Romulus

v1.0

Designers/Submitters (in alphabetical order):

Tetsu Iwata[1], Mustafa Khairallah[2], Kazuhiko Minematsu[3], Thomas Peyrin[2]

[1] Nagoya University, Japan
tetsu.iwata@nagoya-u.jp

[2] Nanyang Technological University, Singapore
mustafam001@e.ntu.edu.sg,
thomas.peyrin@ntu.edu.sg

[3] NEC Corporation, Japan
k-minematsu@ah.jp.nec.com

Contents

Introduction

This document specifies Romulus, an authenticated encryption with associated data (AEAD) scheme based on a tweakable block cipher (TBC) Skinny. Romulus consists of two families, a nonce-based AE (NAE) Romulus-N and a nonce misuse-resistant AE (MRAE) Romulus-M.

A TBC was introduced by Liskov *et al.* at CRYPTO 2002 [29]. Since its inception, TBCs have been acknowledged as a powerful primitive in that it can be used to construct simple and highly secure NAE/MRAE schemes, including ΘCB3 [26] and SCT [34]. While these schemes are computationally efficient (in terms of the number of primitive calls) and have high security, lightweight applications are not the primal use cases of these schemes, and they are not particularly suitable for small devices. With this in mind, Romulus aims at lightweight, efficient, and highly-secure NAE and MRAE schemes, based on a TBC.

The overall structure of Romulus-N shares similarity in part with a (TBC-based variant of) block cipher mode COFB [13, 15], yet, we make numerous refinements to achieve our design goal. Romulus-N generally requires a fewer number of TBC calls than ΘCB3 thanks to the faster MAC computation for associated data, while the hardware implementation is significantly smaller than ΘCB3 thanks to the reduced state size and inverse-freeness (*i.e.*, TBC inverse is not needed). In fact, Romulus-N's state size is essentially what is needed for computing TBC. Moreover, it encrypts an n-bit plaintext block by just one call of the n-bit block TBC, hence there is no efficiency loss. Romulus-N is extremely efficient for small messages, which is particularly important in many lightweight applications, requiring for example only 2 TBC calls to handle one associated data block and one message block (in comparison, other designs like ΘCB3, OCB3, TAE, CCM require from 3 to 5 TBC to calls in the same situation). Romulus-N achieves these advantages without the security penalty, *i.e.*, Romulus-N has the full n-bit security, which is a similar security bound to ΘCB3.

If we compare Romulus-N with other size-oriented and n-bit secure AE schemes, such as conventional permutation-based AEs using $3n$-bit permutation with n-bit rate, the state size is comparable ($3n$ to $3.5n$ bits). Our advantage is that the underlying cryptographic primitive is expected to be much more lightweight and/or faster because of smaller output size ($3n$ vs n bits). In addition, the n-bit security of Romulus-N is proved under the standard model, which provides a high-level assurance for security not only quantitatively but also qualitatively. To elaborate a bit more, with a security proof in the standard model, one can precisely connect the security status of the primitive to the overall security of the mode that uses this primitive. In our case, for each of the members of Romulus, the best attack on it implies a chosen-plaintext attack (CPA) in the single-key setting against Skinny, *i.e.*, unless Skinny is broken by CPA adversaries in the single-key setting, Romulus indeed maintains the claimed n-bit security. Such a guarantee is not possible with non-standard models and it is often not easy to deduce the impact of a found "flaw" of the primitive to the security of the mode. In a more general context, this gap between the proof and the actual security is best exemplified by "uninstantiable" Random Oracle-Model schemes [5, 11]. To evaluate the security of Romulus, with the standard model proof, we can focus on the security evaluation of Skinny, while this type of focus is not possible in schemes with proofs in non-standard models.

Another interesting feature of Romulus-N is that it can reduce area depending on the use cases, without harming security. If it is enough to have a relatively short nonce or a short counter (or both), which is common to low-power networks, we can save the area by truncating the tweak length. This was possible because Skinny allows to reduce area if a part of its tweak is never used. A member of Romulus-N (Romulus-N2) particularly benefits from this feature. Note that this type of area reduction is not possible with conventional permutation-based AE schemes: it only offers a throughput/security tread-off.

Romulus-M follows the general construction of MRAE called SIV [38]. Romulus-M reuses the components of Romulus-N as much as possible, and Romulus-M is simply obtained by processing message twice by Romulus-N. This allows a faster and smaller operation than TBC-based MRAE SCT, yet, we maintain strong security features of SCT. That is, Romulus-M achieves n-bit security against nonce-respecting adversaries and $n/2$-bit security against nonce-misusing adversaries. Moreover, Romulus-M enjoys a useful feature called graceful degradation introduced at SCT. This ensures that the full n-bit security is almost retained if the number of nonce repetitions at encryption is limited. Thanks to the shared components, most of the advantages of Romulus-N mentioned above also hold for Romulus-M.

We present a detailed comparison of Romulus with other AE candidates in Section 6.

As the underlying TBC, we adopt Skinny proposed at CRYPTO 2016 [2]. The security of this TBC has been extensively studied, and it has attractive implementation characteristics.

Organization of the document. In Section 2, we first introduce the basic notations and the notion of tweakable block cipher, followed by the list of parameters for Romulus, the recommended parameter sets, and the specification of TBC Skinny. In the last part of Section 2, we specify two families of Romulus, Romulus-N and Romulus-M. We present our security claims in Section 3 and show our security analysis including the provable security bounds and the status of computational security of Skinny in Section 4. In Section 5, we describe the desirable features of Romulus. The design rationale under our schemes, including some details of modes and choice of the TBC, is presented in Section 6. Finally, we show some implementation aspects of Romulus in Section 7.

Specification

2.1 Notations

Let $\{0,1\}^*$ be the set of all finite bit strings, including the empty string ε. For $X \in \{0,1\}^*$, let $|X|$ denote its bit length. Here $|\varepsilon| = 0$. For integer $n \geq 0$, let $\{0,1\}^n$ be the set of n-bit strings, and let $\{0,1\}^{\leq n} = \bigcup_{i=0,\ldots,n}\{0,1\}^i$, where $\{0,1\}^0 = \{\varepsilon\}$. Let $[\![n]\!] = \{1,\ldots,n\}$ and $[\![n]\!]_0 = \{0,1,\ldots,n-1\}$.

For two bit strings X and Y, $X \,\|\, Y$ is their concatenation. We also write this as XY if it is clear from the context. Let 0^i (1^i) be the string of i zero bits (i one bits), and for instance we write 10^i for $1 \,\|\, 0^i$. Bitwise XOR of two variables X and Y is denoted by $X \oplus Y$, where $|X| = |Y| = c$ for some positive integer c. We write $\mathtt{msb}_x(X)$ (resp. $\mathtt{lsb}_x(X)$) to denote the truncation of X to its x most (resp. least) significant bits. See "Endian" paragraph below.

Padding. For $X \in \{0,1\}^{\leq l}$ of length multiple of 8 (*i.e.*, byte string), let

$$\mathtt{pad}_l(X) = \begin{cases} X & \text{if } |X| = l, \\ X \,\|\, 0^{l-|X|-8} \,\|\, \mathtt{len}_8(X), & \text{if } 0 \leq |X| < l, \end{cases}$$

where $\mathtt{len}_8(X)$ denotes the one-byte encoding of the byte-length of X. Here, $\mathtt{pad}_l(\varepsilon) = 0^l$. When $l = 128$, $\mathtt{len}_8(X)$ has 16 variations (*i.e.*, byte length 0 to 15), and we encode it to the last 4 bits of $\mathtt{len}_8(X)$ (for example, $\mathtt{len}_8(11) = \mathtt{00001011}$). The case $l = 64$ is similarly treated, by using the last 3 bits.

Parsing. For $X \in \{0,1\}^*$, let $|X|_n = \max\{1, \lceil |X|/n \rceil\}$. Let $(X[1],\ldots,X[x]) \xleftarrow{n} X$ be the parsing of X into n-bit blocks. Here $X[1] \,\|\, X[2] \,\|\, \ldots \,\|\, X[x] = X$ and $x = |X|_n$. When $X = \varepsilon$, we have $X[1] \xleftarrow{n} X$ and $X[1] = \varepsilon$. Note in particular that $|\varepsilon|_n = 1$.

Alternating Parsing. Let n and t be positive integers larger than 8. For $X \in \{0,1\}^*$, let $(X[1],\ldots,X[x]) \xleftarrow{n,t} X$ be the parsing of X into n-bit blocks and t-bit blocks in an alternating order. That is, we have $X[1] \,\|\, X[2] \,\|\, \ldots \,\|\, X[x] = X$, where $|X[i]| = n$ for any odd $i \in \{1,\ldots,x-1\}$, $|X[i]| = t$ for any even $i \in \{1,\ldots,x-1\}$, $|X[x]| \in [\![n]\!]$ if x is odd, and $|X[x]| \in [\![t]\!]$ if x is even. When $X \neq \varepsilon$, x is determined as

$$x = \begin{cases} 2\lfloor |X|/(n+t) \rfloor & \text{if } |X| > 0 \text{ and } |X| \bmod (n+t) = 0 \\ 2\lfloor |X|/(n+t) \rfloor + 1 & \text{if } 1 \leq |X| \bmod (n+t) \leq n \\ 2\lfloor |X|/(n+t) \rfloor + 2 & \text{if } n < |X| \bmod (n+t) < n+t. \end{cases}$$

When $X = \varepsilon$, $X[1] \xleftarrow{n,t} X$ (thus $x = 1$) and $X[1] = \varepsilon$.

Galois Field. An element a in the Galois field $\mathrm{GF}(2^n)$ will be interchangeably represented as an n-bit string $a_{n-1} \dots a_1 a_0$, a formal polynomial $a_{n-1}\mathtt{x}^{n-1} + \cdots + a_1\mathtt{x} + a_0$, or an integer $\sum_{i=0}^{n-1} a_i 2^i$.

Matrix. Let G be an $n \times n$ binary matrix defined over $\mathrm{GF}(2)$. For $X \in \{0,1\}^n$, let $G(X)$ denote the matrix-vector multiplication over $\mathrm{GF}(2)$, where X is interpreted as a column vector. We may write $G \cdot X$ instead of $G(X)$.

Endian. We employ little endian for byte ordering: an n-bit string X is received as

$$X_7 X_6 \dots X_0 \,\|\, X_{15} X_{14} \dots X_8 \,\|\, \dots \,\|\, X_{n-1} X_{n-2} \dots X_{n-8},$$

where X_i denotes the $(i+1)$-st bit of X (for $i \in [\![n]\!]_0$). Therefore, when c is a multiple of 8 and X is a byte string, $\mathtt{msb}_c(X)$ and $\mathtt{lsb}_c(X)$ denote the last (rightmost) c bytes of X and the first (leftmost) c bytes of X, respectively. For example, $\mathtt{lsb}_{16}(X) = (X_7 X_6 \dots X_0 \,\|\, X_{15} X_{14} \dots X_8)$ and $\mathtt{msb}_8(X) = (X_{n-1} X_{n-2} \dots X_{n-8})$ with the above X. Since our specification is defined over byte strings, we only consider the above case for $\mathtt{msb}$ and $\mathtt{lsb}$ functions (*i.e.*, the subscript c is always a multiple of 8).

(Tweakable) Block Cipher. A tweakable block cipher (TBC) is a keyed function $\widetilde{E} : \mathcal{K} \times \mathcal{T_W} \times \mathcal{M} \to \mathcal{M}$, where $\mathcal{K}$ is the key space, $\mathcal{T_W}$ is the tweak space, and $\mathcal{M} = \{0,1\}^n$ is the message space, such that for any $(K, T_w) \in \mathcal{K} \times \mathcal{T_W}$, $\widetilde{E}(K, T_w, \cdot)$ is a permutation over $\mathcal{M}$. We interchangeably write $\widetilde{E}(K, T_w, M)$ or $\widetilde{E}_K(T_w, M)$ or $\widetilde{E}_K^{T_w}(M)$. When $\mathcal{T_W}$ is singleton, it is essentially a block cipher and is simply written as $E : \mathcal{K} \times \mathcal{M} \to \mathcal{M}$.

