Network Working Group
Internet-Draft
Intended status: Informational
Expires: 1 August 2026

Y. Nakano
K. Fukushima
KDDI Research, Inc.
T. Isobe
The University of Osaka

28 January 2026

Encryption algorithm Rocca-S
draft-nakano-rocca-s-06

Abstract

This document defines Rocca-S encryption scheme, which is an Authenticated Encryption with Associated Data (AEAD), using a 256-bit key and can be efficiently implemented utilizing the AES New Instruction set (AES-NI).

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 1 August 2026.

Table of Contents

1. Introduction

1.1. Background

Countries such as the USA, China, and South Korea are adapting to the fifth-generation mobile communication systems (5G) technology at an increasingly rapid pace. There are more than 1500 cities worldwide with access to 5G technology. Other countries are also taking significant steps to make 5G networks commercially available to their citizens. As the research in 5G technology is moving toward global standardization, it is important for the research community to focus

on developing solutions beyond 5G and for the 6G era. The first white paper on 6G [WP-6G] was published by 6G Flagship, University of Oulu, Finland under the 6Genesis project in 2019. This white paper identified the key drivers, research requirements, challenges, and essential research questions related to 6G. One of the main requirements as listed in this paper was to look at the problem of transmitting data at a speed of over 100 Gbps per user.

Additionally, 3GPP requires that the cryptographic algorithms proposed for 5G systems should support 256-bit keys [SPEC-5G]. Apart from the need for speeds of more than 100 Gbps and supporting 256-bit keys, 3GPP also discusses the possible impacts of quantum computing in the coming years, especially due to Grover’s algorithm. While describing the impact of quantum computers on symmetric algorithms required for 5G and beyond, 3GPP states the following in Section 5.3 of [SPEC-5G]:

"The threat to symmetric cryptography from quantum computing is lower than that for asymmetric cryptography. As such there is little benefit in transitioning symmetric algorithms without corresponding changes to the asymmetric algorithms that accompany them."

However, it has been shown in numerous articles that quantum computers can be used to either efficiently break or drastically reduce the time necessary to attack some symmetric-key cryptography methods. These results require a serious reevaluation of the premise that has informed beyond 5G quantum security concerns up to this point. Additionally, since NIST will finally standardize quantum-resistant public key algorithms in the coming few years, we believe it is important for the research community to also focus on symmetric algorithms for future telecommunications that would provide security against quantum adversaries. The effectiveness of post-quantum asymmetric cryptography would only be improved if the symmetric cryptography used with it is also quantum resistant. Thus, a symmetric cryptographic algorithm that

* supports 256-bit key and provides 256-bit security with respect to key recovery and forgery attacks,

* has an encryption/decryption speed of more than 100 Gbps, and

* is at least as secure as AES-256 against quantum adversaries (for 128-bit security against a quantum adversary)

is needed.

Rocca-S has been designed as an encryption algorithm for a high speed communication such as future internet and beyond 5G mobile communications. Rocca-S achieves an encryption/decryption speed of more than 200 Gbps in both the raw encryption scheme and the AEAD scheme on an Intel(R) Core(TM) i9-12900K. It can provide 256-bit and 128-bit security against key recovery attacks in classical and quantum adversaries respectively. The high throughput of Rocca-S can be achieved by utilizing the AES-NI [AES-NI]. A similar approach has been taken by the AEGIS family [AEGIS] and Tiaoxin-346 [TIAOXIN], both two submissions to the CAESAR competition [CAESAR]. SNOW-V [SNOW-V] also uses the AES round function as a component so AES-NI can be used.

1.2. Design Concept

In this document, we present an AES-based AEAD encryption scheme with a 256-bit key and 256-bit tag called Rocca-S.

To achieve such a dramatically fast encryption/decryption speed, Rocca-S adopts the design principles such as the SIMD-friendly round function and an efficient permutation-based structure. We explore the class of AES-based structures to further increase its speed and reduce the state size. Specifically, we take the following different approaches:

* To minimize the critical path of the round function, we focus on the structure where each 128-bit block of the internal state is updated by either one AES round (aesenc) or XOR while Jean and Nikolic consider the case of applying both aesenc and XOR in a cascade way for one round.

* We introduce a permutation between the 128-bit state words of the internal state in order to increase the number of possible candidates while maintaining efficiency because executing such a permutation is a cost-free operation in the target software, which was not taken into account in [DESIGN].

1.3. Conventions Used in This Document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Algorithm Description

In this section, the notations and the specification of our designs will be described.

2.1. Notations

The following notations will be used in the document. Throughout this document, a block means a 16-octet value. For the constants Z0 and Z1, we utilize the same ones as Tiaoxin-346 [TIAOXIN].

1. X ^ Y: The bitwise Exclusive OR (XOR) of X and Y.

2. X#Y: For a number X and a positive integer Y, the Y-th power of X.

3. f#(N): For a function f and a non-negative integer N, the N-th iteration of function f.

4. |X|: The length of X in bits.

5. X||Y : The concatenation of X and Y.

6. ZERO(l): A zero string of length l bits.

7. PAD(X): X||ZERO(l), where l is the minimal non-negative integer such that |PAD(X)| is a multiple of 256.

8. PADN(X): X||ZERO(l), where l is the minimal non-negative integer such that |PADN(X)| is a multiple of 128.

9. LE128(X): the little-endian encoding of 128-bit integer X.

10. Truncate(X, n): the result of truncating X to n bits.

11. Write X as X = X[0]||X[1]|| ... ||X[n] with |X[i]| = 256, where |X| is multiple of 256 and n is |X|/256 - 1. In addition, X[i] is written as X[i] = X[i]_0||X[i]_1 where X[i]_0 and X[i]_1 are 128-bit.

