The Authenticated Cipher MORUS (v1)

15 March, 2014

Designers: Hongjun Wu, Tao Huang

Submitters: Hongjun Wu, Tao Huang

Contact: `wuhongjun@gmail.com`

Division of Mathematical Sciences
Nanyang Technological University, Singapore

Table of Contents

1 Introduction 3
2 Specification of MORUS 3
  2.1 Preliminaries 3
    2.1.1 Operations 3
    2.1.2 Notations and Constants 3
  2.2 Parameters 4
  2.3 Recommended parameter sets 5
  2.4 The state update function of MOURS 5
  2.5 MORUS-640 7
    2.5.1 The initialization of MORUS-640 7
    2.5.2 Processing the associated data 7
    2.5.3 The encryption of MORUS-640 9
    2.5.4 The finalization of MORUS-640 9
    2.5.5 The decryption and verification fo MORUS-640 9
  2.6 MORUS-1280. 10
    2.6.1 The initialization of MORUS-1280 10
    2.6.2 Processing the associated data 10
    2.6.3 The encryption of MORUS-1280 11
    2.6.4 The finalization of MORUS-1280 11
3 Security Goals 11
4 Security Analysis 11
  4.1 The security of the initialization 12
  4.2 The security of the encryption process 13
  4.3 The security of message authentication 13
    4.3.1 Internal state collision 13
    4.3.2 Attacks on the finalization 15
5 Features 16
6 Performance 16
7 Design rationale 17
  7.1 State update function 17
  7.2 Encryption and authentication 17
  7.3 Selection of rotation constants 18
8 Intellectual property 18
9 Consent 18

1 Introduction

In this document, we specify the MORUS family of authenticated ciphers with two different internal state sizes: 640 bits and 1280 bits, and two different key sizes: 128 bits and 256 bits. Three MORUS algorithms – MORUS-640-128, MORUS-1280-128, and MORUS-1280-256 are recommended in this specification.

MORUS is a dedicated authenticated cipher which achieves encryption and authentication simultaneously. The encryption/decryption is done by XORing the plaintext with the keystream. And the authentication/verification is done by injecting the message into state and generating/verifying an authentication tag.

The MORUS algorithms are designed to be very fast in hardware, since only the shifts, AND, and XOR operations are used in the cipher, and the critical path is very short. MORUS is very efficient in software. The speed of MORUS-1280 (on Intel i7 4770 Haswell, 64-bit Ubuntu 13.10 and GCC 4.8.1) is 0.69 cpb for 16384 blocks of message.

This document is organized as follows. The MORUS specification is introduced in Section 2. The security of MORUS is discussed in Section 3 and Section 4. The features of MORUS are discussed Section 5. The performance of MORUS is given in Section 6. The design rationale is given in Section 7.

2 Specification of MORUS

2.1 Preliminaries

2.1.1 Operations

The following operations are used in MORUS:

$\oplus$	: bit-wise exclusive OR.
&	: bit-wise AND.
$\|\|$	: concatenation.
<<<	: rotation to the left.
>>>	: rotation to the right.
$\lceil x \rceil$	: ceiling operation, $\lceil x \rceil$ is the smallest integer not less than x.
$Rotl\_128\_32(x, n)$	: Divide a 128-bit block x into 4 32-bit words, rotate each word left by n bits.
$Rotl\_256\_64(x, n)$	: Divide a 256-bit block x into 4 64-bit words, rotate each word left by n bits.

2.1.2 Notations and Constants

The following notations and constants are used in MORUS:

0^n	: n bits of '0's.
1^n	: n bits of '1's.
AD	: associated data (this data will not be encrypted or decrypted).
AD_i^{128}	: a 16-byte associated data block (the last block may be a partial block).
AD_i^{256}	: a 32-byte associated data block (the last block may be a partial block).
$adlen$	: bit length of the associated data with $0 \le adlen < 2^{64}$.
b_i	: rotation constants used in $Rotl\_xxx\_yy$, $0 \le i \le 4$, which are given in Table 2.
C	: ciphertext.
C_i	: a ciphertext block (the last block may be a partial block).
$const$	: a 32-byte constant in the hexadecimal format; $const = 00 \parallel 01 \parallel 01 \parallel 02 \parallel 03 \parallel 05 \parallel 08 \parallel 0d \parallel 15 \parallel 22 \parallel 37 \parallel 59 \parallel 90 \parallel e9 \parallel 79 \parallel 62 \parallel db \parallel 3d \parallel 18 \parallel 55 \parallel 6d \parallel c2 \parallel 2f \parallel f1 \parallel 20 \parallel 11 \parallel 31 \parallel 42 \parallel 73 \parallel b5 \parallel 28 \parallel dd$. This is the Fibonacci sequence modulo 256.
$const_0$	: the first 16 bytes of $const$.
$cosnt_1$	: the second 16 bytes of $const$.
IV_{128}	: 128-bit initialization vector used in MORUS-640.
K_{128}	: 128-bit key used in MORUS.
K_{256}	: 256-bit key used in MORUS.
$msglen$	: bit length of the plaintext/ciphertext with $0 \le msglen < 2^{64}$.
P	: plaintext.
P_i	: a 16-byte plaintext block (the last block may be a partial block).
S^i	: state at the beginning of ith step.
S_j^i	: state at the beginning of jth round at ith step.
$S_{j,k}^i$	: kth element of the state S_j^i. In MORUS-640, each element is 128-bit; in MORUS-1280, each element is 256-bit.
T	: authentication tag.
t	: bit length of the authentication tag.
u	: number of AD blocks after padding, $u = \lceil \frac{adlen}{128} \rceil$ for MORUS-640; $u = \lceil \frac{adlen}{256} \rceil$ for MORUS-128.
v	: number of plaintext blocks after padding, $v = \lceil \frac{msglen}{128} \rceil$ for MORUS-640; $v = \lceil \frac{msglen}{256} \rceil$ for MORUS-1280.
w_i	: constants used in left rotations, which are given in Table 3.