2.2 Parameters

Romulus has the following parameters:

- Nonce length $nl \in \{96, 128\}$.
- Key length $k = 128$.
- Message block length $n = 128$.
- Counter bit length $d \in \{24, 56, 48\}$.
- AD block length $n + t$, where $t \in \{96, 128\}$.
- Tag length $\tau = 128$.
- A TBC $\widetilde{E} : \mathcal{K} \times \overline{\mathcal{T}} \times \mathcal{M} \to \mathcal{M}$, where $\mathcal{K} = \{0,1\}^k$, $\mathcal{M} = \{0,1\}^n$, and $\overline{\mathcal{T}} = \mathcal{T} \times \mathcal{B} \times \mathcal{D}$. Here, $\mathcal{T} = \{0,1\}^t$, $\mathcal{D} = [\![2^d - 1]\!]_0$, and $\mathcal{B} = [\![256]\!]_0$ for parameters t and d, and $\mathcal{B}$ is also represented as a byte (see Section 2.5.1). For tweak $\overline{T} = (T, B, D) \in \overline{\mathcal{T}}$, T is always assumed to be a byte string including ε, and t is a multiple of 8. $\widetilde{E}$ is either Skinny-128-384 or Skinny-128-256 with appropriate tweakey encoding functions as described in Section 2.4.

For $\widetilde{E}$, $\mathcal{T}$ is used to process the nonce or an AD block, $\mathcal{D}$ is used for counter, and $\mathcal{B}$ is for domain separation, *i.e.*, deriving a small number of independent instances.

While our submission fixes $\tau = 128$, a tag for NAE schemes can be truncated if needed, at the cost of decreased security against forgery. See Section 4.

Table 2.1: Members of Romulus.

Family	Name	$\widetilde{E}$	k	nl	n	t	d	τ
Romulus-N	Romulus-N1	Skinny-128-384	128	128	128	128	56	128
	Romulus-N2	Skinny-128-384	128	96	128	96	48	128
	Romulus-N3	Skinny-128-256	128	96	128	96	24	128
Romulus-M	Romulus-M1	Skinny-128-384	128	128	128	128	56	128
	Romulus-M2	Skinny-128-384	128	96	128	96	48	128
	Romulus-M3	Skinny-128-256	128	96	128	96	24	128

NAE and MRAE families. Romulus has two families, Romulus-N and Romulus-M, and each family consists of several members (the sets of parameters). The former implements nonce-based AE (NAE) secure against Nonce-respecting adversaries, and the latter implements nonce Misuse-resistant AE (MRAE) introduced by Rogaway and Shrimpton [38]. The name Romulus stands for the set of two families.

2.3 Recommended Parameter Sets

We present our members (sets of parameters) in Table 2.1. The primary member of our submission is Romulus-N1. Members except Romulus-N3 and Romulus-M3 conform to the requirements for primary member with respect to key length (minimum 128 bits), nonce length (minimum 96 bits), tag length (minimum 64 bits), and maximum input length (minimum $2^{50} - 1$ bytes). Romulus-N3 and Romulus-M3 conform to the requirements of primary member except that they have a 24-bit counter, in return to better efficiency. See Section 6 for our justification of length limits.

2.4 The Tweakable Block Cipher Skinny

In this section, we will recall the Skinny family of tweakable block ciphers [2]. In our submission, we will use two members of the family: Skinny-128-384 and Skinny-128-256.

Skinny Versions.

The lightweight block ciphers of the Skinny family have 64-bit and 128-bit block versions. However, we will only use the $n = 128$ bits versions here. The internal state is viewed as a 4×4 square array of cells, where each cell is a byte. We denote $IS_{i,j}$ the cell of the internal state located at Row i and Column j (counting starting from 0). One can also view this 4×4 square array of cells as a vector of cells by concatenating the rows. Thus, we denote with a single subscript IS_i the cell of the internal state located at Position i in this vector (counting starting from 0) and we have that $IS_{i,j} = IS_{4\cdot i+j}$.

Skinny follows the TWEAKEY framework from [23] and thus takes a tweakey input instead of a key or a pair key/tweak. The family of lightweight block ciphers Skinny have three main tweakey size versions, but we will use only two of them: for a block size n, we will use versions with tweakey size $t = 2n$ and $t = 3n$. We denote $z = t/n$ the tweakey size to block size ratio. The tweakey state is also viewed as a collection of z 4×4 square arrays of cells. We denote these arrays $TK1$ and $TK2$ when $z = 2$, and $TK1$, $TK2$ and $TK3$ when $z = 3$. Moreover, we denote $TKz_{i,j}$ the cell of the tweakey state located at Row i and Column j of the z-th cell array. As for the internal state, we extend this notation to a vector view with a single subscript: $TK1_i$, $TK2_i$ and $TK3_i$. Moreover, we define the adversarial model **SK** (resp. **TK1**, **TK2** or **TK3**) where the attacker cannot (resp. can) introduce differences in the tweakey state.

Initialization.

The cipher receives a plaintext $m = m_0\|m_1\|\cdots\|m_{14}\|m_{15}$, where the m_i are bytes. The initialization of the cipher's internal state is performed by simply setting $IS_i = m_i$ for $0 \leq i \leq 15$:

$$IS = \begin{bmatrix} m_0 & m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 & m_7 \\ m_8 & m_9 & m_{10} & m_{11} \\ m_{12} & m_{13} & m_{14} & m_{15} \end{bmatrix}$$

This is the initial value of the cipher internal state and note that the state is loaded row-wise rather than in the column-wise fashion we have come to expect from the AES; this is a more hardware-friendly choice, as pointed out in [33].

The cipher receives a tweakey input $tk = tk_0\|tk_1\|\cdots\|tk_{30}\|tk_{16z-1}$, where the tk_i are 8-bit cells. The initialization of the cipher's tweakey state is performed by simply setting for $0 \leq i \leq 15$: $TK1_i = tk_i$ and $TK2_i = tk_{16+i}$ when $z = 2$, and finally $TK1_i = tk_i$, $TK2_i = tk_{16+i}$ and $TK3_i = tk_{32+i}$ when $z = 3$. We note that the tweakey states are loaded row-wise.

The Round Function.

Skinny-128-256 has 48 rounds and Skinny-128-384 has 56 rounds. One encryption round is composed of five operations in the following order: `SubCells`, `AddConstants`, `AddRoundTweakey`, `ShiftRows` and `MixColumns` (see illustration in Figure 2.1). Note that no whitening key is used in Skinny.

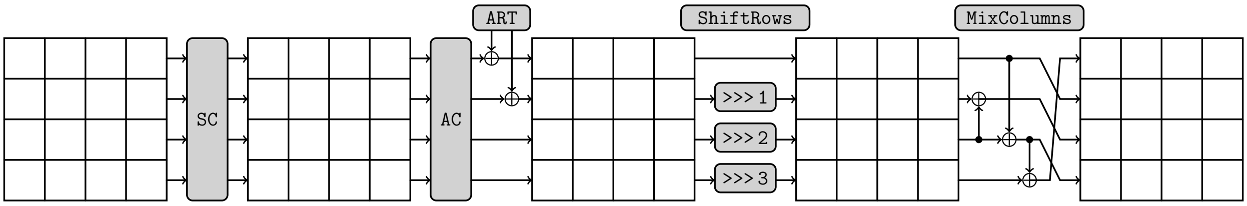


Figure 2.1: The Skinny round function applies five different transformations: `SubCells` (SC), `AddConstants` (AC), `AddRoundTweakey` (ART), `ShiftRows` (SR) and `MixColumns` (MC).

`SubCells`. An 8-bit Sbox is applied to every cell of the cipher internal state. The action of this Sbox is given in hexadecimal notation by the following Table 2.2.

Note that $\mathcal{S}_8$ can also be described with eight NOR and eight XOR operations, as depicted in Figure 2.2. If x_0, ..., x_7 represent the eight inputs bits of the Sbox (x_0 being the least significant bit), it basically applies the below transformation on the 8-bit state:

$$(x_7, x_6, x_5, x_4, x_3, x_2, x_1, x_0) \rightarrow (x_7, x_6, x_5, x_4 \oplus (\overline{x_7 \vee x_6}), x_3, x_2, x_1, x_0 \oplus (\overline{x_3 \vee x_2})),$$

followed by the bit permutation:

$$(x_7, x_6, x_5, x_4, x_3, x_2, x_1, x_0) \longrightarrow (x_2, x_1, x_7, x_6, x_4, x_0, x_3, x_5),$$

repeating this process four times, except for the last iteration where there is just a bit swap between x_1 and x_2.

Table 2.2: 8-bit Sbox $\mathcal{S}_8$ used in Skinny when $s = 8$.

```
uint8_t S8[256] = {
 0x65,0x4c,0x6a,0x42,0x4b,0x63,0x43,0x6b,0x55,0x75,0x5a,0x7a,0x53,0x73,0x5b,0x7b,
 0x35,0x8c,0x3a,0x81,0x89,0x33,0x80,0x3b,0x95,0x25,0x98,0x2a,0x90,0x23,0x99,0x2b,
 0xe5,0xcc,0xe8,0xc1,0xc9,0xe0,0xc0,0xe9,0xd5,0xf5,0xd8,0xf8,0xd0,0xf0,0xd9,0xf9,
 0xa5,0x1c,0xa8,0x12,0x1b,0xa0,0x13,0xa9,0x05,0xb5,0x0a,0xb8,0x03,0xb0,0x0b,0xb9,
 0x32,0x88,0x3c,0x85,0x8d,0x34,0x84,0x3d,0x91,0x22,0x9c,0x2c,0x94,0x24,0x9d,0x2d,
 0x62,0x4a,0x6c,0x45,0x4d,0x64,0x44,0x6d,0x52,0x72,0x5c,0x7c,0x54,0x74,0x5d,0x7d,
 0xa1,0x1a,0xac,0x15,0x1d,0xa4,0x14,0xad,0x02,0xb1,0x0c,0xbc,0x04,0xb4,0x0d,0xbd,
 0xe1,0xc8,0xec,0xc5,0xcd,0xe4,0xc4,0xed,0xd1,0xf1,0xdc,0xfc,0xd4,0xf4,0xdd,0xfd,
 0x36,0x8e,0x38,0x82,0x8b,0x30,0x83,0x39,0x96,0x26,0x9a,0x28,0x93,0x20,0x9b,0x29,
 0x66,0x4e,0x68,0x41,0x49,0x60,0x40,0x69,0x56,0x76,0x58,0x78,0x50,0x70,0x59,0x79,
 0xa6,0x1e,0xaa,0x11,0x19,0xa3,0x10,0xab,0x06,0xb6,0x08,0xba,0x00,0xb3,0x09,0xbb,
 0xe6,0xce,0xea,0xc2,0xcb,0xe3,0xc3,0xeb,0xd6,0xf6,0xda,0xfa,0xd3,0xf3,0xdb,0xfb,
 0x31,0x8a,0x3e,0x86,0x8f,0x37,0x87,0x3f,0x92,0x21,0x9e,0x2e,0x97,0x27,0x9f,0x2f,
 0x61,0x48,0x6e,0x46,0x4f,0x67,0x47,0x6f,0x51,0x71,0x5e,0x7e,0x57,0x77,0x5f,0x7f,
 0xa2,0x18,0xae,0x16,0x1f,0xa7,0x17,0xaf,0x01,0xb2,0x0e,0xbe,0x07,0xb7,0x0f,0xbf,
 0xe2,0xca,0xee,0xc6,0xcf,0xe7,0xc7,0xef,0xd2,0xf2,0xde,0xfe,0xd7,0xf7,0xdf,0xff
};
```

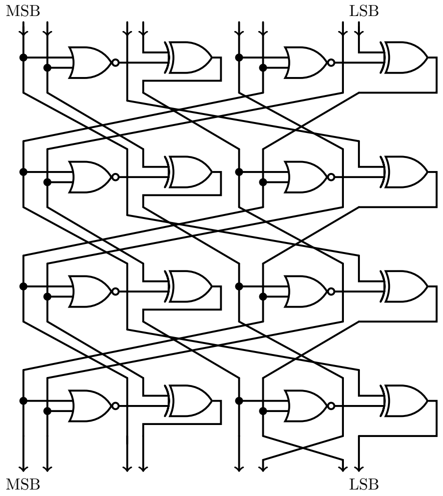


Figure 2.2: Construction of the Sbox $\mathcal{S}_8$.