12. S: The state of Rocca-S, which is composed of 7 blocks, i.e., S = (S[0], S[1], ..., S[6]), where S[i] (0 <= i <= 6) are blocks and S[0] is the first block.

13. Z0: A 128-bit constant block defined as Z0 = 428a2f98d728ae227137449123ef65cd.

14. Z1: A 128-bit constant block defined as Z1 = b5c0fbcfec4d3b2fe9b5dba58189dbbc.

15. A(X): The AES round function without the constant addition operation, as defined below:
A(X) = MixColumns(ShiftRows(SubBytes(X))), where MixColumns, ShiftRows and SubBytes are the same operations as defined in AES [AES].

16. AES(X,Y): One AES round is applied to the block X, where the round constant is Y, as defined below:
AES(X,Y) = A(X) ^ Y.
This operation is the same as aesenc, which is one of the instructions of AES-NI and performs one regular (not the last) round of AES on an input state X with a subkey Y.

17. R(S,X,Y): The round function is used to update the state S, as defined in Section 2.2.

2.2. The Round Function

The input of the round function R(S,X,Y) of Rocca-S consists of the state S and two blocks (X,Y). If denoting the output by Snew, Snew:=R(S,X,Y) can be defined as follows:

```
Snew[0] = S[6] ^ S[1]
Snew[1] = AES(S[0],X)
Snew[2] = AES(S[1],S[0])
Snew[3] = AES(S[2],S[6])
Snew[4] = AES(S[3],Y)
Snew[5] = AES(S[4],S[3])
Snew[6] = AES(S[5],S[4])
```

The corresponding illustration can be found in Figure 1.

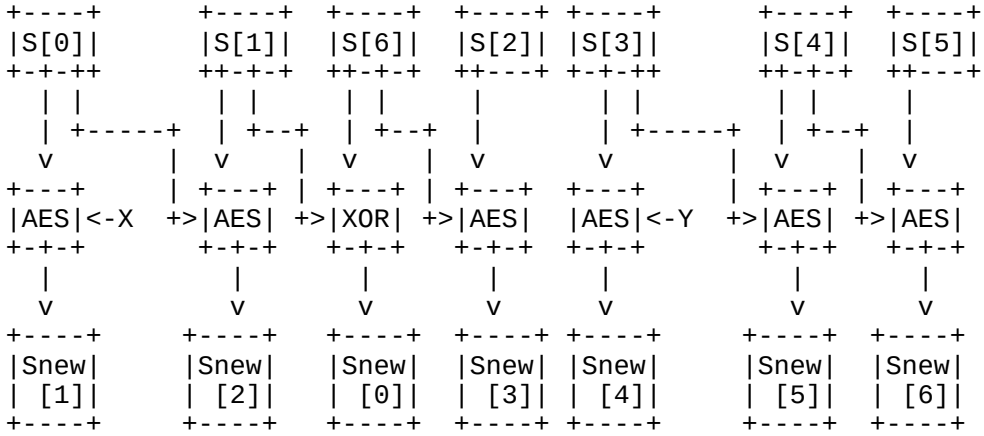

Figure 1: Illustration of the Round Function

2.3. Specification

Rocca-S is an AEAD scheme composed of four phases: initialization, processing the associated data, encryption, and finalization. The input consists of a 256-bit key K = K0||K1, a nonce N of between 12 and 16 octets (both inclusive) in length, the associated data AD, and the message M. The output is the corresponding ciphertext C and a 256-bit tag T.

The settings described below are required for the parameters:

* The key K MUST be generated in a way that is uniformly random or pseudorandom.

* PADN(N), where N is the nonce, which MUST be distinct for any particular value of the key. Recommended nonce formation can be found in [RFC5116].

2.3.1. Initialization

First, (N,K) is loaded into the state S in the following way:

```
K0||K1 = K
S[0] = K1
S[1] = PADN(N)
S[2] = Z0
S[3] = K0
S[4] = Z1
S[5] = PADN(N) ^ K1
S[6] = ZERO(128)
```

Then, 16 iterations of the round function R(S,Z0,Z1), which is written as R(S,Z0,Z1)#(16), are applied to state S.

After 16 iterations of the round function, two 128-bit keys are XORed with the state S in the following way:

```
S[0] = S[0] ^ K0
S[1] = S[1] ^ K0
S[2] = S[2] ^ K1
S[3] = S[3] ^ K0
S[4] = S[4] ^ K0
S[5] = S[5] ^ K1
S[6] = S[6] ^ K1
```

2.3.2. Processing the Associated Data

If AD is empty, this phase will be skipped. Otherwise, AD is padded to PAD(AD), and the state is updated as follows:

```
for i = 0 to d - 1
   R(S, PAD(AD)[i]_0, PAD(AD)[i]_1)
endfor
```

where d = |PAD(AD)| / 256.

2.3.3. Encryption

The encryption phase is similar to the phase to process the associated data. If M is empty, the encryption phase will be skipped. Otherwise, M is first padded to PAD(M), and then PAD(M) will be absorbed with the round function. During this procedure, the ciphertext C is generated. If the last block of M is incomplete and its length is b bits, i.e., 0 < b < 256, the last block of C will be truncated to the first b bits. A detailed description is shown below:

```
for i = 0 to m - 1
   C[i]_0 = AES(S[3] ^ S[5], S[0]) ^ PAD(M)[i]_0
   C[i]_1 = AES(S[4] ^ S[6], S[2]) ^ PAD(M)[i]_1
   R(S, PAD(M)[i]_0, PAD(M)[i]_1)
endfor
```

where m = |PAD(M)| / 256.