2.2 Parameters

MORUS is a family of authenticated ciphers with two internal state sizes: 640 and 1280 bits. 128-bit and 256-bit key sizes are supported in MORUS. The associated data length and the plaintext length are less than 2^{64} bits. The authentication tag is less than or equal to 128 bits. We strongly recommend the use of a 128-bit tag. We do not require any secret message number in our design, and the public message number (IV) is 128-bit in MORUS. Table 1 summarizes the parameters.

Parameters	Length in bits		
	MORUS-640-128	MORUS-1280-128	MORUS-1280-256
Plaintext (P)	$< 2^{64}$	$< 2^{64}$	$< 2^{64}$
Associated data (AD)	$< 2^{64}$	$< 2^{64}$	$< 2^{64}$
Key (K)	128	128	256
Tag (T)	128	128	128
Initialization vector (IV)	128	128	128
State (S)	640	1280	1280

Table 1: The MORUS parameters.

2.3 Recommended parameter sets

- Primary recommendation: MORUS-1280-128
  128-bit key, 128-bit nonce, 1280-bit state, 128-bit tag
  Reason: high speed for both software and hardware applications.
- Secondary recommendation: MORUS-640-128
  128-bit key, 128-bit nonce, 640-bit state, 128-bit tag
  Reason: smaller state.
- Tertiary recommendation: MORUS-1280-256
  256-bit key, 128-bit nonce, 1280-bit state, 128-bit tag
  Reason: MORUS-1280-256 uses 256-bit secret key.

2.4 The state update function of MOURS

In each step of MORUS, there are 5 rounds with similar operations to update the state S. Notice that the message block is used in the updates of *Round* 2 to *Round* 5 but not in *Round* 1. The operation `Rotl_xxx_yy` is to divide an xxx-bit state element into 4 words of yy-bit, and perform left rotation operation for every yy-bit word. `Rotl_128_32` is used in MORUS-640 and `Rotl_256_64` is used in MORUS-1280. The rotation constants for each round are defined in Table 2.

	MORUS-640	MORUS-1280
b_0	5	13
b_1	31	46
b_2	7	38
b_3	22	7
b_4	13	4

Table 2: Rotation constants used in `Rotl_xxx_yy` in MORUS

Besides the the `Rotl_xxx_yy` operation, the left rotation of a whole state element is used for diffusion. The rotation constants for the left rotation are list in Table 3.

	MORUS-640	MORUS-1280
w_0	32	64
w_1	64	128
w_2	96	192
w_3	64	128
w_4	32	64

Table 3: Rotation constants used in left rotation in MORUS

$S^{i+1} = \texttt{StateUpdate}(S^i, p_i)$ is given as follows:

$$
\begin{aligned}
\textit{Round } 1: \quad & S^i_{1,0} = \texttt{Rotl\_xxx\_yy}(S^i_{0,0} \oplus (S^i_{0,1} \,\&\, S^i_{0,2}) \oplus S^i_{0,3},\ b_0);\\
& S^i_{1,3} = S^i_{0,3} \lll w_0;\\
& S^i_{1,1} = S^i_{0,1};\\
& S^i_{1,2} = S^i_{0,2};\\
& S^i_{1,4} = S^i_{0,4};\\
\textit{Round } 2: \quad & S^i_{2,1} = \texttt{Rotl\_xxx\_yy}(S^i_{1,1} \oplus (S^i_{1,2} \,\&\, S^i_{1,3}) \oplus S^i_{1,4} \oplus m_i,\ b_1);\\
& S^i_{2,4} = S^i_{1,4} \lll w_1;\\
& S^i_{2,0} = S^i_{1,0};\\
& S^i_{2,2} = S^i_{1,2};\\
& S^i_{2,3} = S^i_{1,3};\\
\textit{Round } 3: \quad & S^i_{3,2} = \texttt{Rotl\_xxx\_yy}(S^i_{2,2} \oplus (S^i_{2,3} \,\&\, S^i_{2,4}) \oplus S^i_{2,0} \oplus m_i,\ b_2);\\
& S^i_{3,0} = S^i_{2,0} \lll w_2;\\
& S^i_{3,1} = S^i_{2,1};\\
& S^i_{3,3} = S^i_{2,3};\\
& S^i_{3,4} = S^i_{2,4};\\
\textit{Round } 4: \quad & S^i_{4,3} = \texttt{Rotl\_xxx\_yy}(S^i_{3,3} \oplus (S^i_{3,4} \,\&\, S^i_{3,0}) \oplus S^i_{3,1} \oplus m_i,\ b_3);\\
& S^i_{4,1} = S^i_{3,1} \lll w_3;\\
& S^i_{4,0} = S^i_{3,0};\\
& S^i_{4,2} = S^i_{3,2};\\
& S^i_{4,4} = S^i_{3,4};
\end{aligned}
$$