`AddConstants`. A 6-bit affine LFSR, whose state is denoted $(\mathrm{rc}_5, \mathrm{rc}_4, \mathrm{rc}_3, \mathrm{rc}_2, \mathrm{rc}_1, \mathrm{rc}_0)$ (with rc_0 being the least significant bit), is used to generate round constants. Its update function is defined as:

$$(\mathrm{rc}_5||\mathrm{rc}_4||\mathrm{rc}_3||\mathrm{rc}_2||\mathrm{rc}_1||\mathrm{rc}_0) \rightarrow (\mathrm{rc}_4||\mathrm{rc}_3||\mathrm{rc}_2||\mathrm{rc}_1||\mathrm{rc}_0||\mathrm{rc}_5 \oplus \mathrm{rc}_4 \oplus 1).$$

The six bits are initialized to zero, and updated *before* use in a given round. The bits from the LFSR are arranged into a 4×4 array (only the first column of the state is affected by the LFSR bits), depending on the size of internal state:

$$\begin{bmatrix} c_0 & 0 & 0 & 0 \\ c_1 & 0 & 0 & 0 \\ c_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix},$$

with $c_2 = \texttt{0x2}$ and

$$(c_0, c_1) = (\mathrm{rc}_3 \| \mathrm{rc}_2 \| \mathrm{rc}_1 \| \mathrm{rc}_0,\ 0 \| 0 \| \mathrm{rc}_5 \| \mathrm{rc}_4) \text{ when } s = 4$$
$$(c_0, c_1) = (0 \| 0 \| 0 \| 0 \| \mathrm{rc}_3 \| \mathrm{rc}_2 \| \mathrm{rc}_1 \| \mathrm{rc}_0,\ 0 \| 0 \| 0 \| 0 \| 0 \| 0 \| \mathrm{rc}_5 \| \mathrm{rc}_4) \text{ when } s = 8.$$

The round constants are combined with the state, respecting array positioning, using bitwise exclusive-or. The values of the $(\mathrm{rc}_5, \mathrm{rc}_4, \mathrm{rc}_3, \mathrm{rc}_2, \mathrm{rc}_1, \mathrm{rc}_0)$ constants for each round are given in the table below, encoded to byte values for each round, with rc_0 being the least significant bit.

Rounds	Constants
1 - 16	01,03,07,0F,1F,3E,3D,3B,37,2F,1E,3C,39,33,27,0E
17 - 32	1D,3A,35,2B,16,2C,18,30,21,02,05,0B,17,2E,1C,38
33 - 48	31,23,06,0D,1B,36,2D,1A,34,29,12,24,08,11,22,04
49 - 62	09,13,26,0C,19,32,25,0A,15,2A,14,28,10,20

`AddRoundTweakey`. The first and second rows of all tweakey arrays are extracted and bitwise exclusive-ored to the cipher internal state, respecting the array positioning. More formally, for $i = \{0, 1\}$ and $j = \{0, 1, 2, 3\}$, we have:

- $IS_{i,j} = IS_{i,j} \oplus TK1_{i,j} \oplus TK2_{i,j}$ when $z = 2$,
- $IS_{i,j} = IS_{i,j} \oplus TK1_{i,j} \oplus TK2_{i,j} \oplus TK3_{i,j}$ when $z = 3$.

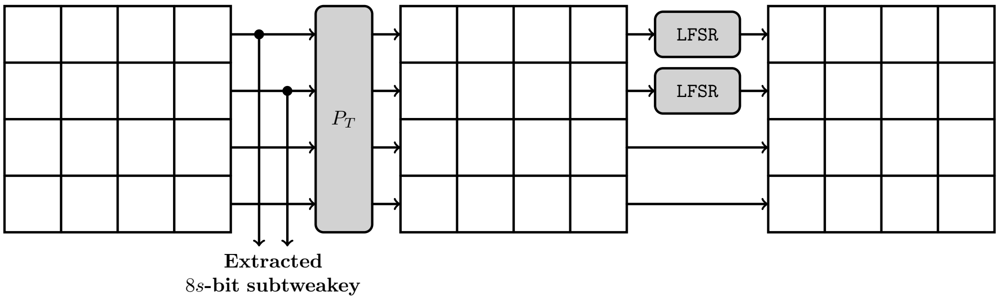


Figure 2.3: The tweakey schedule in Skinny. Each tweakey word $TK1$, $TK2$ and $TK3$ (if any) follows a similar transformation update, except that no LFSR is applied to $TK1$.

Then, the tweakey arrays are updated as follows (this tweakey schedule is illustrated in Figure 2.3). First, a permutation P_T is applied on the cells positions of all tweakey arrays: for all $0 \leq i \leq 15$, we set $TK1_i \leftarrow TK1_{P_T[i]}$ with

$$P_T = [9, 15, 8, 13, 10, 14, 12, 11, 0, 1, 2, 3, 4, 5, 6, 7],$$

and similarly for $TK2$ when $z = 2$, and for $TK2$ and $TK3$ when $z = 3$. This corresponds to the following reordering of the matrix cells, where indices are taken row-wise:

$$(0, \ldots, 15) \xmapsto{P_T} (9, 15, 8, 13, 10, 14, 12, 11, 0, 1, 2, 3, 4, 5, 6, 7)$$

Finally, every cell of the first and second rows of $TK2$ and $TK3$ (for the Skinny versions where $TK2$ and $TK3$ are used) are individually updated with an LFSR. The LFSRs used are given in Table 2.3 (x_0 stands for the LSB of the cell).

Table 2.3: The LFSRs used in Skinny to generate the round constants. The TK parameter gives the number of tweakey words in the cipher.

TK	s	**LFSR**
$TK2$	8	$(x_7\|\|x_6\|\|x_5\|\|x_4\|\|x_3\|\|x_2\|\|x_1\|\|x_0) \rightarrow (x_6\|\|x_5\|\|x_4\|\|x_3\|\|x_2\|\|x_1\|\|x_0\|\|x_7 \oplus x_5)$
$TK3$	8	$(x_7\|\|x_6\|\|x_5\|\|x_4\|\|x_3\|\|x_2\|\|x_1\|\|x_0) \rightarrow (x_0 \oplus x_6\|\|x_7\|\|x_6\|\|x_5\|\|x_4\|\|x_3\|\|x_2\|\|x_1)$

`ShiftRows`. As in AES, in this layer the rows of the cipher state cell array are rotated, but they are to the right. More precisely, the second, third, and fourth cell rows are rotated by 1, 2 and 3 positions to the right, respectively. In other words, a permutation P is applied on the cells positions of the cipher internal state cell array: for all $0 \leq i \leq 15$, we set $IS_i \leftarrow IS_{P[i]}$ with

$$P = [0, 1, 2, 3, 7, 4, 5, 6, 10, 11, 8, 9, 13, 14, 15, 12].$$

`MixColumns`. Each column of the cipher internal state array is multiplied by the following binary matrix $\mathbf{M}$:

$$\mathbf{M} = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}.$$

The final value of the internal state array provides the ciphertext with cells being unpacked in the same way as the packing during initialization. Test vectors for Skinny-128-256 or Skinny-128-384 are provided below.

```
/* Skinny-128-256 */
Key:        009cec81605d4ac1d2ae9e3085d7a1f3
            1ac123ebfc00fddcf01046ceeddfcab3
Plaintext:  3a0c47767a26a68dd382a695e7022e25
Ciphertext: b731d98a4bde147a7ed4a6f16b9b587f

/* Skinny-128-384 */
Key:        df889548cfc7ea52d296339301797449
            ab588a34a47f1ab2dfe9c8293fbea9a5
            ab1afac2611012cd8cef952618c3ebe8
Plaintext:  a3994b66ad85a3459f44e92b08f550cb
Ciphertext: 94ecf589e2017c601b38c6346a10dcfa
```

2.5 The Authenticated Encryption Romulus

2.5.1 The Tweakey Encoding

Domain separation. We will use a domain separation byte B to ensure appropriate independence between the tweakable block cipher calls and the various versions of Romulus. Let $B = (b_7\|b_6\|b_5\|b_4\|b_3\|b_2\|b_1\|b_0)$ be the bitwise representation of this byte, where b_7 is the MSB and b_0 is the LSB (see also Figure 2.4). Then, we have the following:

- $b_7b_6b_5$ will specify the parameter sets. They are fixed to:
  - 000 for Romulus-N1
  - 001 for Romulus-M1
  - 010 for Romulus-N2
  - 011 for Romulus-M2
  - 100 for Romulus-N3
  - 101 for Romulus-M3

  Note that all nonce-respecting modes have $b_5 = 0$ and all nonce-misuse resistant modes have $b_5 = 1$.
- b_4 is set to 1 once we have handled the last block of data (AD and message chains are treated separately), to 0 otherwise.
- b_3 is set to 1 when we are performing the authentication phase of the operating mode (*i.e.*, when no ciphertext data is produced), to 0 otherwise. In the special case where $b_5 = 1$ and $b_4 = 1$ (*i.e.*, last block for the nonce-misuse mode), b_3 will instead denote if the number of message blocks is even ($b_5 = 1$ if that is the case, 0 otherwise).
- b_2 is set to 1 when we are handling a message block, to 0 otherwise. Note that in the case of the misuse-resistant modes, the message blocks will be used during authentication phase (in which case we will have $b_3 = 1$ and $b_2 = 1$). In the special case where $b_5 = 1$ and $b_4 = 1$ (*i.e.*, last block for the nonce-misuse mode), b_3 will instead denote if the number of message blocks is even ($b_5 = 1$ if that is the case, 0 otherwise).
- b_1 is set to 1 when we are handling a padded AD block, to 0 otherwise.
- b_0 is set to 1 when we are handling a padded message block, to 0 otherwise.

The reader can refer to Table 8.1 in the Appendix to obtain the exact specifications of the domain separation values depending on the various cases.