2.3.4. Finalization

The state S will again pass through 16 iterations of the round function R(S,LE128(|AD|),LE128(|M|)) and then the 256-bit tag T is computed in the following way:

```
T = (S[0] ^ S[1] ^ S[2] ^ S[3]) || (S[4] ^ S[5] ^ S[6])
```

2.3.5. Rocca-S Algorithm

A formal description of Rocca-S can be seen in Figure 2, and the corresponding illustration is shown in Figure 3.

```
// Rocca-S Algorithm. The specification of Rocca-S
   procedure RoccaEncrypt(K, N, AD, M)
      S = Initialization(K, N)
      if |AD| > 0 then
         S = ProcessAD(S,AD)
      endif
      if |M| > 0 then
         (S, C) =  Encryption(S, M)
         Truncate(C, |M|)
      endif
      T = Finalization(S, |AD|, |M|)
      return (C, T)

   procedure RoccaDecrypt(K, N, AD, C, T)
      S = Initialization(K, N)
      if |AD| > 0 then
         S = ProcessAD(S, AD)
      endif
      if |C| > 0 then
         (M, S) = Decryption(S, C)
         Truncate(M, |C|)
      endif
      if T == Finalization(S, |AD|, |C|) then
         return M
      else
         return nil
      endif

   procedure Initialization(K, N)
      K0||K1 = K
      S[0] = K1
      S[1] = PADN(N)
      S[2] = Z0
      S[3] = K0
      S[4] = Z1
```

```
      S[5] = PADN(N) ^ K1
      S[6] = ZERO(128)
      for i = 0 to 15 do
         S = R(S, Z0, Z1)
      endfor
      S[0] = S[0] ^ K0
      S[1] = S[1] ^ K0
      S[2] = S[2] ^ K1
      S[3] = S[3] ^ K0
      S[4] = S[4] ^ K0
      S[5] = S[5] ^ K1
      S[6] = S[6] ^ K1
      return S

   procedure ProcessAD(S, AD)
      AD = PAD(AD)
      d = |AD|/256
      for i = 0 to d - 1 do
         S =  R(S, AD[i]_0, AD[i]_1)
      endfor
      return S

   procedure Encryption(S, M, C)
      M = PAD(M)
      m = |M|/256
      for i = 0 to m - 1 do
         C[i]_0 = AES(S[3] ^ S[5], S[0]) ^ M[i]_0
         C[i]_1 = AES(S[4] ^ S[6], S[2]) ^ M[i]_1
         S = R(S,M[i]_0, M[i]_1)
      endfor
      return (S, C)

   procedure Decryption(S, C)
      C = PAD(C)
      c = |C|/256
      for i = 0 to c - 1 do
         M[i]_0 = AES(S[3] ^ S[5], S[0]) ^ C[i]_0
         M[i]_1 = AES(S[4] ^ S[6], S[2]) ^ C[i]_1
         S = R(S,M[i]_0, M[i]_1)
      endfor
      return (S, M)

   procedure Finalization(S, ad_len, m_len)
      for i = 0 to 15 do
         S = R(S, LE128(ad_len), LE128(m_len))
      endfor
      T = (S[0] ^ S[1] ^ S[2] ^ S[3]) || (S[4] ^ S[5] ^ S[6])
      return T
```