$$\begin{aligned}
\textit{Round }5: \quad & S_{0,4}^{i+1} = \texttt{Rotl\_xxx\_yy}(S_{4,4}^{i} \oplus (S_{4,0}^{i} \,\&\, S_{4,1}^{i}) \oplus S_{4,2}^{i} \oplus m_i,\ b_4);\\
& S_{0,2}^{i+1} = S_{4,2}^{i} \lll w_4;\\
& S_{0,0}^{i+1} = S_{4,0}^{i};\\
& S_{0,1}^{i+1} = S_{4,1}^{i};\\
& S_{0,3}^{i+1} = S_{4,3}^{i};
\end{aligned}$$

The state update function is shown in Fig.1

2.5 MORUS-640

MORUS-640 uses 5 128-bit registers in its internal state. In each step, it processes one block of 128-bit associated data or plaintext.

2.5.1 The initialization of MORUS-640

The initialization of MORUS-640 consists of loading the key and IV into the state, and running the cipher for 16 steps. The key and IV are loaded into the state as follows:

$$\begin{aligned}
& S_{0,0}^{-16} = IV_{128};\\
& S_{0,1}^{-16} = K_{128};\\
& S_{0,2}^{-16} = 1^{128};\\
& S_{0,3}^{-16} = const_0;\\
& S_{0,4}^{-16} = const_1;
\end{aligned}$$

After loading the key and IV, the internal state is updated 16 times using the state update function:

$$\text{For } i = -16 \text{ to} -1,\ S^{i+1} = \texttt{StateUpdate}(S^i, 0);$$

Then the key is xored to the state again:

$$S_{0,1}^{0} = S_{0,1}^{0} \oplus K_{128}.$$

2.5.2 Processing the associated data

After the initialization, the associated data AD is processed using the state update function.

1. If the last associated data block is not a full block, use '0' bits to pad it to 128 bits for MORUS-640, and the padded full block is used to update the state. Note that if $adlen = 0$, the state will not be updated.