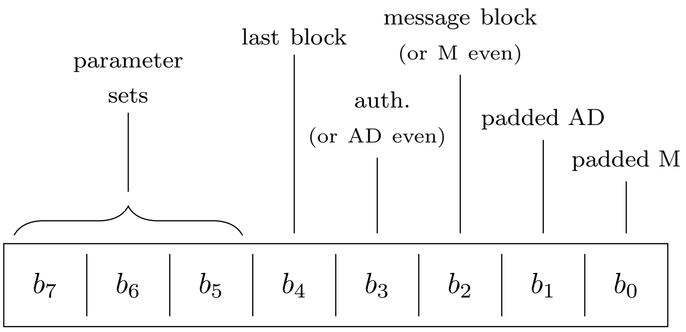


Figure 2.4: Domain separation when using the tweakable block cipher

LFSR. We use LFSRs for counter. For positive integer c, $\mathtt{lfsr}_c$ is a one-to-one mapping $\mathtt{lfsr}_c : [\![2^c - 1]\!]_0 \rightarrow \{0,1\}^c \setminus \{0^c\}$ defined as follows. For positive integer c, let $F_c(\mathtt{x})$ be the lexicographically-first polynomial among the the irreducible degree c polynomials of a minimum number of coefficients. Specifically $F_c(\mathtt{x})$ for $c \in \{56, 24\}$ are

$$F_{56}(\mathtt{x}) = \mathtt{x}^{56} + \mathtt{x}^7 + \mathtt{x}^4 + \mathtt{x}^2 + 1,$$
$$F_{24}(\mathtt{x}) = \mathtt{x}^{24} + \mathtt{x}^4 + \mathtt{x}^3 + \mathtt{x} + 1,$$

and

$$\mathtt{lfsr}_c(D) = 2^D \bmod F_c(\mathtt{x}).$$

Note that we use $\mathtt{lfsr}_c(D)$ as a block counter, so most of the time D changes incrementally with a step of 1, and this enables $\mathtt{lfsr}_c(D)$ to generate a sequence of $2^c - 1$ pairwise-distinct values. From an implementation point of view, it should be implemented in the sequence form, $\mathtt{x}_{i+1} = 2 \cdot \mathtt{x}_i \bmod F_c(\mathtt{x})$.

Let $(z_{c-1} \,\|\, z_{c-2} \,\|\, \ldots \,\|\, z_1 \,\|\, z_0)$ denote the state of c-bit LFSR. In our modes, these LFSRs are initialized to 1 mod $F_c(\mathtt{x})$, *i.e.*, $(0^7 1 \,\|\, 0^{c-8})$, in little-endian format. Incrementation of LFSRs is defined as follows: for $c = 56$,

$$\begin{aligned}
&z_i \leftarrow z_{i-1} \text{ for } i \in [\![56]\!]_0 \setminus \{7, 4, 2, 0\},\\
&z_7 \leftarrow z_6 \oplus z_{55},\\
&z_4 \leftarrow z_3 \oplus z_{55},\\
&z_2 \leftarrow z_1 \oplus z_{55},\\
&z_0 \leftarrow z_{55}.
\end{aligned}$$

Similarly for $c = 24$,

$$\begin{aligned}
&z_i \leftarrow z_{i-1} \text{ for } i \in [\![24]\!]_0 \setminus \{4, 3, 1, 0\},\\
&z_4 \leftarrow z_3 \oplus z_{23},\\
&z_3 \leftarrow z_2 \oplus z_{23},\\
&z_1 \leftarrow z_0 \oplus z_{23},\\
&z_0 \leftarrow z_{23}.
\end{aligned}$$

Our LFSRs are also called *doubling* over GF(2^c) in the context of modes [36].

Tweakey Encoding. We specify the following tweakey encoding functions for implementing TBC $\widetilde{E} : \mathcal{K} \times \overline{\mathcal{T}} \times \mathcal{M} \to \mathcal{M}$ using Skinny-128-256 or Skinny-128-384. The tweakey encoding is a function

$$\mathtt{encode}_{m,t} : \mathcal{K} \times \overline{\mathcal{T}} \to \mathcal{K}_{\mathcal{T}},$$

where $\mathcal{K}_{\mathcal{T}} = \{0,1\}^m$ is the tweakey space for either Skinny-128-256 with $m = 256$ or Skinny-128-384 with $m = 384$. As defined earlier, $\overline{\mathcal{T}} = \mathcal{T} \times \mathcal{B} \times \mathcal{D}$, $\mathcal{K} = \{0,1\}^k$ and $\mathcal{T} = \{0,1\}^t$, $\mathcal{D} = [\![2^d - 1]\!]_0$, $\mathcal{B} = [\![256]\!]_0$.

- Case $(m,t) = (384, 128)$: this variant is used for Romulus-N1 and Romulus-M1. The encode function is defined as follows:

$$\mathtt{encode}_{384,128}(K, T, B, D) = \mathtt{lfsr}_{56}(D) \,\|\, B \,\|\, 0^{64} \,\|\, T \,\|\, K$$

- Case $(m,t) = (384, 96)$: this variant is used for Romulus-N2 and Romulus-M2. The encode function is defined as follows:

$$\mathtt{encode}_{384,96}(K, T, B, D) = \mathtt{lfsr}_{24}(D_1) \,\|\, B \,\|\, T \,\|\, K \,\|\, \mathtt{lfsr}_{24}(D_2) \,\|\, 0^{104},$$

  where $D_1, D_2 \in \mathcal{D}_s$ with $\mathcal{D}_s = [\![2^{24} - 1]\!]$. The set $\mathcal{D}$ is defined as $\mathcal{D} = \mathcal{D}_s \times \mathcal{D}_s$, and the components are determined from D as $D_1 = (D/(2^{24} - 1)) + 1$ and $D_2 = (D \bmod (2^{24} - 1)) + 1$. For the first $2^{24} - 1$ cycles starting from $D = 0$ (but note that $D = 1$ is the initial value in our scheme and $D = 0$ is not used), D_1 is fixed to 1 and D_2 takes all integers of $[\![2^{24} - 1]\!]$. For the next $2^{24} - 1$ cycles, D_1 is fixed to 2 and D_2 takes all values of $[\![2^{24} - 1]\!]$ again, and so on. We stress that the counter cycle is $(2^{24} - 1)^2$ which is slightly smaller than the original range of D. One can also interpret D as a two-dimensional vector: for example, if $D_1 = 0^{23}1$ and $D_2 = 0^{20}1^4$, then $D = (0^{23}1 \parallel 0^{20}1^4)$. In this case, the initial value of D is $[0^7 10^{16}, 0^7 10^{16}]$, in little-endian format.

- Case $(m, t) = (256, 96)$: this variant is used for Romulus-N3 and Romulus-M3. The `encode` function is defined as follows:

$$\texttt{encode}_{256,96}(K, T, B, D) = \texttt{lfsr}_{24}(D) \parallel B \parallel T \parallel K$$

For plaintext $M \in \{0,1\}^n$ and tweak $\overline{\mathcal{T}} = (T, B, D) \in \mathcal{T} \times \mathcal{B} \times \mathcal{D}$, $\widetilde{E}_K^{(T,B,D)}(M)$ denotes encryption of M with m-bit tweakey state $\texttt{encode}_{m,t}(K, T, B, D)$. Tweakey encode is always implicitly applied, hence the counter D is never arithmetic in the tweakey state. To avoid confusion, we may write $\overline{D}$ (in particular when it appears in a part of tweak) in order to emphasize that this is indeed an LFSR counter. One can interpret $\overline{D}$ as a state of LFSR when clocked D times (but in that case it is a part of tweakey state and *not* a part of input of `encode`).

2.5.2 State Update Function

Let G be an $n \times n$ binary matrix defined as an $n/8 \times n/8$ diagonal matrix of 8×8 binary sub-matrices:

$$G = \begin{pmatrix} G_s & 0 & 0 & \dots & 0 \\ 0 & G_s & 0 & \dots & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & \dots & 0 & G_s & 0 \\ 0 & \dots & 0 & 0 & G_s \end{pmatrix},$$

where 0 here represents the 8×8 zero matrix, and G_s is an 8×8 binary matrix, defined as

$$G_s = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

Alternatively, let $X \in \{0,1\}^n$, where n is a multiple of 8, then the matrix-vector multiplication $G \cdot X$ can be represented as

$$G \cdot X = (G_s \cdot X[0], G_s \cdot X[1], G_s \cdot X[2], \dots, G_s \cdot X[n/8 - 1]),$$

where

$$G_s \cdot X[i] = (X[i][1], X[i][2], X[i][3], X[i][4], X[i][5], X[i][6], X[i][7], X[i][7] \oplus X[i][0])$$

for all $i \in [\![n/8]\!]_0$, such that $(X[0], \dots, X[n/8-1]) \xleftarrow{8} X$ and $(X[i][0], \dots, X[i][7]) \xleftarrow{1} X[i]$, for all $i \in [\![n/8]\!]_0$.

The state update function $\rho : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n \times \{0,1\}^n$ and its inverse $\rho^{-1} : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n \times \{0,1\}^n$ are defined as

$$\rho(S, M) = (S', C),$$

where $C = M \oplus G(S)$ and $S' = S \oplus M$. Similarly,

$$\rho^{-1}(S, C) = (S', M),$$

where $M = C \oplus G(S)$ and $S' = S \oplus M$. We note that we abuse the notation by writing ρ^{-1} as this function is only the invert of ρ according to its second parameter. For any $(S, M) \in \{0,1\}^n \times \{0,1\}^n$, if $\rho(S, M) = (S', C)$ holds then $\rho^{-1}(S, C) = (S', M)$. Besides, we remark that $\rho(S, 0^n) = (S, G(S))$ holds.

2.5.3 Romulus-N nonce-based AE mode

The specification of Romulus-N is shown in Figure 2.5. Figure 2.6 shows the encryption of Romulus-N. For completeness, the definition of ρ is also included. Note that the algorithm always assumes $t = nl$.

2.5.4 Romulus-M misuse-resistant AE mode

The specification of Romulus-M is shown in Figure 2.7. Figure 2.8 shows the encryption of Romulus-M. For completeness, the definition of ρ is also included. Note that the algorithm always assumes $t = nl$.

```
Algorithm Romulus-N.Enc$_K(N, A, M)$
    $S \leftarrow 0^n$
    $(A[1], \ldots, A[a]) \xleftarrow{n,t} A$
    if $a \bmod 2 = 0$ then $u \leftarrow t$ else $n$
    if $|A[a]| < u$ then $w_A \leftarrow 26$ else $24$
    $A[a] \leftarrow \mathtt{pad}_u(A[a])$
    for $i = 1$ to $\lfloor a/2 \rfloor$
        $(S, \eta) \leftarrow \rho(S, A[2i-1])$
        $S \leftarrow \widetilde{E}_K^{(A[2i],8,\overline{2i-1})}(S)$
    end for
    if $a \bmod 2 = 0$ then $V \leftarrow 0^n$ else $A[a]$
    $(S, \eta) \leftarrow \rho(S, V)$
    $S \leftarrow \widetilde{E}_K^{(N,w_A,\overline{a})}(S)$
    $(M[1], \ldots, M[m]) \xleftarrow{n} M$
    if $|M[m]| < n$ then $w_M \leftarrow 21$ else $20$
    for $i = 1$ to $m - 1$
        $(S, C[i]) \leftarrow \rho(S, M[i])$
        $S \leftarrow \widetilde{E}_K^{(N,4,\bar{i})}(S)$
    end for
    $M'[m] \leftarrow \mathtt{pad}_n(M[m])$
    $(S, C'[m]) \leftarrow \rho(S, M'[m])$
    $C[m] \leftarrow \mathtt{lsb}_{|M[m]|}(C'[m])$
    $S \leftarrow \widetilde{E}_K^{(N,w_M,\overline{m})}(S)$
    $(\eta, T) \leftarrow \rho(S, 0^n)$
    $C \leftarrow C[1] \,\|\, \ldots \,\|\, C[m-1] \,\|\, C[m]$
    return $(C, T)$
```

```
Algorithm Romulus-N.Dec$_K(N, A, C, T)$
    $S \leftarrow 0^n$
    $(A[1], \ldots, A[a]) \xleftarrow{n,t} A$
    if $a \bmod 2 = 0$ then $u \leftarrow t$ else $n$
    if $|A[a]| < u$ then $w_A \leftarrow 26$ else $24$
    $A[a] \leftarrow \mathtt{pad}_u(A[a])$
    for $i = 1$ to $\lfloor a/2 \rfloor$
        $(S, \eta) \leftarrow \rho(S, A[2i-1])$
        $S \leftarrow \widetilde{E}_K^{(A[2i],8,\overline{2i-1})}(S)$
    end for
    if $a \bmod 2 = 0$ then $V \leftarrow 0^n$ else $A[a]$
    $(S, \eta) \leftarrow \rho(S, V)$
    $S \leftarrow \widetilde{E}_K^{(N,w_A,\overline{a})}(S)$
    $(C[1], \ldots, C[m]) \xleftarrow{n} C$
    if $|C[m]| < n$ then $w_C \leftarrow 21$ else $20$
    for $i = 1$ to $m - 1$
        $(S, M[i]) \leftarrow \rho^{-1}(S, C[i])$
        $S \leftarrow \widetilde{E}_K^{(N,4,\bar{i})}(S)$
    end for
    $\widetilde{S} \leftarrow (0^{|C[m]|} \,\|\, \mathtt{msb}_{n-|C[m]|}(G(S)))$
    $C'[m] \leftarrow \mathtt{pad}_n(C[m]) \oplus \widetilde{S}$
    $(S, M'[m]) \leftarrow \rho^{-1}(S, C'[m])$
    $M[m] \leftarrow \mathtt{lsb}_{|C[m]|}(M'[m])$
    $S \leftarrow \widetilde{E}_K^{(N,w_C,\overline{m})}(S)$
    $(\eta, T^*) \leftarrow \rho(S, 0^n)$
    $M \leftarrow M[1] \,\|\, \ldots \,\|\, M[m-1] \,\|\, M[m]$
    if $T^* = T$ then return $M$ else $\bot$
```

```
Algorithm $\rho(S, M)$
    $C \leftarrow M \oplus G(S)$
    $S' \leftarrow S \oplus M$
    return $(S', C)$
```

```
Algorithm $\rho^{-1}(S, C)$
    $M \leftarrow C \oplus G(S)$
    $S' \leftarrow S \oplus M$
    return $(S', M)$
```