Figure 2: The Specification of Rocca-S

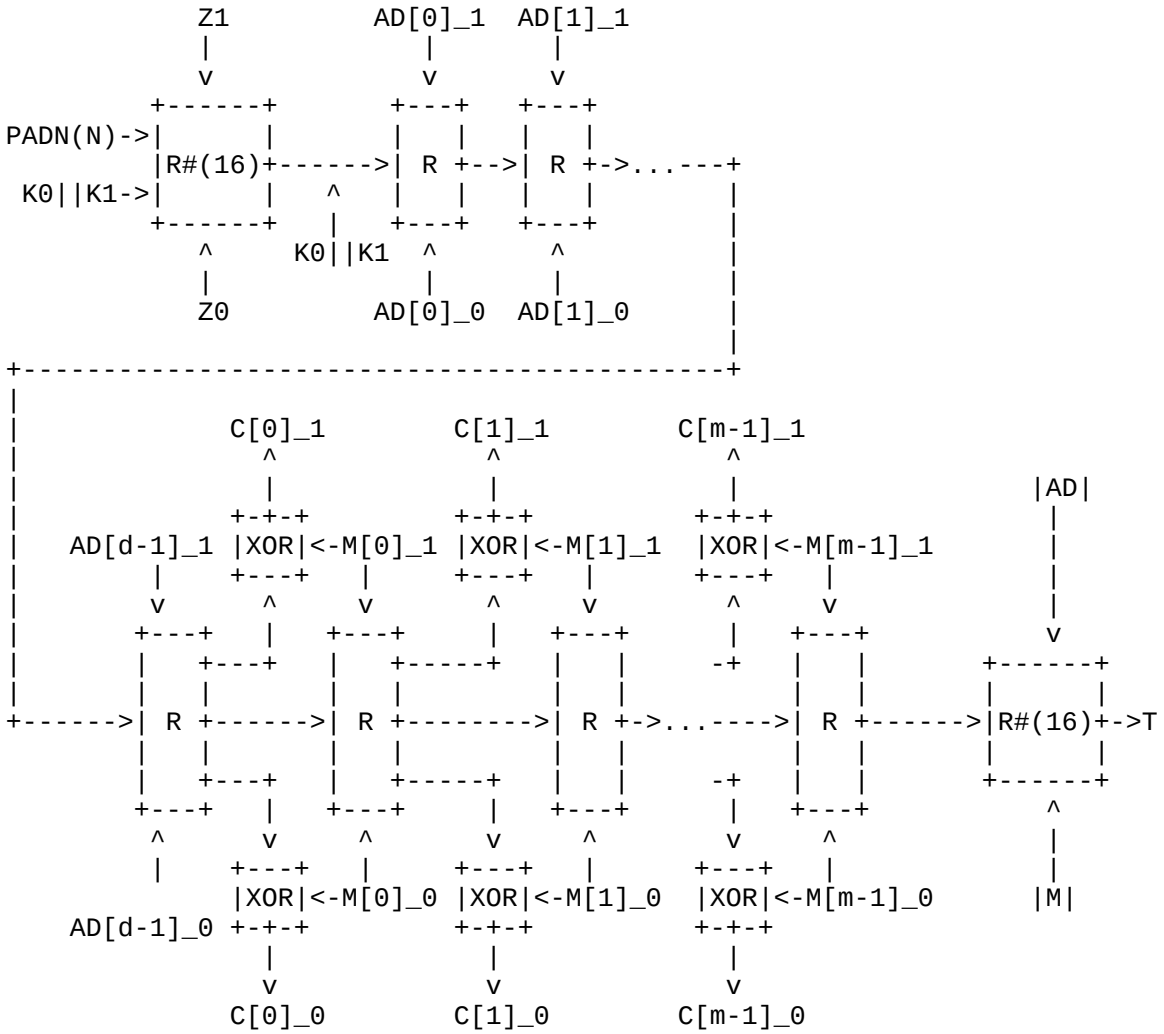

Figure 3: The Procedure of Rocca-S

2.3.6. A Raw Encryption Scheme

If the phases of processing the associated data and finalization are removed, a raw encryption scheme is obtained.

2.3.7. A Keystream Generation Scheme

If the phases of processing the associated data and finalization are removed, and there is no message injection into the round function such that R(S,0,0), a keystream generation scheme is obtained. This scheme can be used as a general stream cipher and for random bit generation.

2.3.8. Support for Shorter Key Length

For Rocca-S to support 128-bit or 192-bit keys, the given key needs to be expanded to 256 bits. When a 128-bit key is given, it will be set to K0, and K1 is defined as K1 = ZERO(128). When a 192-bit key is given, the first 128-bit will be set to K0, and the remaining 64-bit will be set to K1 after padding with zeros.

The use of Key Derivation Functions (KDF) [KDF] to stretch the key length to 256-bit could be another option. The given 128-bit or 192-bit key will be used as a key derivation key, and the output of the KDF will be 256-bit.

When Rocca-S is initialized with shorter key than 256-bit, the initial value for S[6] in the initialization phase will be changed. S[6] is initialized as S[6]= 1||ZERO(127) when 128-bit key is given, and S[6]= 11||ZERO(126) for 192-bit key. The key-length-dependent initial value is introduced thanks to the note by Takeuchi et al. [ePrint2024_901] who pointed out the possibility of the identical initial states due to key expansion.

2.3.9. Settings as AEAD Algorithm Specifications

To comply with the requirements defined in Section 4 of [RFC5116], the settings of the parameters for Rocca-S are defined as follows:

* K_LEN (key length) is 32 octets (256 bits), and K (key) does not require any particular data format.

* P_MAX (maximum size of the plaintext) is 2#125 octets.

* A_MAX (maximum size of the associated data) is 2#61 octets.

* N_MIN (minimum size of the nonce) = 12 octets, and N_MAX (maximum size of the nonce) = 16 octets.

* C_MAX (the largest possible AEAD ciphertext) = P_MAX + tag length = 2#125 + 32 octets.

In addition,

* Rocca-S does not structure its ciphertext output with the authentication tag.

* Rocca-S is not randomized or is not stateful in the meanings of Section 4 of [RFC5116].

2.4. Security Claims

2.4.1. Classic Setting

As described in Section 3, Rocca-S provides 256-bit security against key-recovery and 192-bit security against forgery attacks in the nonce-respecting setting. We do not claim its security in the related-key and known-key settings.

The message length for a fixed key is limited to at most 2#128, and we also limit the number of different messages that are produced for a fixed key to be at most 2#128. The length of the associated data for a fixed key is up to 2#64.

2.4.2. Quantum Setting

Rocca-S provides 128-bit security against key-recovery and forgery attacks against quantum adversary with classical online queries. Rocca-S does not claim security against online quantum superposition attacks.

3. Security Considerations

3.1. Security Against Attacks

Rocca-S is secure against the following attacks:

1. Key-Recovery Attack: 256-bit security against key-recovery attacks.

2. Differential Attack: Secure against differential attacks in the initialization phase.

3. Forgery Attack: 192-bit security against forgery attacks.

4. Integral Attack: Secure against integral attacks.

5. State-recovery Attack:

    * Guess-and-Determine Attack: The time complexity of the guess-and-determine attack cannot be lower than 2#256.

* Algebraic Attack: The system of equations, which needs to be solved in algebraic attacks to Rocca-S, cannot be solved with time complexity 2#256.