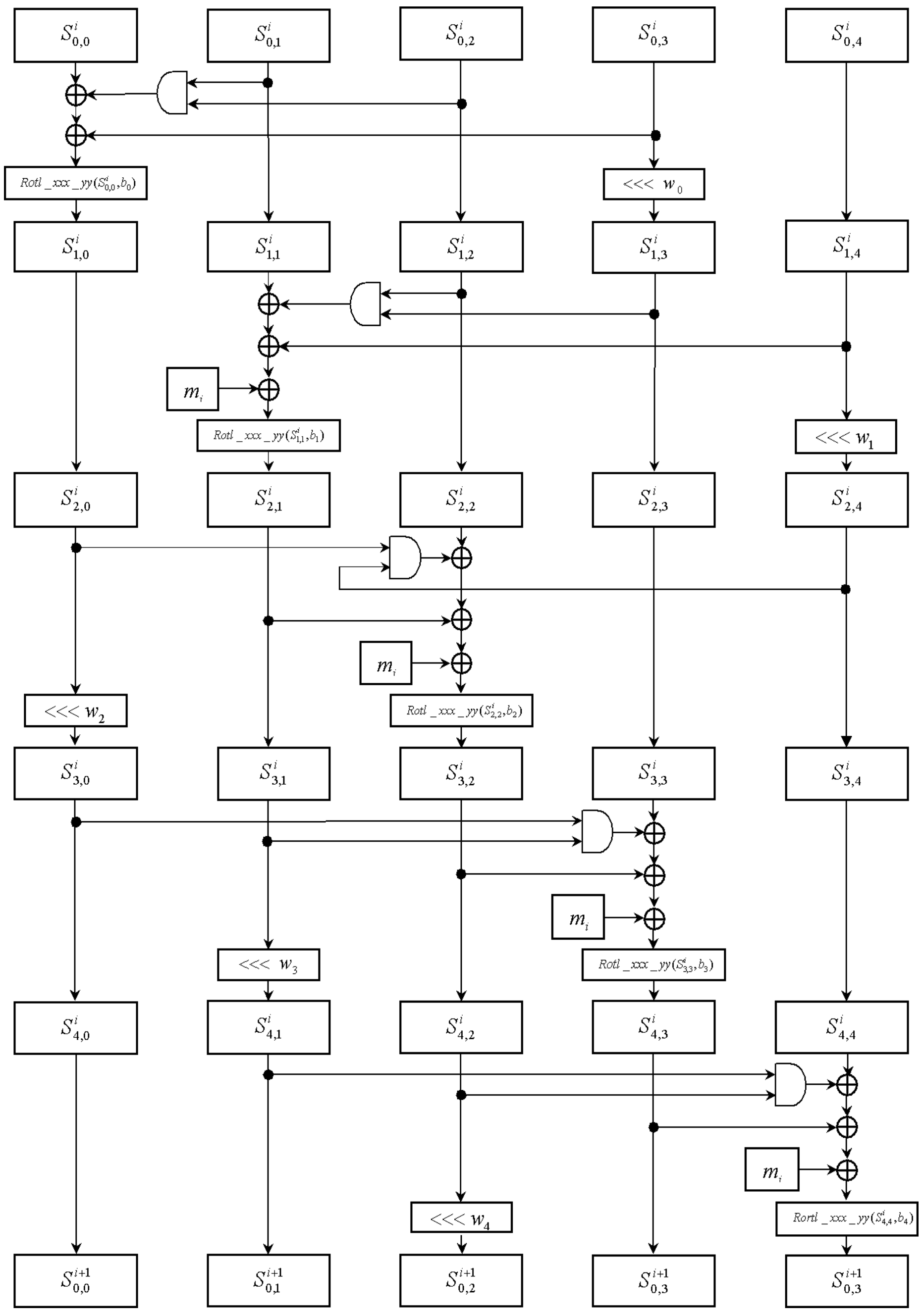

Fig. 1: The state update function of MORUS. In *Rotl_xxx_yy*, *xxx_yy* is 128_32 for MORUS-640 and 256_64 for MORUS-1280.

2. For $i = 0$ to l, we update the state:

$$S^{i+1} = \texttt{StateUpdate}(S^i, AD_i^{128});$$

where $l = \lceil \frac{adlen}{128} \rceil - 1$.

2.5.3 The encryption of MORUS-640

After processing the associated data, at each step of the encryption, a 16-byte plaintext block P_i is used to update the state, and P_i is encrypted to C_i.

1. If the last plaintext block is not a full block, use '0' bits to pad it to 128 bits, and the padded full block is used to update the state. But only the partial block is encrypted. Note that if $msglen = 0$, the state will not get updated, and there is no encryption.
2. Let $u = \lceil \frac{adlen}{128} \rceil$ and $v = \lceil \frac{msglen}{128} \rceil$. For $i = 0$ to $v - 1$, we perform encryption and update state:

$$C_i = P_i \oplus S_0^{u+i} \oplus (S_1^{u+i} \lll 96) \oplus (S_2^{u+i} \& S_3^{u+i});$$
$$S^{u+i+1} = \texttt{StateUpdate}(S^{u+i}, P_i);$$

2.5.4 The finalization of MORUS-640

After encrypting all the plaintext blocks, we generate the authentication tag using eight more steps. The length of the associated data and the length of the message are used to update the state.

1. $tmp = S_3^{u+v} \oplus (adlen \parallel msglen)$, where $adlen$ and $msglen$ are represented as 64-bit integers.
2. $S_4^{u+v} = S_4^{u+v} \oplus S_0^{u+v}$.
3. For $i = u + v$ to $u + v + 7$, we update the state:

$$S^{i+1} = \texttt{StateUpdate}(S^i, tmp);$$

4. We generate the authentication tag from the state S^{u+v+8} as follows:

$$T' = \oplus_{i=1}^{4} S_i^{u+v+8};$$

2.5.5 The decryption and verification fo MORUS-640

The exact values of key size, IV size, and tag size should be known to the decryption and verification processes. The decryption starts with the initialization and the processing of authenticated data. Then the ciphertext is decrypted as follows:

1. If the last ciphertext block is not a full block, decrypt only the partial ciphertext block. The partial plaintext block is padded with 0 bits, and the padded full plaintext block is used to update the state.

2. For $i = 0$ to $v-1$, we perform decryption and update the state.

$$P_i = C_i \oplus S_0^{u+i} \oplus (S_1^{u+i} \lll 96) \oplus (S_2^{u+i} \& S_3^{u+i});$$
$$S^{u+i+1} = \texttt{StateUpdate}(S^{u+i}, P_i);$$

The finalization in the decryption process is the same as that in the encryption process. We emphasize that if the verification fails, the ciphertext and the newly generated authentication tag should not be given as output; otherwise, the state of MORUS-640 is vulnerable to known-plaintext or chosen-ciphertext attacks (using a fixed IV). This requirement also applies to MORUS-1280.

2.6 MORUS-1280.

MORUS-1280 uses 5 256-bit registers in its internal state. In each step, it processes one block of 256-bit associated data or plaintext.