Figure 2.5: The Romulus-N nonce-based AE mode. Lines of [**if (statement) then** $X \leftarrow x$ **else** x'] are shorthand for [**if (statement) then** $X \leftarrow x$ **else** $X \leftarrow x'$]. The dummy variable η is always discarded. We use Romulus-N1 as working example. For other Romulus-N members, the values of the bits b_7 and b_6 in the domain separation need to be adapted accordingly.

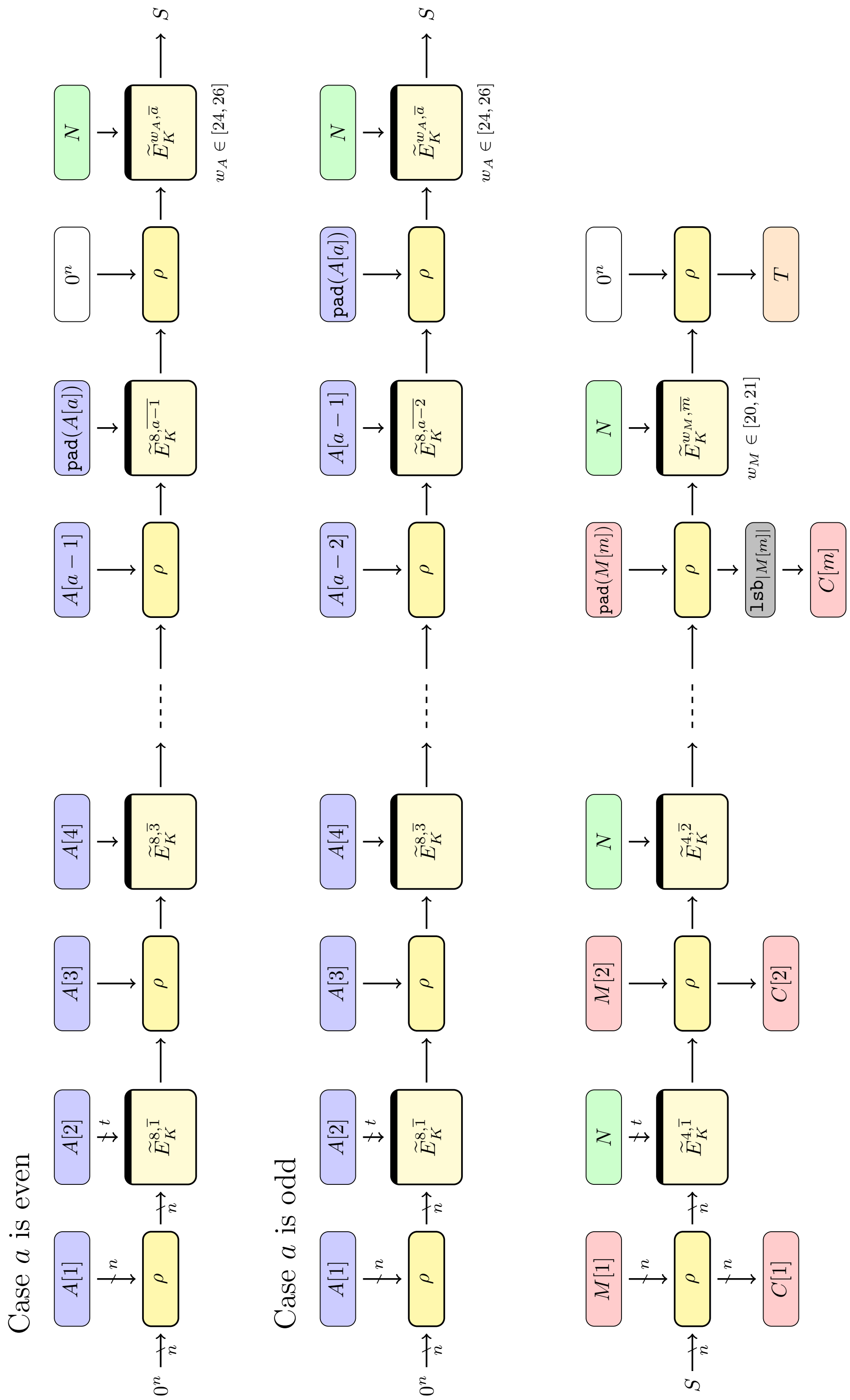

Case a is even
0^n
n
$A[1]$
ρ
$A[2]$
t
$\widetilde{E}_K^{8,\overline{1}}$
$A[3]$
$A[4]$
$\widetilde{E}_K^{8,\overline{3}}$
$A[a-1]$
$\mathsf{pad}(A[a])$
$\widetilde{E}_K^{8,\overline{a-1}}$
0^n
N
$\widetilde{E}_K^{w_A,\overline{a}}$
$w_A \in [24,26]$
S
Case a is odd
0^n
$A[1]$
$A[2]$
$\widetilde{E}_K^{8,\overline{1}}$
$A[3]$
$A[4]$
$\widetilde{E}_K^{8,\overline{3}}$
$A[a-2]$
$A[a-1]$
$\widetilde{E}_K^{8,\overline{a-2}}$
$\mathsf{pad}(A[a])$
N
$\widetilde{E}_K^{w_A,\overline{a}}$
$w_A \in [24,26]$
S
S
$M[1]$
$C[1]$
N
$\widetilde{E}_K^{4,\overline{1}}$
$M[2]$
$C[2]$
N
$\widetilde{E}_K^{4,\overline{2}}$
$\mathsf{pad}(M[m])$
$\mathsf{lsb}_{|M[m]|}$
$C[m]$
N
$\widetilde{E}_K^{w_M,\overline{m}}$
$w_M \in [20,21]$
0^n
T

Figure 2.6: The Romulus-N nonce-based AE mode. (Top) process of AD with even AD blocks (Middle) process of AD with odd AD blocks (Bottom) Encryption. We use Romulus-N1 as working example. For other Romulus-N members, the values of the bits b_7 and b_6 in the domain separation need to be adapted accordingly.

```
Algorithm Romulus-M.Enc_K(N, A, M)
  $S \leftarrow 0^n$
  $(X[1],\ldots,X[a]) \xleftarrow{n,t} A$
  if $a \bmod 2 = 0$ then $u \leftarrow t$ else $n$
  $(X[a+1],\ldots,X[a+m]) \xleftarrow{n+t-u,u} M$
  if $m \bmod 2 = 0$ then $v \leftarrow u$ else $n+t-u$
  $w \leftarrow 48$
  if $|X[a]| < u$ then $w \leftarrow w \oplus 2$
  if $|X[a+m]| < v$ then $w \leftarrow w \oplus 1$
  if $a \bmod 2 = 0$ then $w \leftarrow w \oplus 8$
  if $m \bmod 2 = 0$ then $w \leftarrow w \oplus 4$
  $X[a] \leftarrow \mathtt{pad}_u(X[a])$
  $X[a+m] \leftarrow \mathtt{pad}_v(X[a+m])$
  $x \leftarrow 40$
  for $i = 1$ to $\lfloor (a+m)/2 \rfloor$
    $(S,\eta) \leftarrow \rho(S, X[2i-1])$
    if $i = \lfloor a/2 \rfloor + 1$ then $x \leftarrow x \oplus 4$
    $S \leftarrow \widetilde{E}_K^{(X[2i],x,\overline{2i-1})}(S)$
  end for
  if $a \bmod 2 = m \bmod 2$ then
    $(S,\eta) \leftarrow \rho(S, 0^n)$
  else
    $(S,\eta) \leftarrow \rho(S, X[a+m])$
  $S \leftarrow \widetilde{E}_K^{(N,w,\overline{a+m})}(S)$
  $(\eta, T) \leftarrow \rho(S, 0^n)$
  if $M = \epsilon$ then return $(\epsilon, T)$
  $S \leftarrow T$
  $(M[1],\ldots,M[m']) \xleftarrow{n} M$
  $z \leftarrow |M[m']|$
  $M[m'] \leftarrow \mathtt{pad}_n(M[m'])$
  for $i = 1$ to $m'$
    $S \leftarrow \widetilde{E}_K^{(N,36,\overline{i-1})}(S)$
    $(S, C[i]) \leftarrow \rho(S, M[i])$
  end for
  $C[m'] \leftarrow \mathtt{lsb}_z(C[m'])$
  $C \leftarrow C[1] \,\|\, \ldots \,\|\, C[m'-1] \,\|\, C[m']$
  return $(C, T)$
```

```
Algorithm Romulus-M.Dec_K(N, A, C, T)
  if $C = \epsilon$ then $M \leftarrow \epsilon$
  else
    $S \leftarrow T$
    $(C[1],\ldots,C[m']) \xleftarrow{n} C$
    $z \leftarrow |C[m']|$
    $C[m'] \leftarrow \mathtt{pad}_n(C[m'])$
    for $i = 1$ to $m'$
      $S \leftarrow \widetilde{E}_K^{(N,36,\overline{i-1})}(S)$
      $(S, M[i]) \leftarrow \rho^{-1}(S, C[i])$
    end for
    $M[m'] \leftarrow \mathtt{lsb}_z(M[m'])$
    $M \leftarrow M[1] \,\|\, \ldots \,\|\, M[m'-1] \,\|\, M[m']$
  $S \leftarrow 0^n$
  $(X[1],\ldots,X[a]) \xleftarrow{n,t} A$
  if $a \bmod 2 = 0$ then $u \leftarrow t$ else $n$
  $(X[a+1],\ldots,X[a+m]) \xleftarrow{n+t-u,u} M$
  if $m \bmod 2 = 0$ then $v \leftarrow u$ else $n+t-u$
  $w \leftarrow 48$
  if $|X[a]| < u$ then $w \leftarrow w \oplus 2$
  if $|X[a+m]| < v$ then $w \leftarrow w \oplus 1$
  if $a \bmod 2 = 0$ then $w \leftarrow w \oplus 8$
  if $m \bmod 2 = 0$ then $w \leftarrow w \oplus 4$
  $X[a] \leftarrow \mathtt{pad}_u(X[a])$
  $X[a+m] \leftarrow \mathtt{pad}_v(X[a+m])$
  $x \leftarrow 40$
  for $i = 1$ to $\lfloor (a+m)/2 \rfloor$
    $(S,\eta) \leftarrow \rho(S, X[2i-1])$
    if $i = \lfloor a/2 \rfloor + 1$ then $x \leftarrow x \oplus 4$
    $S \leftarrow \widetilde{E}_K^{(X[2i],x,\overline{2i-1})}(S)$
  end for
  if $a \bmod 2 = m \bmod 2$ then
    $(S,\eta) \leftarrow \rho(S, 0^n)$
  else
    $(S,\eta) \leftarrow \rho(S, X[a+m])$
  $S \leftarrow \widetilde{E}_K^{(N,w,\overline{a+m})}(S)$
  $(\eta, T) \leftarrow \rho(S, 0^n)$
  if $T^* = T$ then return $M$ else $\perp$
```

```
Algorithm $\rho(S, M)$
  $C \leftarrow M \oplus G(S)$
  $S' \leftarrow S \oplus M$
  return $(S', C)$
```

```
Algorithm $\rho^{-1}(S, C)$
  $M \leftarrow C \oplus G(S)$
  $S' \leftarrow S \oplus M$
  return $(S', M)$
```