6. The Linear Bias: Secure against a statistical attack.

The details can be found in the paper [ROCCA-S].

3.2. Other Attacks

While there are many attack vectors for block ciphers, their application to Rocca-S is restrictive, as the attackers can only know partial information about the internal state from the ciphertext blocks. In other words, reversing the round function is impossible in Rocca-S without guessing many secret state blocks. Therefore, only the above potential attack vectors are taken into account. In addition, due to the usage of the constant (Z0,Z1) at the initialization phase, the attack based on the similarity in the four columns of the AES state is also excluded.

3.3. Nonce Reuse

Inadvertent reuse of the same nonce by two invocations of the Rocca-S encryption operation, with the same key, undermines the security of the messages processed with those invocations. A loss of confidentiality ensues because an adversary will be able to reconstruct the bitwise exclusive-or of the two plaintext values.

3.4. Tag Verification Failure

When the tag verification fails during the decryption phase, it is recommended to erase the plaintext and computed tag.

4. IANA Considerations

IANA has assigned value TBD in the AEAD Algorithms registry to AEAD_ROCCA.

5. References

5.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <https://www.rfc-editor.org/info/rfc5116>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

5.2. Informative References

[AEGIS] Wu, H. and B. Preneel, "AEGIS: A fast authenticated encryption algorithm", Selected Areas in Cryptography (SAC 2013) pp.185-201, 2013.

[AES] National Institute of Standards and Technology, "FIPS 197 Advanced Encryption Standard (AES)", 2001, <https://doi.org/10.6028/NIST.FIPS.197-upd1>.

[AES-NI] Gueron, S., "Intel(R) Advanced Encryption Standard (AES) New Instructions Set", 2010, <https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf>.

[CAESAR] "CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness", 2018, <https://competitions.cr.yp.to/caesar.html>.

[DESIGN] Jean, J. and I. Nikolic, "Efficient Design Strategies Based on the AES Round Function", In: Peyrin, T. (eds) Fast Software Encryption. FSE 2016. Lecture Notes in Computer Science, vol 9783, 2016, <https://doi.org/10.1007/978-3-662-52993-5_17>.

[ePrint2024_901] Takeuchi, R., Todo, Y., and T. Iwata, "Practical Committing Attacks against Rocca-S", 2024, <https://eprint.iacr.org/2024/901>.

[KDF] Chen, L., "Recommendation for Key Derivation Using Pseudorandom Functions", NIST Special Publication 800-108, 2022, <https://doi.org/10.6028/NIST.SP.800-108r1-upd1>.

[ROCCA-S] Anand, R., Banik, S., Caforio, A., Fukushima, K., Isobe, T., Kiyomoto, S., Liu, F., Nakano, Y., Sakamoto, K., and N. Takeuchi, "An Ultra-High Throughput AES-Based Authenticated Encryption Scheme for 6G: Design and Implementation", 28th European Symposium on Research in Computer Security, ESORICS 2023, 2024, <https://doi.org/10.1007/978-3-031-50594-2_12>.

[SNOW-V] Ekdahl, P., Johansson, T., Maximov, A., and J. Yang, "A new SNOW stream cipher called SNOW-V", IACR Transactions on Symmetric Cryptology, 2019(3), 1-42, 2019, <https://doi.org/10.13154/tosc.v2019.i3.1-42>.

[SPEC-5G] 3GPP SA3, "Study on the support of 256-bit algorithms for 5G", 2018, <https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3422>.

[TIAOXIN] Nikolic, I., "Tiaoxin-346: VERSION 2.0", CAESAR Competition, 2014, <https://competitions.cr.yp.to/round2/tiaoxinv2.pdf>.

[WP-6G] Latva-aho, M. and K. Leppaenen, "Key drivers and research challenges for 6G ubiquitous wireless intelligence", 2019.