2.6.1 The initialization of MORUS-1280

The initialization of MORUS-1280 consists of loading the key and IV into the state, and running the cipher for 16 steps. Let k_0 set as follows:

- When the key size is 128-bit, $k_0 = K_{128} \parallel K_{128}$.
- When the key size is 256-bit, $k_0 = K_{256}$.

the key and IV are loaded into the state as follows:

$S_{0,0}^{-16} = IV_{128} \parallel 0^{128};$

$S_{0,1}^{-16} = k_0;$

$S_{0,2}^{-16} = 1^{256};$

$S_{0,3}^{-16} = 0^{256};$

$S_{0,4}^{-16} = const_0 \parallel const_1.$

After loading the key and IV, the internal state is updated 16 steps:

For $i = -16$ to $-1, S^{i+1} = \texttt{StateUpdate}(S^i, 0);$

Then the key is XORed with the state again:

$S_{0,1}^0 = S_{0,1}^0 \oplus k_0;$

2.6.2 Processing the associated data

After the initialization, the associated data AD is used to update the state.

1. If the last associated data block is not a full block, use '0' bits to pad it to 256 bits for MORUS-1280, and the padded full block is used to update the state. Note that if $adlen = 0$, the state will not be updated.
2. For $i = 0$ to l, we update the state:

$$S^{i+1} = \texttt{StateUpdate}(S^i, AD_i^{256});$$

where $l = \lceil \frac{adlen}{256} \rceil - 1$.

2.6.3 The encryption of MORUS-1280

After processing the associated data, at each step of the encryption, a 32-byte plaintext block P_i is used to update the state, and P_i is encrypted to C_i.

1. If the last plaintext block is not a full block, use '0' bits to pad it to 256 bits, and the padded full block is used to update the state. But only the partial block is encrypted. Note that if $msglen = 0$, the state will not get updated, and there is no encryption.
2. Let $u = \lceil \frac{adlen}{256} \rceil$ and $v = \lceil \frac{msglen}{256} \rceil$. For $i = 0$ to $v - 1$, we perform encryption and update state:

$$C_i = P_i \oplus S_0^{u+i} \oplus (S_1^{u+i} \lll 192) \oplus (S_2^{u+i} \& S_3^{u+i});$$
$$S^{u+i+1} = \texttt{StateUpdate}(S^{u+i}, P_i);$$

2.6.4 The finalization of MORUS-1280

After encrypting all the plaintext blocks, we generate the authentication tag using eight more steps. The length of the associated data and the length of the message are used to update the state.

1. $tmp = S_3^{u+v} \oplus (adlen \parallel msglen \parallel 0^{128})$, where $adlen$ and $msglen$ are represented as 64-bit integers.
2. $S_4^{u+v} = S_4^{u+v} \oplus S_0^{u+v}$.
3. For $i = u + v$ to $u + v + 7$, we update the state:

$$S^{i+1} = \texttt{StateUpdate}(S^i, tmp);$$

4. We generate the authentication tag from the state S^{u+v+8} as follows:

$$T' = \oplus_{i=1}^{4} S_i^{u+v+8};$$

The authentication tag T consists of the first t bits of T'.

3 Security Goals

The security goals of MORUS are given in Table 4. In MORUS, each key and IV pair should be used to protect only one message. If verification fails, the new tag and the decrypted ciphertext should not be given as output.

Note that the encryption security is under the assumption that the attacker could not forge a message through repeated trials. The integrity security in Table 4 includes the integrity security of plaintext, associated data and nonce and under the assumption that the secret key is unknown to the attacker, and 128-bit tag is used.

4 Security Analysis

In this section, we will give our initial analysis on the security of MORUS. Most of the analysis is on MORUS-640, but can be extended to MORUS-1280 trivially.

Table 4: Security Goals of MORUS.

	Confidentiality (bits)	Integrity (bits)
MORUS-640-128	128	128
MORUS-1280-128	128	128
MORUS-1280-256	256	128

4.1 The security of the initialization

The initialization of MORUS is designed to ensure the IV and key are properly mixed so that the internal state after initialization is secret. We consider the differential attack using IV differences as the main threat to the initialization of MORUS. The reason is that the IV is the only input in the initial state controlled by an attacker under our security assumptions (differences in key are considered uncontrollable by an attacker).

In order to maximize the differential probability, we will model the AND operation in following way:

- Any difference passes through the AND operation will be eliminated unless it is able to cancel a difference in XOR operations to reduce the weight of the updated state element.

For example, suppose that S_A, S_B, S_C, and S_D are four state elements, and $S_A = S_B \oplus (S_C \ \& \ S_D)$ is computed. If there is a difference at bit i of either S_C or S_D, the difference in the output of AND operation for bit i will be the same as the ith bit of S_B to ensure the ith bit of S_A has no difference.

This approximation does not always lead to the minimal number of active bits. But in most of the circumstances we analyzed, especially when the number of active bits is small, it does give a good approximation to the minimal number of active bits.

We discuss the differential probability of the initialization of MORUS-640 with input differences on IV through following two cases.

