC62F1, X3 = 0xC3AC9128,
X4 = 0x2E3D36FE, X5 = 0x62F1DC51, X6 = 0x91281329, X7 = 0x36FE3BFC,
C0 = 0x36FE2E3D, C1 = 0xDC5162F1, C2 = 0x13299128, C3 = 0x3BFC36FE,
C4 = 0xC3ACDC51, C5 = 0x2E3D1329, C6 = 0x62F13BFC, C7 = 0x9128C3AC

Inner state after first key setup iteration:
b  = 1
X0 = 0xF2E8C8B1, X1 = 0x38E06FA7, X2 = 0x9A0D72C0, X3 = 0xF21F5334,
X4 = 0xCACDCCC3, X5 = 0x4B239CBE, X6 = 0x0565DCCC, X7 = 0xB1587C8D,
C0 = 0x8433018A, C1 = 0xAF9E97C4, C2 = 0x47FCDE5D, C3 = 0x89310A4B,
C4 = 0x96FA1124, C5 = 0x6310605E, C6 = 0xB0260F49, C7 = 0x6475F87F

Inner state after fourth key setup iteration:
b  = 0
X0 = 0x1D059312, X1 = 0xBDDC3E45, X2 = 0xF440927D, X3 = 0x50CBB553,
X4 = 0x36709423, X5 = 0x0B6F0711, X6 = 0x3ADA3A7B, X7 = 0xEB9800C8,
C0 = 0x6BD17B74, C1 = 0x2986363E, C2 = 0xE676C5FC, C3 = 0x70CF8432,
C4 = 0x10E1AF9E, C5 = 0x018A47FD, C6 = 0x97C48931, C7 = 0xDE5D96F9

Inner state after final key setup xor:
b  = 0
X0 = 0x1D059312, X1 = 0xBDDC3E45, X2 = 0xF440927D, X3 = 0x50CBB553,
X4 = 0x36709423, X5 = 0x0B6F0711, X6 = 0x3ADA3A7B, X7 = 0xEB9800C8,
C0 = 0x5DA1EF57, C1 = 0x22E9312F, C2 = 0xDCACFF87, C3 = 0x9B5784FA,
C4 = 0x0DE43C8C, C5 = 0xBC5679B8, C6 = 0x63841B4C, C7 = 0x8E9623AA

Inner state after generation of 48 bytes of output:
b  = 1
X0 = 0xB5428566, X1 = 0xA2593617, X2 = 0xFF5578DE, X3 = 0x7293950F,
X4 = 0x145CE109, X5 = 0xC93875B0, X6 = 0xD34306E0, X7 = 0x43FEEF87,
C0 = 0x45406940, C1 = 0x9CD0CFA9, C2 = 0x7B26E725, C3 = 0x82F5FEE2,
C4 = 0x87CBDB06, C5 = 0x5AD06156, C6 = 0x4B229534, C7 = 0x087DC224
```

```
The 48 output bytes:
S[0] = [3D 2D F3 C8 3E F6 27 A1 E9 7F C3 84 87 E2 51 9C]
S[1] = [F5 76 CD 61 F4 40 5B 88 96 BF 53 AA 85 54 FC 19]
S[2] = [E5 54 74 73 FB DB 43 50 8A E5 3B 20 20 4D 4C 5E]
```

B.2. Testing the IV Setup

```
key  = [91 28 13 29 2E ED 36 FE 3B FC 62 F1 DC 51 C3 AC]
iv   = [C3 73 F5 75 C1 26 7E 59]

Inner state during key setup:
as above

Inner state after IV expansion:
b  = 0
X0 = 0x1D059312, X1 = 0xBDDC3E45, X2 = 0xF440927D, X3 = 0x50CBB553,
X4 = 0x36709423, X5 = 0x0B6F0711, X6 = 0x3ADA3A7B, X7 = 0xEB9800C8,
C0 = 0x9C87910E, C1 = 0xE19AF009, C2 = 0x1FDF0AF2, C3 = 0x6E22FAA3,
C4 = 0xCCC242D5, C5 = 0x7F25B89E, C6 = 0xA0F7EE39, C7 = 0x7BE35DF3

Inner state after first IV setup iteration:
b  = 1
X0 = 0xC4FF831A, X1 = 0xEF5CD094, X2 = 0xC5933855, X3 = 0xC05A5C03,
X4 = 0x4A50522F, X5 = 0xDF487BE4, X6 = 0xA45FA013, X7 = 0x05531179,
C0 = 0xE9BC645B, C1 = 0xB4E824DC, C2 = 0x54B25827, C3 = 0xBB57CDF0,
C4 = 0xA00F77A8, C5 = 0xB3F905D3, C6 = 0xEE2CC186, C7 = 0x4F3092C6

Inner state after fourth IV setup iteration:
b  = 1
X0 = 0x6274E424, X1 = 0xE14CE120, X2 = 0xDA8739D9, X3 = 0x65E0402D,
X4 = 0xD1281D10, X5 = 0xBD435BAA, X6 = 0x4E9E7A02, X7 = 0x9B467ABD,
C0 = 0xD15ADE44, C1 = 0x2ECFC356, C2 = 0xF32C3FC6, C3 = 0xA2F647D7,
C4 = 0x19F71622, C5 = 0x5272ED72, C6 = 0xD5CB3B6E, C7 = 0xC9183140
```

Authors' Addresses

Martin Boesgaard
Cryptico A/S
Fruebjergvej 3
2100 Copenhagen
Denmark

Phone: +45 39 17 96 06
EMail: mab@cryptico.com
URL: http://www.cryptico.com

Mette Vesterager
Cryptico A/S
Fruebjergvej 3
2100 Copenhagen
Denmark

Phone: +45 39 17 96 06
EMail: mvp@cryptico.com
URL: http://www.cryptico.com

Erik Zenner
Cryptico A/S
Fruebjergvej 3
2100 Copenhagen
Denmark

Phone: +45 39 17 96 06
EMail: ez@cryptico.com
URL: http://www.cryptico.com

Full Copyright Statement

Copyright (C) The Internet Society (2006).

This document is subject to the rights, licenses and restrictions contained in BCP 78, and except as set forth therein, the authors retain all their rights.

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Intellectual Property

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the procedures with respect to rights in RFC documents can be found in BCP 78 and BCP 79.

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at http://www.ietf.org/ipr.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at ietf-ipr@ietf.org.

Acknowledgement

Funding for the RFC Editor function is provided by the IETF Administrative Support Activity (IASA).