Appendix A. Software Implementation

A.1. Implementation with SIMD Instructions

Figure 4 shows a sample implementation of Rocca-S.

```
#include <memory.h>
#include <immintrin.h>
#include <stdlib.h>
#include <stdint.h>

#define ROCCA_KEY_SIZE       (32)
#define ROCCA_IV_SIZE        (16)
#define ROCCA_MSG_BLOCK_SIZE (32)
#define ROCCA_TAG_SIZE       (32)
#define ROCCA_STATE_NUM      ( 7)

typedef struct ROCCA_CTX {
   uint8_t key[ROCCA_KEY_SIZE/16][16];
   uint8_t state[ROCCA_STATE_NUM][16];
   size_t size_ad;
   size_t size_m;
} rocca_context;
```

```
#define load(m)    _mm_loadu_si128((const __m128i *)(m))
#define store(m,a) _mm_storeu_si128((__m128i *)(m),a)
#define xor(a,b)   _mm_xor_si128(a,b)
#define and(a,b)   _mm_and_si128(a,b)
#define enc(a,k)   _mm_aesenc_si128(a,k)
#define setzero()  _mm_setzero_si128()

#define ENCODE_IN_LITTLE_ENDIAN(bytes, v) \
  bytes[ 0] = ((uint64_t)(v) << (        3)); \
  bytes[ 1] = ((uint64_t)(v) >> (1 * 8 - 3)); \
  bytes[ 2] = ((uint64_t)(v) >> (2 * 8 - 3)); \
  bytes[ 3] = ((uint64_t)(v) >> (3 * 8 - 3)); \
  bytes[ 4] = ((uint64_t)(v) >> (4 * 8 - 3)); \
  bytes[ 5] = ((uint64_t)(v) >> (5 * 8 - 3)); \
  bytes[ 6] = ((uint64_t)(v) >> (6 * 8 - 3)); \
  bytes[ 7] = ((uint64_t)(v) >> (7 * 8 - 3)); \
  bytes[ 8] = ((uint64_t)(v) >> (8 * 8 - 3)); \
  bytes[ 9] = 0; \
  bytes[10] = 0; \
  bytes[11] = 0; \
  bytes[12] = 0; \
  bytes[13] = 0; \
  bytes[14] = 0; \
  bytes[15] = 0;

#define FLOORTO(a,b) ((a) / (b) * (b))

#define S_NUM     ROCCA_STATE_NUM
#define M_NUM     ( 2)
#define INIT_LOOP (16)
#define TAG_LOOP  (16)

#define VARS4UPDATE \
  __m128i k[2], state[S_NUM], stateNew[S_NUM], M[M_NUM];

#define VARS4ENCRYPT \
  VARS4UPDATE \
  __m128i Z[M_NUM], C[M_NUM];

#define COPY_TO_LOCAL(ctx) \
  for(size_t i = 0; i < S_NUM; ++i) \
  { state[i] = load(&((ctx)->state[i][0])); }

#define COPY_FROM_LOCAL(ctx) \
  for(size_t i = 0; i < S_NUM; ++i) \
  { store(&((ctx)->state[i][0]), state[i]); }

#define COPY_TO_LOCAL_IN_TAG(ctx) \
```

```
  COPY_TO_LOCAL(ctx)    for(size_t i = 0; i < 2; ++i) \
  { k[i] = load(&((ctx)->key[i][0])); }

#define COPY_FROM_LOCAL_IN_INIT(ctx) \
  COPY_FROM_LOCAL(ctx)  for(size_t i = 0; i < 2; ++i) \
  { store(&((ctx)->key[i][0]), k[i]); }

#define UPDATE_STATE(X) \
  stateNew[0] = xor(state[6], state[1]); \
  stateNew[1] = enc(state[0],     X[0]); \
  stateNew[2] = enc(state[1], state[0]); \
  stateNew[3] = enc(state[2], state[6]); \
  stateNew[4] = enc(state[3],     X[1]); \
  stateNew[5] = enc(state[4], state[3]); \
  stateNew[6] = enc(state[5], state[4]); \
  for(size_t i = 0; i < S_NUM; ++i) \
  {state[i] = stateNew[i];}

#define INIT_STATE(key, iv) \
  k[0] = load((key) + 16*0); \
  k[1] = load((key) + 16*1); \
  state[0] = k[1]; \
  state[1] = load(iv); \
  state[2] = load(Z0); \
  state[3] = k[0]; \
  state[4] = load(Z1); \
  state[5] = xor(state[1], state[0]); \
  state[6] = setzero(); \
  M[0] = state[2]; \
  M[1] = state[4]; \
  for(size_t i = 0; i < INIT_LOOP; ++i) { \
    UPDATE_STATE(M) \
  } \
  state[0] = xor(state[0], k[0]); \
  state[1] = xor(state[1], k[0]); \
  state[2] = xor(state[2], k[1]); \
  state[3] = xor(state[3], k[0]); \
  state[4] = xor(state[4], k[0]); \
  state[5] = xor(state[5], k[1]); \
  state[6] = xor(state[6], k[1]);


#define MAKE_STRM \
  Z[0] = enc(xor(state[3], state[5]), state[0]); \
  Z[1] = enc(xor(state[4], state[6]), state[2]);

#define MSG_LOAD(mem, reg) \
  reg[0] = load((mem) +  0); \
```

```
  reg[1] = load((mem) + 16);

#define MSG_STORE(mem, reg) \
  store((mem) +  0, reg[0]); \
  store((mem) + 16, reg[1]);

#define XOR_BLOCK(dst, src1, src2) \
  dst[0] = xor(src1[0], src2[0]); \
  dst[1] = xor(src1[1], src2[1]);

#define MASKXOR_BLOCK(dst, src1, src2, mask) \
  dst[0] = and(xor(src1[0], src2[0]), mask[0]); \
  dst[1] = and(xor(src1[1], src2[1]), mask[1]);