Case 1. There is only 1 bit difference in IV. Notice that MORUS uses bit operations in the state update function. As a result, the position of the one bit difference will not affect the differential probability. We assume the difference is at bit 0 of the IV in our analysis. Using the above mentioned approximation, we find that after 6 steps, the differential probability is 2^{-201} when the value of initial state is assumed unknown. And after 7 steps, the differential probability is 2^{-357}. Since the initial state is given except the key, the differential probability can be 1 for some bits in the first two steps. However, after two steps, the values of state elements are mixed with the key and thus can be considered unknown. We may exclude the differential probability in first three steps, which is 2^{-10} for a more accurate bound.

Case 2. There are 2 or 3 bits difference in IV. We fix a bit difference at bit 0 of IV, and enumerated all the cases that there are 2 and 3 active bits in the

IV. For 2 bits cases, there are 127 possible ways to choose the position of the other bit. And for 3 bits case, there are $\frac{127\times126}{2} = 8001$ possible ways to choose the positions of the other two bits. After 5 steps, the differential probability is 2^{-401} for for the 2 bits differences on IV and 2^{-432} for 3 bits difference on IV.

Case 3. When the difference on IV is more than 3 bits, we expect that the differential probability will decrease until the weight in the internal state elements is high enough so that a large number of active bits get canceled. But in that case, the weight of active bits will increase fast in the early steps and differential probability is likely to fall below 2^{-256} more quickly than the 1 bit difference case.

Hence, it is unlikely that there is any differential characteristic with probability hight than 2^{-128} after 16 steps of MORUS initialization.

4.2 The security of the encryption process

The MORUS encryption is a stream cipher with a large state which is updated continuously. The attacks against a block cipher cannot be applied directly to MORUS. We emphasize that the security of encryption is under the assumption that the IV is not reused for the same key. Once the IV is reused, it will lead to serious attacks on recovering the state.

Statistical Attacks. If the IV is used only once for each key, it is impossible to apply a differential attack to the encryption process. And in general, the statistical attacks are difficult to launch against MORUS. And non-linear operations are used in the state update and keystream generation, which makes it difficult to recover the state by applying algebraic attacks.

4.3 The security of message authentication

The security of message authentication of MORUS is related to the length of the tag generated. In our analysis, we will only consider the case that the tag length is 128-bit since it implies the security of the cases with shorter tag length. To analyze the authentication of the MORUS, we will compute the probability that a forged message would bypass the verification.

4.3.1 Internal state collision Construction of internal state collision is a typical method used in attacking the message authentication. For MORUS, since the internal state size is at least 640-bit, any internal collision through birthday attack requires about 2^{-320} blocks to be encrypted, which is far too expensive for an attacker. Another method to construct an internal state collision is to inject a message difference at certain step and cancel it at a later step. Our analysis will show that this attack will lead to an internal state collision with probability below 2^{-128}.

First, we consider the case that the difference gets eliminated in two steps, *i.e.,* the difference is injected to the internal state at one step and get eliminated immediately at the next step. There are 10 rounds (2 steps) in this case, which we number as Round 1,..., Round 10. Recall that two state elements get updated in each round: one is to compute a state element using 4 of the previous state elements and the other is to left rotate a state element computed two rounds ago. For simplicity, we will slightly change the order of these operations so that only one state element is updated in each round. And we call this state element as CV_i when it is updated at Round i. What we do is to perform the left rotation immediate after one state element get updated and rotate back when it is used after two rounds. Hence the state update in Round i becomes:

$$CV_i = CV_{i-5} \oplus (CV_{i-4} \mathbin{\&} CV_{i-3}) \oplus (CV_{i-2} \ggg w_{i \bmod 5}) \oplus m_i$$

Notice that m_i is the plaintext block used in each step and $m_i = 0$ if $i = 0 \bmod 5$. And the difference in plaintext will inject to Round 2 and be the same in Round 3-5.

To eliminate the difference after two steps, we need that $CV_6, \ldots, CV_{10}$ have no difference. In our study, we will focus on following two conditions:

1: No difference at CV_6. This is because CV_6 is completely determined by the previous state elements and has nothing to do with the plaintext block in the second step.
2: For each difference at bit i in CV_3 or CV_4 there must be a difference at bit i in CV_5. Otherwise, is impossible to eliminate the difference using the difference in the second plaintext block.

Then we search the input difference bits to find a lower bound for the number of bits with difference (active bits) in the input. And we found that for the input difference with weight less than or equal to 25, there is no valid 10-round differential characteristics for MORUS. Now we may evaluate the bound for the differential probabilities. When input difference is n bits, there are n bits differences at CV_2, CV_3 and CV_5. Since each bit difference will be involved in two AND operations, and each AND operation on one bit has differential probability 2^{-1}, the differential probability is at least 2^{-5n} (5 AND operations for CV_i and CV_{i+1}, $i = 1, 2, 3, 4, 5$). The differential probability is less than $2^{-26} \times 5 = 2^{-130}$.