Figure 2.7: The Romulus-M misuse-resistant AE mode. Lines of [**if (statement) then** $X \leftarrow x$ **else** x'] are shorthand for [**if (statement) then** $X \leftarrow x$ **else** $X \leftarrow x'$]. The dummy variable η is always discarded. We use Romulus-M1 as working example. For other Romulus-M members, the values of the bits b_7 and b_6 in the domain separation need to be adapted accordingly. Note that in the case of empty message, no encryption call has to be performed in the encryption part.

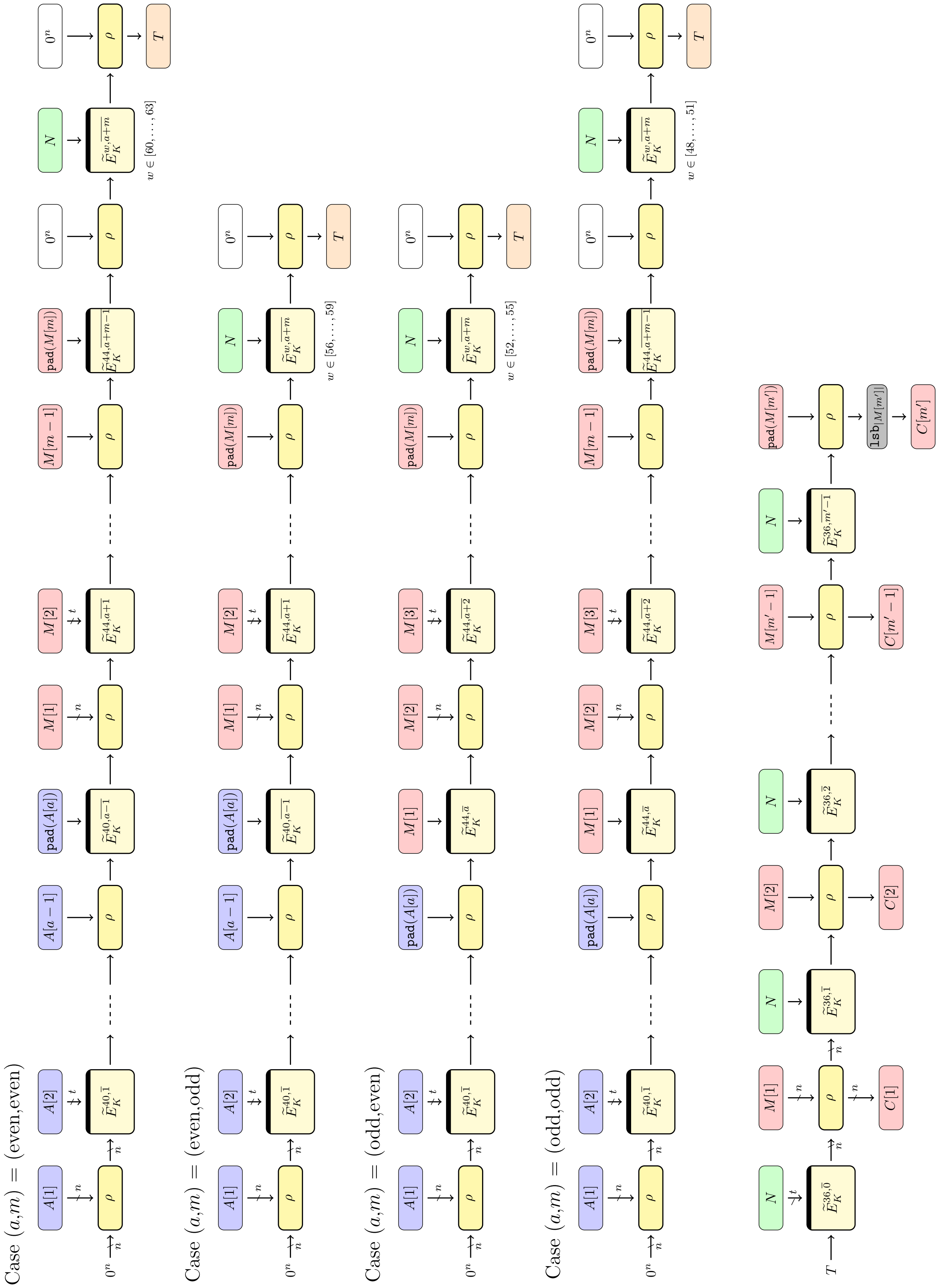

Case $(a,m) = $ (even,even)
Case $(a,m) = $ (even,odd)
Case $(a,m) = $ (odd,even)
Case $(a,m) = $ (odd,odd)
0^n
$A[1]$
$A[2]$
$A[a-1]$
pad$(A[a])$
$M[1]$
$M[2]$
$M[3]$
$M[m-1]$
pad$(M[m])$
N
ρ
T
n
t
$\widetilde{E}_K^{40,\bar{1}}$
$\widetilde{E}_K^{40,\overline{a-1}}$
$\widetilde{E}_K^{44,\bar{a}}$
$\widetilde{E}_K^{44,\overline{a+1}}$
$\widetilde{E}_K^{44,\overline{a+2}}$
$\widetilde{E}_K^{44,\overline{a+m-1}}$
$\widetilde{E}_K^{w,\overline{a+m}}$
$w \in [60,\ldots,63]$
$w \in [56,\ldots,59]$
$w \in [52,\ldots,55]$
$w \in [48,\ldots,51]$
$\widetilde{E}_K^{36,\bar{0}}$
$\widetilde{E}_K^{36,\bar{1}}$
$\widetilde{E}_K^{36,\bar{2}}$
$\widetilde{E}_K^{36,\overline{m'-1}}$
$M[m'-1]$
pad$(M[m'])$
$C[1]$
$C[2]$
$C[m'-1]$
$C[m']$
$\mathsf{lsb}_{|M[m']|}$

Figure 2.8: The Romulus-M misuse-resistant AE mode. (Top) process of AD with even/even, even/odd, odd/even, odd/odd AD and M blocks respectively (Bottom) Encryption. We use Romulus-M1 as working example. For other Romulus-M members, the values of the bits b_7 and b_6 in the domain separation need to be adapted accordingly.

Security Claims

Attack Models. We consider two models of adversaries: nonce-respecting (NR) and nonce-misusing (NM)[1]. In the former model, nonce values in encryption queries (the tuples (N, A, M)) may be chosen by the adversary but they must be distinct. In the latter, nonce values in encryption queries can repeat. Basically, an NM adversary can arbitrarily repeat a nonce, hence even using the same nonce for all queries is possible. We can further specify NM by the distribution of a nonce, such as the maximum number of repetition of a nonce in the encryption queries.

For both models, adversaries can use any nonce values in decryption queries (the tuples (N, A, C, T)): it can collide with a nonce in an encryption query or with other decryption queries.

Security Claims. Our security claims are summarized in Table 3.1. The variables in the table denote the required workload, in terms of data complexity, of an adversary to break the cipher, in logarithm base 2. The data complexity of attacker consists of the number of queries and the total amount of processed message blocks. If it reaches the suggested number, then there is no security guarantee anymore, and the cipher can be broken. For simplicity, small constant factors, which are determined from the concrete security bounds, are neglected in these tables. A more detailed analysis is given in Section 4.

We claim these numbers hold as long as Skinny is a tweakable pseudorandom permutation, that is, it is computationally hard to distinguish Skinny from the set of uniform random permutations (URP) indexed by the tweak (a tweakable URP or TURP), using chosen-plaintext queries in the single-key setting.

Table 3.1: Security claims of Romulus. NR denotes Nonce-Respecting adversary and NM denotes Nonce-Misusing adversary.

Family	NR-Priv	NR-Auth	NM-Priv	NM-Auth
Romulus-N	128	128	–	–
Romulus-M	128	128	$64 \sim 128$	$64 \sim 128$

For all the members of Romulus-N, Table 3.1 shows n-bit security for privacy and authenticity against NR adversary. For all the members of Romulus-M, Table 3.1 shows n-bit security for privacy and authenticity against NR adversary and in addition, $n/2$-bit security for privacy and authenticity against NM adversary. The $n/2$-bit security assumes that the NM adversary has full control over the nonce, but in practice, the nonce repetition can happen accidentally, and it is conceivable that the nonce is repeated only a few times. As we present in Section 4, the security bounds of Romulus-M show the notable property of graceful security degradation with respect to the number

[1] Also known as Nonce Repeating or Nonce Ignoring. We chose "Nonce Misuse" for notational convenience of using acronyms, NR for nonce-respecting and NM for nonce-misuse.

of nonce repetition [34]. This property is similar to SCT, and if the number of nonce repetition is limited, the actual security bound is close to the full n-bit security.

Table 3.1 does not show the time complexity. We claim k-bit time complexity of attacker for all the members of Romulus that use Skinny with k-bit keys, which is common to schemes having security proofs in the standard model. This also indicates that the time complexity of key recovery is k bits, *i.e.*, key recovery is no easier than attacking Skinny itself, under the single-key setting. Note that all members have $k = 128$. See Table 3.2.

Table 3.2: Security claims of Romulus against key recovery.

Family	Key Recovery
Romulus-N	128
Romulus-M	128

Security Analysis

4.1 Security Notions

Security Notions for NAE. We consider the standard security notions for nonce-based AE [6, 7, 37]. Let Π denote an NAE scheme consisting of an encryption procedure $\Pi.\mathcal{E}_K$ and a decryption procedure $\Pi.\mathcal{D}_K$, for secret key K uniform over set $\mathcal{K}$ (denoted as $K \xleftarrow{\$} \mathcal{K}$). For plaintext M with nonce N and associated data A, $\Pi.\mathcal{E}_K$ takes (N, A, M) and returns ciphertext C (typically $|C| = |M|$) and tag T. For decryption, $\Pi.\mathcal{D}_K$ takes (N, A, C, T) and returns a decrypted plaintext M if authentication check is successful, and otherwise an error symbol, $\perp$.

The privacy notion is the indistinguishability of encryption oracle $\Pi.\mathcal{E}_K$ from the random-bit oracle \$ which returns random $|M| + \tau$ bits for any query (N, A, M). The adversary is assumed to be nonce-respecting. We define the privacy advantage as

$$\mathbf{Adv}_{\Pi}^{\mathtt{priv}}(\mathcal{A}) \overset{\text{def}}{=} \Pr\left[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{\Pi.\mathcal{E}_K(\cdot,\cdot,\cdot)} \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{\$(\cdot,\cdot,\cdot)} \Rightarrow 1\right],$$

which measures the hardness of breaking privacy notion for $\mathcal{A}$.