#define ADD_AD(input) \
  MSG_LOAD(input, M) \
  UPDATE_STATE(M)

#define ADD_AD_LAST_BLOCK(input, size) \
  uint8_t tmpblk[ROCCA_MSG_BLOCK_SIZE] = {0}; \
  memcpy(tmpblk, input, size); \
  MSG_LOAD(tmpblk, M) \
  UPDATE_STATE(M)

#define ENCRYPT(output, input) \
  MSG_LOAD(input, M) \
  MAKE_STRM \
  XOR_BLOCK(C, M, Z) \
  MSG_STORE(output, C) \
  UPDATE_STATE(M)

#define ENCRYPT_LAST_BLOCK(output, input, size) \
  uint8_t tmpblk[ROCCA_MSG_BLOCK_SIZE] = {0}; \
  memcpy(tmpblk, input, size); \
  MSG_LOAD(tmpblk, M) \
  MAKE_STRM \
  XOR_BLOCK(C, M, Z) \
  MSG_STORE(tmpblk, C) \
  memcpy(output, tmpblk, size); \
  UPDATE_STATE(M)

#define DECRYPT(output, input) \
  MSG_LOAD(input, C) \
  MAKE_STRM \
  XOR_BLOCK(M, C, Z) \
  MSG_STORE(output, M) \
  UPDATE_STATE(M)
```

```
 #define DECRYPT_LAST_BLOCK(output, input, size) \
   uint8_t tmpblk[ROCCA_MSG_BLOCK_SIZE] = {0}; \
   uint8_t tmpmsk[ROCCA_MSG_BLOCK_SIZE] = {0}; \
   __m128i mask[M_NUM]; \
   memcpy(tmpblk, input, size); \
   memset(tmpmsk, 0xFF , size); \
   MSG_LOAD(tmpblk, C   ) \
   MSG_LOAD(tmpmsk, mask) \
   MAKE_STRM \
   MASKXOR_BLOCK(M, C, Z, mask) \
   MSG_STORE(tmpblk, M) \
   memcpy(output, tmpblk, size); \
   UPDATE_STATE(M)

 #define SET_AD_BITLEN_MSG_BITLEN(sizeAD, sizeM) \
   uint8_t bitlenAD[16]; \
   uint8_t bitlenM [16]; \
   ENCODE_IN_LITTLE_ENDIAN(bitlenAD, sizeAD); \
   ENCODE_IN_LITTLE_ENDIAN(bitlenM , sizeM ); \
   M[0] = load(bitlenAD); \
   M[1] = load(bitlenM );

 #define MAKE_TAG(sizeAD, sizeM, tag) \
   SET_AD_BITLEN_MSG_BITLEN(sizeAD, sizeM) \
   for(size_t i = 0; i < TAG_LOOP; ++i) { \
     UPDATE_STATE(M) \
   } \
   __m128i tag128a = setzero(); \
   for(size_t i = 0; i <= 3; ++i) { \
     tag128a = xor(tag128a, state[i]); \
   } \
   __m128i tag128b = setzero(); \
   for(size_t i = 4; i <= 6; ++i) { \
     tag128b = xor(tag128b, state[i]); \
   } \
   store((tag)   , tag128a); \
   store((tag)+16, tag128b);

 static const uint8_t Z0[] = {0xcd,0x65,0xef,0x23,0x91, \
 0x44,0x37,0x71,0x22,0xae,0x28,0xd7,0x98,0x2f,0x8a,0x42};
 static const uint8_t Z1[] = {0xbc,0xdb,0x89,0x81,0xa5, \
 0xdb,0xb5,0xe9,0x2f,0x3b,0x4d,0xec,0xcf,0xfb,0xc0,0xb5};

 void rocca_init(rocca_context * ctx, const uint8_t * key, \
 const uint8_t * iv) {
    VARS4UPDATE
    INIT_STATE(key, iv);
    COPY_FROM_LOCAL_IN_INIT(ctx);
```

```
    ctx->size_ad = 0;
    ctx->size_m  = 0;
 }

 void rocca_add_ad(rocca_context * ctx, const uint8_t * in, size_t size)
 {
    VARS4UPDATE
    COPY_TO_LOCAL(ctx);
    size_t i = 0;
    for(size_t size2 = FLOORTO(size, ROCCA_MSG_BLOCK_SIZE); \
    i < size2; i += ROCCA_MSG_BLOCK_SIZE) {
       ADD_AD(in + i);
    }
    if(i < size) {
       ADD_AD_LAST_BLOCK(in + i, size - i);
    }
    COPY_FROM_LOCAL(ctx);
    ctx->size_ad += size;
 }

 void rocca_encrypt(rocca_context * ctx, uint8_t * out, \
 const uint8_t * in, size_t size) {
    VARS4ENCRYPT
    COPY_TO_LOCAL(ctx);
    size_t i = 0;
    for(size_t size2 = FLOORTO(size, ROCCA_MSG_BLOCK_SIZE); \
    i < size2; i += ROCCA_MSG_BLOCK_SIZE) {
       ENCRYPT(out + i, in + i);
    }
    if(i < size) {
       ENCRYPT_LAST_BLOCK(out + i, in + i, size - i);
    }
    COPY_FROM_LOCAL(ctx);
    ctx->size_m += size;
 }

 void rocca_decrypt(rocca_context * ctx, uint8_t * out, \
 const uint8_t * in, size_t size) {
    VARS4ENCRYPT
    COPY_TO_LOCAL(ctx);
    size_t i = 0;
    for(size_t size2 = FLOORTO(size, ROCCA_MSG_BLOCK_SIZE); \
    i < size2; i += ROCCA_MSG_BLOCK_SIZE) {
       DECRYPT(out + i, in + i);
    }
    if(i < size) {
       DECRYPT_LAST_BLOCK(out + i, in + i, size - i);
    }
```

```
    COPY_FROM_LOCAL(ctx);
    ctx->size_m += size;
 }

 void rocca_tag(rocca_context * ctx, uint8_t *tag) {
    VARS4UPDATE
    COPY_TO_LOCAL_IN_TAG(ctx);
    MAKE_TAG(ctx->size_ad, ctx->size_m, tag);
 }
```