Next, we consider the case that the input difference get eliminated in 3 steps. If there are 3 active bits in the input, the differential probability after 3 step is 2^{-132} by our approximation. Note that the difference is not eliminated through the approximation. And much stronger conditions are needed to eliminate the differences. Hence the probability that the input difference get eliminated after 3 steps will be much lower than 2^{-132} when the number of active bits is 3. When we increase the number of active bits in the input, the trend is to increase the weight of active bits in the states, which we can observe in the previous cases. Intuitively, this is can be explained as when the weight of active bits is low, the increased number of active bits exceeds the number of active bits get eliminated. And when the weight is high enough such that the cancellation effect is dominated, we can expect the overall weight will be higher than the single difference case in the first 3 steps. Hence, although it is impossible to enumerate all the input differences, we believe that there is no differential characteristic with probability higher than 2^{-128} which can eliminate the input difference in 3 steps.

Now we deal with the cases that the number of active bits in the input is less than three.

- Only one active bit in the input. Since the position of active has no impact on the differentials, we assume the active bit is at bit 0. Then, we propagate the difference up to 3 steps (15 rounds), assuming no input difference at next two steps. Now, we enumerate the input difference at step 2 such that following two conditions are satisfied:

  1. There is no difference at Round 11. Again, it is because the difference cannot be eliminated through the message in step 3.
  2. The active bits at CV_{10} covers the actives bits at CV_8 and CV_9.

  Our search show that even if we increase the number of active bits to 20 in the input of the second step, it is impossible to find a differential characteristic satisfied the above conditions. With similar evaluation of probability, and take consideration to the differential probability introduced by the initial difference, we can conclude that the probability that the internal state collision is less than 2^{-128} in this case.
- Two active bits in the input. By our approximation, the differential probability is at least 2^{-101} for any two active bits propagate to 3 steps. We think it is safe to consider the probability for internal state collision to be less than 2^{-128} if the number of active bits in the second step is larger than 20, in spite that some difference in the internal state may be canceled each other. In our search, we fix one bit difference at bit 0 and try to impose a difference at the other 127 possible positions. And the search result confirms that no valid differential characteristic is found when the number of active bits is less than 21.

Now, consider the rest cases: the difference get eliminated after at least 4 steps. If there is one bit difference at the input, the differential probability is at least 2^{-196} using our approximation, which is much lower than 2^{-128}. And if we want to eliminate the differences, more conditions are required. Hence, it is reasonable to consider the probability to eliminated the internal difference in these cases to be less than 2^{-128}. This conclude our analysis when the internal state collision is constructed through injection of plaintext differences.

4.3.2 Attacks on the finalization In addition to the internal state collision, we analyze the security of the finalization. When there is a difference in the internal state before the finalization, we expect that the differential probability is less than 2^{-128} after 8 steps of the MORUS state update function. Hence, the difference at the tag is not predictable in this case. Another situation is that there is no difference at the internal state but there is some difference at the *adlen* or *msglen*. Then the difference will be used as message difference in the state update function for 8 steps. So the differential probability is expected to be less than 2^{-128} in this case as well.

5 Features

MORUS has the following advantages over previous ciphers:

1. MORUS is efficient in software. According to the previous section, the speed of MORUS-1280 is 0.69 cpb on Intel Haswell processors for long messages, which is around 30% faster than AES-GCM [4].
2. MORUS is fast in hardware performance. In MORUS, the critical path to generate a keystream block is 3 AND gates and 8 XOR gates.
3. MORUS is efficient across platforms. In constructing authenticated encryption schemes, AES is frequently used as a building block. There are authenticated encryption modes so that the AES can be used as underlying block cipher, *e.g.,* EAX [1], CCM [6], GCM [4] and OCB 2.0 [5]. And a number of dedicated AE schemes use AES round function, *e.g.,* AEGIS [7] and ALE [2]. These schemes can be benefited from the AES-NI which performs one round AES encryption/decryption in a single instruction. On the other hand, although the widely use of AES, there are platforms which do not support the AES-NI instruction set. For example, in the ARM architecture, AES-NI instruction set is not implemented so far. The performance of AES based authenticated encryption schemes will be notably slower on these platforms. In contrast, the MORUS family offer a more steady performance across platforms since its performance does not rely on the use of AES-NI instruction set.
4. Secure. MORUS provides 128-bit authentication security, stronger than AES-GCM.

6 Performance

We implemented MORUS in C code. We tested the speed on the Intel Core i7-4770 processor (Haswell) running 64-bit Ubuntu 13.01. Turbo boost is turned off in the experiment. The compiler being used is gcc 4.8.1, and the options "-O3 -mavx2" are used. The test is performed by encrypting/decrypting a message repeatedly, and printing out the final message. To ensure that the tag generation is not removed during the compiler optimization process, we use the tag as the IV for processing the next message. To ensure that the tag verification is not removed during the compiler optimization process, we sum up the number of failed verifications and print out the final result.

Table 5 shows the speed comparison of the MORUS. For long message, the speed of MORUS-640 and MOURS-1280 is about 1.11 cpb and 0.69 cpb, respectively. The speed of MOURS-1280 is faster than that of AES-128-GCM on the Haswell, which is 1.03 cpb [3].

Table 5: The speed comparison (in cycles per byte) for different message length on Intel Haswell. EA means encryption-authentication; DV means decryption-verification.