The authenticity notion is the probability of successful forgery via queries to $\Pi.\mathcal{E}_K$ and $\Pi.\mathcal{D}_K$ oracles. We define the authenticity advantage as

$$\mathbf{Adv}_{\Pi}^{\mathtt{auth}}(\mathcal{A}) \overset{\text{def}}{=} \Pr\left[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{\Pi.\mathcal{E}_K(\cdot,\cdot,\cdot),\Pi.\mathcal{D}_K(\cdot,\cdot,\cdot,\cdot)} \text{ forges}\right],$$

where $\mathcal{A}$ forges if it receives a value $M' \neq \perp$ from $\Pi.\mathcal{D}_K$. Here, to prevent trivial wins, if $(C, T) \leftarrow \Pi.\mathcal{E}_K(N, A, M)$ is obtained earlier, $\mathcal{A}$ cannot query (N, A, C, T) to $\Pi.\mathcal{D}_K$. The adversary is assumed to be nonce-respecting for encryption queries.

Security Notion for TBC. The security of TBC: $\mathcal{K} \times \mathcal{T} \times \mathcal{M} \rightarrow \mathcal{M}$ is defined by the indistinguishability from an ideal object, tweakable uniform random permutation (TURP), denoted by $\widetilde{\mathsf{P}}$, using chosen-plaintext, chosen-tweak queries. It is a set of independent uniform random permutations (URPs) over $\mathcal{M}$ indexed by tweak $T \in \mathcal{T}$. Let $\mathbf{Adv}_{\widetilde{E}}^{\mathtt{tprp}}(\mathcal{A})$ denote the TPRP advantage of TBC $\widetilde{E}$ against adversary $\mathcal{A}$. It is defined as

$$\mathbf{Adv}_{\widetilde{E}}^{\mathtt{tprp}}(\mathcal{A}) \overset{\text{def}}{=} \Pr\left[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{\widetilde{E}_K(\cdot,\cdot)} \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{\widetilde{\mathsf{P}}(\cdot,\cdot)} \Rightarrow 1\right].$$

Security Notions for MRAE. We adopt the security notions of MRAE following the same security definitions as above, with the exception that the adversary can now repeat nonces. We write the corresponding privacy advantage as

$$\mathbf{Adv}_{\Pi}^{\mathtt{nm\text{-}priv}}(\mathcal{A}) \overset{\text{def}}{=} \Pr\left[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{\Pi.\mathcal{E}_K(\cdot,\cdot,\cdot)} \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{\$(\cdot,\cdot,\cdot)} \Rightarrow 1\right],$$

and the authenticity advantage as

$$\mathbf{Adv}^{\texttt{nm-auth}}_{\Pi}(\mathcal{A}) \overset{\text{def}}{=} \Pr\left[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{\Pi.\mathcal{E}_K(\cdot,\cdot,\cdot),\Pi.\mathcal{D}_K(\cdot,\cdot,\cdot,\cdot)} \text{ forges }\right].$$

We note that while NM adversaries can repeat nonces, we without loss of generality assume that they do not repeat the same query. See also [38] for reference.

4.2 Security of Romulus-N

For $A \in \{0,1\}^*$, we say A has a AD blocks if it is parsed as $(A[1],\dots,A[a]) \xleftarrow{n,t} A$. Let $\tilde{a} = \lfloor a/2 \rfloor + 1$ which is a bound of actual number of primitive calls for AD. Similarly for plaintext $M \in \{0,1\}^*$, we say M has m message blocks if $|M|_n \overset{\text{def}}{=} \lceil |M|/n \rceil = m$. The same applies to ciphertext C. For encryption query (N, A, M) or decryption query (N, A, C, T) of a AD blocks and m message blocks, the number of total TBC calls is at most $\tilde{a} + m$, which is called the number of *effective blocks* of a query.

Let $\mathcal{A}$ be an NR adversary against Romulus-N using q encryption queries with time complexity t_A and with total number of effective blocks σ_{priv}. Moreover, let $\mathcal{B}$ be an NR adversary using q_e encryption queries and q_d decryption queries, with total number of effective blocks for encryption and decryption queries σ_{auth} and those only for decryption queries σ_{dec}, and time complexity t_B. Then

$$\mathbf{Adv}^{\texttt{priv}}_{\text{Romulus-N}}(\mathcal{A}) \le \mathbf{Adv}^{\texttt{tprp}}_{\widetilde{E}}(\mathcal{A}'),$$
$$\mathbf{Adv}^{\texttt{auth}}_{\text{Romulus-N}}(\mathcal{B}) \le \mathbf{Adv}^{\texttt{tprp}}_{\widetilde{E}}(\mathcal{B}') + \frac{2\sigma_{\text{dec}}}{2^n} + \frac{2q_d}{2^\tau}$$

hold for some $\mathcal{A}'$ using σ_{priv} chosen-plaintext queries with time complexity $t_A + O(\sigma_{\text{priv}})$, and for some $\mathcal{B}'$ using σ_{auth} chosen-plaintext queries with time complexity $t_B + O(\sigma_{\text{auth}})$. These bounds hold for all the members of Romulus-N. Note that $(n,\tau) = (128,128)$ holds for all the members. If $1 \le \tau < n$ (which is not a part of our submission), it still keeps n-bit privacy and τ-bit authenticity.

The security of Romulus-N crucially relies on the $n \times n$ matrix G defined over GF(2). Let $G^{(i)}$ be an $n \times n$ matrix that is equal to G except the $(i+1)$-st to n-th rows, which are set to all zero. Here, $G^{(0)}$ is the zero matrix and $G^{(n)} = G$, and for $X \in \{0,1\}^n$, $G^{(i)}(X) = \texttt{lsb}_i(G(X)) \| 0^{n-i}$ for all $i = 0, 8, 16, \dots, n$; note that all variables are byte strings, and $\texttt{lsb}_i(X)$ is the leftmost $i/8$ bytes (Section 2). Let I denote the $n \times n$ identity matrix. We say G is sound if (1) G is regular and (2) $G^{(i)} + I$ is regular for all $i = 8, 16, \dots, n$. The above security bounds hold as long as G is sound. The proofs are similar to those for iCOFB [15]. We have verified the soundness of our G, for a range of n including $n = 64$ and $n = 128$, by a computer program.

4.3 Security of Romulus-M

Let $\mathcal{A}$ be an adversary against Romulus-M using q encryption queries with time complexity t_A and with total number of effective blocks σ_{priv}. Here, for an encryption query (N, A, M), we define the number of effective blocks as $\lfloor a/2 \rfloor + \lfloor m/2 \rfloor + 2 + m'$, where $(A[1],\dots,A[a]) \xleftarrow{n,t} A$, $(M[1],\dots,M[m]) \xleftarrow{n,t} M$, and $(M[1],\dots,M[m']) \xleftarrow{n} M$. In the NR case, we have

$$\mathbf{Adv}^{\texttt{priv}}_{\text{Romulus-M}}(\mathcal{A}) \le \mathbf{Adv}^{\texttt{tprp}}_{\widetilde{E}}(\mathcal{A}'),$$

and in the NM case, we have

$$\mathbf{Adv}^{\texttt{nm-priv}}_{\text{Romulus-M}}(\mathcal{A}) \le \mathbf{Adv}^{\texttt{tprp}}_{\widetilde{E}}(\mathcal{A}') + \frac{4r\sigma_{\text{priv}}}{2^n},$$

where r is the maximum number of the repetition of a nonce in encryption queries, and $\mathcal{A}'$ uses σ_{priv} chosen-plaintext queries with time complexity $t_A + O(\sigma_{\mathrm{priv}})$.

Let $\mathcal{B}$ be an adversary using q_e encryption queries and q_d decryption queries, with total number of effective blocks for encryption and decryption queries σ_{auth}, and time complexity t_B. Here, for a decryption query (N, A, C, T), the number of effective blocks is defined as $\lfloor a/2 \rfloor + \lfloor m/2 \rfloor + 2 + m'$, where $(A[1], \dots, A[a]) \xleftarrow{n,t} A$, $(C[1], \dots, C[m]) \xleftarrow{n,t} C$, and $(C[1], \dots, C[m']) \xleftarrow{n} C$. We also define ℓ as the maximum effective block length of all the encryption and decryption queries.

Then in the NR case, we have

$$\mathbf{Adv}^{\mathtt{auth}}_{\mathsf{Romulus\text{-}M}}(\mathcal{B}) \le \mathbf{Adv}^{\mathtt{tprp}}_{\widetilde{E}}(\mathcal{B}') + \frac{2\ell q_d}{2^n} + \frac{2q_d}{2^n}.$$

In the NM case, we have

$$\mathbf{Adv}^{\mathtt{nm\text{-}auth}}_{\mathsf{Romulus\text{-}M}}(\mathcal{B}) \le \mathbf{Adv}^{\mathtt{tprp}}_{\widetilde{E}}(\mathcal{B}') + \frac{2r\ell q_d}{2^n} + \frac{2rq_d}{2^n}.$$

Here, r is the maximum number of the repetition of a nonce in encryption queries, and $\mathcal{B}'$ uses σ_{auth} chosen-plaintext queries with time complexity $t_B + O(\sigma_{\mathrm{auth}})$. While the term $O(\frac{r\ell q_d}{2^n})$ is present, this term is very likely not tight, and we expect that this can be improved to $O(r\sigma_{\mathrm{dec}}/2^n)$ with total effective blocks in decryption queries σ_{dec}.

4.4 Security of Skinny

Skinny [2,3] is claimed to be secure against related-tweakey attacks, an attack model very generous to the adversary as he can fully control the tweak input. We refer to the original research paper for the extensive security analysis provided by the authors (differential cryptanalysis, linear cryptanalysis, meet-in-the-middle attacks, impossible differential attacks, integral attacks, slide attacks, invariant subspace cryptanalysis, and algebraic attacks). In particular, strong security guarantees for Skinny have been provided with regards to differential and linear cryptanalysis.

In addition, since the publication of the cipher in 2016 there has been lots of cryptanalysis or structural analysis (improvement of security bounds) of Skinny by third parties. This was also further motivated by the organization of cryptanalysis competitions of Skinny by the designers.

To the best of our knowledge, the cryptanalysis that can attack the highest number of rounds (related-tweakey impossible differential attack [30, 39]) can only reach 23 of the 48 rounds of Skinny-128-256 and 27 of the 56 rounds of Skinny-128-384, with a very high data/memory/time complexity.

All in all, we can conclude that all versions of Skinny that we use have a very large security margin (more than 50%), even after numerous third party cryptanalysis. This is a very strong argument for Skinny, as it provides excellent performances while maintaining a very safe security margin. We emphasize that comparison between ciphers should take into account this security margin aspect (for example by normalizing performances by the maximum ratio of attacked rounds).

Features

The primary goal of Romulus is to provide a lightweight, yet highly-secure, highly-efficient AE based on a TBC. Romulus has a number of desirable features. Below we detail some representative ones:

- **Security margin.** Skinny family of tweakable block ciphers was published at CRYPTO 2016. Even though a thorough security analysis was provided by the authors in the original article, these primitives attracted a lot of attention and third party cryptanalysis in the past years. So far, Skinny functions still offer a very comfortable security margin. For example, the Skinny members used in Romulus still have more than 50% security margin in the related-key related-tweakey model. Actually the security margin rate is probably even higher as these attacks can't be directly applied to Skinny in the Romulus setting due to data limitations, limited tweak space, etc. Moreover, our security assumption on the internal primitive is only single-key, not related-key.

- **Security proofs.** Both Romulus-N and Romulus-M have provable security reductions to Skinny in the standard model. This is very important for high security confidence of Romulus and allows us to rely on the security of Romulus to that of Skinny, which has been extensively studied since the proposal in 2016.