Figure 4: Reference Implementation with SIMD

A.2. Test Vector

This section gives test vectors of Rocca-S. The least significant octet of the vector is shown on the left and the first 128-bit value is shown on the first line.

```
=== test vector #1===
key =
00000000000000000000000000000000
00000000000000000000000000000000
nonce =
00000000000000000000000000000000
associated data =
00000000000000000000000000000000
00000000000000000000000000000000
plaintext =
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
ciphertext =
9ac3326495a8d414fe407f47b5441050
2481cf79cab8c0a669323e07711e4617
0de5b2fbba0fae8de7c1fccaeefc3626
24fcfdc15f8bb3e64457e8b7e37557bb
tag =
8df934d1483710c9410f6a089c4ced97
91901b7e2e661206202db2cc7a24a386

=== test vector #2===
key =
01010101010101010101010101010101
01010101010101010101010101010101
nonce =
01010101010101010101010101010101
associated data =
```

```
   0101010101010101010101010101010101
   0101010101010101010101010101010101
   plaintext =
   0000000000000000000000000000000000
   0000000000000000000000000000000000
   0000000000000000000000000000000000
   0000000000000000000000000000000000
   ciphertext =
   559ecb253bcfe26b483bf00e9c748345
   978ff921036a6c1fdcb712172836504f
   bc64d430a73fc67acd3c3b9c1976d807
   90f48357e7fe0c0682624569d3a658fb
   tag =
   c1fdf39762eca77da8b0f1dae5fff75a
   92fb0adfa7940a28c8cadbbbe8e4ca8d

   === test vector #3===
   key =
   0123456789abcdef0123456789abcdef
   0123456789abcdef0123456789abcdef
   nonce =
   0123456789abcdef0123456789abcdef
   associated data =
   0123456789abcdef0123456789abcdef
   0123456789abcdef0123456789abcdef
   plaintext =
   0000000000000000000000000000000000
   0000000000000000000000000000000000
   0000000000000000000000000000000000
   0000000000000000000000000000000000
   ciphertext =
   b5fc4e2a72b86d1a133c0f0202bdf790
   af14a24b2cdb676e427865e12fcc9d30
   21d18418fc75dc1912dd2cd79a3beeb2
   a98b235de2299b9dda93fd2b5ac8f436
   tag =
   a078e1351ef2420c8e3a93fd31f5b113
   5b15315a5f205534148efbcd63f79f00

   === test vector #4===
   key =
   1111111111111111111111111111111111
   2222222222222222222222222222222222
   nonce =
   4444444444444444444444444444444444
   associated data =

   plaintext =
```

```
   808182838485868788898a8b8c8d8e8f
   909192939495969798999a9b9c9d9e9f
   a0a1a2a3a4a5a6a7
   ciphertext =
   e8c7adcc58302893b253c544f5d8e62d
   8fbd81160c2f4a95123962088d29f106
   422d3f26882fd7b1
   tag =
   f650eba86fb19dc14a3bbe8bbfad9ec5
   b5dd77a4c3f83d2c19ac0393dd47928f

   === test vector #5===
   key =
   1111111111111111111111111111111
   2222222222222222222222222222222
   nonce =
   4444444444444444444444444444444
   associated data =

   plaintext =
   808182838485868788898a8b8c8d8e8f
   909192939495969798999a9b9c9d9e9f
   a0a1a2a3a4a5a6a7a8a9aaabacadaeaf
   ciphertext =
   e8c7adcc58302893b253c544f5d8e62d
   8fbd81160c2f4a95123962088d29f106
   422d3f26882fd7b1fdee5680476e7e6e
   tag =
   49bb0ec78cab2c5f40a535925fa2d827
   52aba9606426537fc774f06fc0f6fc12

   === test vector #6===
   key =
   1111111111111111111111111111111
   2222222222222222222222222222222
   nonce =
   4444444444444444444444444444444
   associated data =

   plaintext =
   808182838485868788898a8b8c8d8e8f
   909192939495969798999a9b9c9d9e9f
   a0a1a2a3a4a5a6a7a8a9aaabacadaeaf
   b0b1b2b3b4b5b6b7b8
   ciphertext =
   e8c7adcc58302893b253c544f5d8e62d
   8fbd81160c2f4a95123962088d29f106
   422d3f26882fd7b1fdee5680476e7e6e
```

```
1fc473cdb2dded85c6
tag =
c674604803963a4b51685fda1f2aa043
934736db2fbab6d188a09f5e0d1c0bf3

=== test vector #7===
key =
1111111111111111111111111111111111
2222222222222222222222222222222222
nonce =
44444444444444444444444444444444
associated data =

plaintext =
808182838485868788898a8b8c8d8e8f
909192939495969798999a9b9c9d9e9f
a0a1a2a3a4a5a6a7a8a9aaabacadaeaf
b0b1b2b3b4b5b6b7b8b9babbbcbdbebf
ciphertext =
e8c7adcc58302893b253c544f5d8e62d
8fbd81160c2f4a95123962088d29f106
422d3f26882fd7b1fdee5680476e7e6e
1fc473cdb2dded85c692344f3ab85af0
tag =
850599a6624a3e936a77768c7717b926
cc519081730df447127654d6980bcb02
```

Acknowledgements

This draft is partially supported by a contract of "Research and development on new generation cryptography for secure wireless communication services" among "Research and Development for Expansion of Radio Wave Resources (JPJ000254)", which was supported by the Ministry of Internal Affairs and Communications, Japan.

We thank Ryunosuke Takeuchi, Yosuke Todo and Tetsu Iwata for their analysis and pointing out the possibility of identical initial states due to the key expansion.

Authors’ Addresses

Yuto Nakano
KDDI Research, Inc.
2-1-15 Ohara, Fujimino-shi, Saitama,
356-8502
Japan
Email: yt-nakano@kddi.com

Kazuhide Fukushima
KDDI Research, Inc.
2-1-15 Ohara, Fujimino-shi, Saitama,
356-8502
Japan
Email: ka-fukushima@kddi.com

Takanori Isobe
The University of Osaka
1-5 Yamadaoka, Suita-shi, Osaka,
565-0871
Japan
Email: takanori.isobe@ist.osaka-u.ac.jp