	16B	64B	512B	1024B	4096B	16384B
MORUS-640(EA)	28	7.72	1.95	1.58	1.18	1.11
MORUS-640(DV)	28	7.99	1.97	1.56	1.23	1.16
MORUS-1280(EA)	33.9	8.28	1.59	1.12	0.78	0.69
MORUS-1280(DV)	35.8	8.46	1.63	1.13	0.80	0.69

7 Design rationale

In our design of MORUS, we are trying to design a fast authenticated cipher which is not based on AES so that this cipher can run fast in platforms with no AES-NI. Our design is aimed at achieving the following goals:

- Simple
- Secure
- Fast in hardware
- Efficient in software
- Avoid using AES round function

7.1 State update function

The construction of state update function of MORUS is based on 5 small round functions with similar operations. In each round function, only XOR, AND and rotations are used. The diffusion of MORUS is from two types of rotations: the rotations on the whole registers (`<<<`) and the rotations on four partial words inside a register (`Rotl_xxx_yy`). The later operation takes advantage of the SSE2 and AVX instructions in which the shifts on four word can be done in one instruction. We choose the AND non-linear function since it can be easily and efficiently implemented in both software and hardware. Two internal state elements get updated in a round function. Hence, every internal state element will get updated twice in a step. It is remarkable that MORUS is constructed using simple bit-wise operations, which makes it fast in hardware implementations.

7.2 Encryption and authentication

The encryption of MORUS adopts the method used in stream ciphers. The key and nonce are mixed into the state during initialization and after that, the cipher generates keystreams and XORs the keystreams with the plaintext to produce ciphertext. In MORUS, message blocks are injected into its state update function so as to authenticate the message simultaneously with the encryption.

In the initialization of MORUS, we use 16 steps of state update function (80 rounds). This is to ensure the state cannot be recovered and the differential probability is small after the initialization.

In the finalization, we introduce an extra XOR operation to distinguish the finalization from the encryption and we use a similar method as used in AEGIS: mixing the length of associated data and plaintext is XORed to one of the internal state elements and used as a message block to update the states for 8 steps. In this way, any change in the internal state or the length of message will be involved in computing the tag.

7.3 Selection of rotation constants

The diffusion in MORUS relies on the 10 rotations. Therefore, the rotation constants need to be carefully chosen. We use following rules in the selection of rotations constants:

1. The rotation constants should exclude the multiples of 8.
2. No rotation constant should be a multiple of another rotation constant.
3. The sum of any two constants modular 32 (or 64 for MORUS-1280) is not equal to 0 or another constant.

We enumerate the possible choices of rotation constants satisfied the above requirements and propagate a 1-bit difference on message to count the weight after four steps for MORUS-640 and five steps for MORUS-1280. Then we select a set of the rotation constants which results in high weight.

The designers have not hidden any weaknesses in this cipher.

8 Intellectual property

MOURS is not patented and it is free of intellectual property restrictions. If any of this information changes, the submitter/submitters will promptly (and within at most one month) announce these changes on the crypto-competitions mailing list.

9 Consent

The submitter/submitters hereby consent to all decisions of the CAESAR selection committee regarding the selection or non-selection of this submission as a second-round candidate, a third-round candidate, a finalist, a member of the final portfolio, or any other designation provided by the committee. The submitter/submitters understand that the committee will not comment on the algorithms, except that for each selected algorithm the committee will simply cite the previously published analyses that led to the selection of the algorithm. The submitter/submitters understand that the selection of some algorithms is not a negative comment regarding other algorithms, and that an excellent algorithm might fail to be selected simply because not enough analysis was available at the time of the committee decision. The submitter/submitters acknowledge that the committee decisions reflect the collective expert judgments of the committee members and are not subject to appeal. The submitter/submitters understand that if they disagree with published analyses then they are expected to promptly and publicly respond to those analyses, not to wait for subsequent committee decisions. The submitter/submitters understand that this statement is required as a condition of consideration of this submission by the CAESAR selection committee.

References

1. M. Bellare, P. Rogaway, and D. Wagner. The EAX mode of operation. In *Fast Software Encryption*, pages 389–407. Springer, 2004.
2. A. Bogdanov, F. Mendel, F. Regazzoni, V. Rijmen, and E. Tischhauser. ALE: AES-Based Lightweight Authenticated Encryption. In *Fast Software Encryption*, 2013.
3. S. Gueron. AES-GCM software performance on the current high end CPUs as a performance baseline for CAESAR. DIAC 2013: Directions in Authenticated Ciphers, Augest 2013.
4. D. McGrew and J. Viega. The Galois/Counter Mode of Operation (GCM). http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/gcm/gcm-spec.pdf.
5. P. Rogaway. Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC. In *Advances in Cryptology–ASIACRYPT 2004*, pages 16–31. Springer, 2004.
6. D. Whiting, R. Housley, and N. Ferguson. Counter with CBC-MAC (CCM). Available from http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/ proposedmodes/ccm/ccm.pdf, 2003.
7. H. Wu and B. Preneel. AEGIS: A Fast Authenticated Encryption Algorithm. Selected Areas in Cryptography – SAC 2013, 2013.