- **Beyond-birthday-bound security.** The security bounds of Romulus shown in Section 4 are comparable to the state-of-the-art TBC modes of operation, namely ΘCB3 for NAE and SCT for MRAE. In particular, Romulus-N and Romulus-M (under NR adversary) achieve beyond-birthday-bound (BBB) security with respect to the block length. This level of security is much stronger than the up-to-birthday-bound, $n/2$-bit security achieved by conventional block cipher modes using n-bit block ciphers, *e.g.* GCM. Our provable security results are in the standard model, where there is a reduction from the security of the entire modes to the underlying primitive, Skinny, where the security of Skinny refers to the standard single-key setting. This implies that, up to the security bounds, our schemes cannot be broken without breaking the security of the underlying primitive in the single-key setting.

- **Misuse resistance.** Romulus-M is an MRAE mode which is secure against misuse (repeat) of nonces in encryption queries. More formally, it provides the best-possible security against nonce repeat in that ciphertexts do not give any information as long as the uniqueness of the input tuple (N, A, M) is maintained. In contrast to this, popular nonce-based AE modes are often vulnerable against nonce repeat, even one repetition can be significant. For example, the famous nonce repeat attack against GCM [19, 24] reveals its authentication key.

- **Performances.** Romulus-N is smaller than ΘCB3 in that it does not need an additional state beyond the internal TBC. Besides, it is faster as it processes $(n+t)$-bit AD blocks per TBC call. In general, it requires only $\lceil \frac{|A|-n}{n+t} \rceil + \lceil \frac{|M|}{n} \rceil + 1$ TBC calls, as opposed to ΘCB3, which requires $\lceil \frac{|A|}{n} \rceil + \lceil \frac{|M|}{n} \rceil + 1$. Although Romulus is serial in nature, *i.e.*, not parallelizable, it was shown during the CAESAR competition that parallelizability does not lead to significant performance gains in hardware performance, [17, 25, 28]. Moreover, parallelizability is not considered crucial for in lightweight applications, so it is a small price for a simple, small and fast design.

In Romulus-M, a plaintext is processed twice, once for generating a tag and once for encryption. Romulus-M inherits the overall design of Romulus-N, and thanks to the highly efficient tag generation, the efficiency loss is minimized. Romulus-M is about only 1.5 times slower than Romulus-N when associated data is empty, and becomes closer to Romulus-N for long associated data.

- **Simplicity/Small footprint.** Romulus has a quite small footprint. Especially for Romulus-N, we essentially need what is needed to implement the TBC Skinny itself. We remark that this becomes possible thanks to the permutation-based structure of Skinny's tweakey schedule, which allows to share the state registers used for storing input variable and for deriving round-key values. Thus, this feature is specific to our use of Skinny, though one can expect a similar effect with TBC using a simple tweak(ey) schedule. There is no OCB-like masks applied to the primitive, and we do not need the inverse circuit for Skinny which was needed for ΘCB3. A comparison in Section 6 (Table 6.1) shows that Romulus-N is quite small and especially efficient in terms of a combined metric of size and speed, compared with other schemes.

  Romulus-M also has a small footprint due to the shared structure with Romulus-N.

- **Small messages.** Romulus-N has a small computational overhead, thus has a good performance for small messages. For example, it just needs two TBC calls to encrypt one-block AD and one-block message, *i.e.*, 16 Bytes of AD and 16 Bytes of message. In particular, in the authentication part, the first 16 Bytes of AD can be processed for free in that it is processed without calling the TBC.

- **Flexibility.** Romulus has a large flexibility. Generally, it is defined as a generic mode for TBCs, and the provable security reduction under standard model contributes to a high confidence of the scheme when combined with a secure TBC.

- **Side channels and Fault Attacks.** Romulus does not inherently guarantee security against Side Channel Analysis and Fault Attacks. However, standard countermeasures are easily adaptable for Romulus, e.g. Fresh Rekeying [31], Masking [32], etc. Moreover, powerful fault attacks that require a small number of faults and pairs of faulty and non-faulty ciphertexts, such as DFA, are not applicable to Romulus-N without violating the security model, *i.e.*, repeating the nonce or releasing unverified plaintexts.

  We also note that in Romulus-N1, Romulus-N2, Romulus-M1 and Romulus-M3, we do not require the full tweakey size of Skinny-128-384, so a potential countermeasure to both SCA and DFA is to randomize the round keys by adding a random value to each TBC call. The downfall of this idea is that the randomness needs to be synchronized for correct decryption, but this property is shared with most SCA and DFA randomized countermeasures. However, we plan to analyze this idea and other ideas to make Romulus resistant to such attacks in details in subsequent works.

Design Rationale

6.1 Overview

Romulus is designed with the following goals in mind:

1. Have a very small area compared to other TBC/BC based AEAD modes.
2. Have relatively high efficiency in general.
3. Smaller overhead and fewer TBC calls for the AD processing.
4. Use the underlying TBC as a black box, with the standard security reduction to the TBC.

6.2 Mode Design

Rationale of NAE Mode. Romulus-N has a similar structure as a mode called iCOFB, which appeared in the full version of CHES 2017 paper [15]. Because it was introduced to show the feasibility of the main proposal of [13], block cipher mode COFB, it does not work as a full-fledged AE using conventional TBCs. Therefore, starting from iCOFB, we apply numerous changes for improving efficiency while achieving high security. As a result, Romulus-N becomes a much more advanced, sophisticated NAE mode based on a TBC. The security bound of Romulus-N is essentially equivalent to ΘCB3, having full n-bit security (though a difference in their bounds regarding the decryption queries exists, $q_d/2^n$ for ΘCB3 and $\sigma_{\text{dec}}/2^n$ for Romulus-N).

Rationale of MRAE Mode. Romulus-M is designed as an MRAE mode following the structure of SIV [38] and SCT [34]. Romulus-M reuses the components of Romulus-N as much as possible to inherit its implementation advantages and the security. In fact, this brings us several advantages (not only for implementation aspects) over SIV/SCT. Compared with SCT, Romulus-M needs a fewer number of primitive calls thanks to the faster MAC part. Moreover, Romulus-M has a smaller state than SCT because of single-state encryption part taken from Romulus-N (SCT employs a variant of counter mode). The provable security of Romulus-M is equivalent to SCT: the security depends on the maximum number of repetition of a nonce in encryption (r), and if $r = 1$ (*i.e.*, NR adversary) we have the full n-bit security. Security will gradually decreasing as r increases, also known as "graceful degradation", and even if r equals to the number of encryption queries, implying nonces are fixed, we maintain the birthday-bound, $n/2$-bit security.

ZAE [21] is another TBC-based MRAE. Although it is faster than SCT, the state size is much larger than SCT and Romulus-M.

Table 6.1: Features of Romulus-N members compared to ΘCB3 and other lightweight AEAD algorithms: λ is the bit security level of a mode. Here, (n, k)-BC is a block cipher of n-bit block and k-bit key, (n, t, k)-TBC is a TBC of n-bit block and k-bit key and t-bit tweak, and n-Perm is an n-bit cryptographic permutation.

Scheme	Number of Primitive Calls	Primitive	Security (λ)	State Size (S)	Rate (R)	S/R	Inverse Free
Romulus-N1	$\lceil \frac{\lvert A\rvert-n}{2n} \rceil + \lceil \frac{\lvert M\rvert}{n} \rceil + 1$	$(n, 1.5n, k)$-TBC†, $n = k$	n	$n + 2.5k = 3.5\lambda$	1	3.5λ	Yes
Romulus-N2	$\lceil \frac{\lvert A\rvert-n}{1.75n} \rceil + \lceil \frac{\lvert M\rvert}{n} \rceil + 1$	$(n, 1.2n, k)$-TBC†, $n = k$	n	$n + 2.2k = 3.2\lambda$	1	3.2λ	Yes
Romulus-N3	$\lceil \frac{\lvert A\rvert-n}{1.75n} \rceil + \lceil \frac{\lvert M\rvert}{n} \rceil + 1$	(n, n, k)-TBC, $n = k$	n	$n + 2k = 3\lambda$	1	3λ	Yes
COFB [14]	$\lceil \frac{\lvert A\rvert}{n} \rceil + \lceil \frac{\lvert M\rvert}{n} \rceil + 1$	(n, k)-BC, $n = k$	$n/2 - \log_2 n/2$	$1.5n + k = 5.4\lambda^\ddagger$	1	5.4λ	Yes
ΘCB3 [27]	$\lceil \frac{\lvert A\rvert}{n} \rceil + \lceil \frac{\lvert M\rvert}{n} \rceil + 1$	$(n, 1.5n, k)$-TBC$^\sharp$, $n = k$	n	$2n + 2.5k = 4.5\lambda$	1	4.5λ	No
Beetle [12]	$\lceil \frac{\lvert A\rvert}{n} \rceil + \lceil \frac{\lvert M\rvert}{n} \rceil + 2$	$2n$-Perm, $n = k$	$n - \log_2 n$	$2n = 2.12\lambda$	1/2	4.24λ	Yes
Ascon-128 [16]	$\lceil \frac{\lvert A\rvert}{n} \rceil + \lceil \frac{\lvert M\rvert}{n} \rceil + 1$	$5n$-Perm, $n = k/2$	$n/2$	$7n = 3.5\lambda$	1/5	17.5λ	Yes
Ascon-128a [16]	$\lceil \frac{\lvert A\rvert}{n} \rceil + \lceil \frac{\lvert M\rvert}{n} \rceil + 1$	$2.5n$-Perm, $n = k$	n	$3.5n = 3.5\lambda$	1/2.5	8.75λ	Yes
SpongeAE $^\flat$ [9]	$\lceil \frac{\lvert A\rvert}{n} \rceil + \lceil \frac{\lvert M\rvert}{n} \rceil + 1$	$3n$-Perm, $n = k$	n	$3n = 3\lambda$	1/3	9λ	Yes

† Unused part of tweakey is not a part of state thus not considered;
‡ Can possibly be enhanced to about 4λ with a $2n$-bit block cipher;
$^\sharp$ $1.5n$-bit tweak for n-bit nonce and $0.5n$-bit counter;
$^\flat$ Duplex construction with n-bit rate, $2n$-bit capacity.

Efficiency Comparison. In Table 6.1, we compare Romulus-N to ΘCB3, a well-studied TBC-based AEAD mode, in addition to a group of recently proposed lightweight AEAD modes. State size is the minimum number of bits that the mode has to maintain during its operation, and rate is the ratio of input data length divided by the total output length of the primitive needed to process that input. The comparison follows the following guidelines, while trying to be fair in comparing designs that follow completely different approaches:

1. $k = 128$ for all the designs.
2. n is the input block size (in bits) for each primitive call.
3. λ is the security level of the design.
4. For BC/TBC based designs, the key is considered to be stored inside the design, but we also consider that the encryption and decryption keys are interchangeable, *i.e.*, the encryption key can be derived from the decryption key and vice versa. Hence, no need to store the master key in additional storage. The same applies for the nonce.
5. For Sponge and Sponge-like designs, if the key/nonce are used only during initialization, then they are counted as part of the state and do not need extra storage. However, in designs like Ascon, where the key is used again during finalization, we assume the key storage is part of the state, as the key should be supplied only once as an input.

Our comparative analysis shows that Romulus-N is smaller and more efficient than ΘCB3 for the same security level. Moreover, the cost of processing AD is about half that of the message. For example, in the case of Romulus-N1, if the message and AD have equal length, there is an extra speed up of ~ 1.33x, which means that the efficiency even increases from 3.5λ to 2.625λ, compared to 4.5λ in case of ΘCB3, which makes Romulus-N a very promising candidate for NAE, for both short and long messages.

Similar comparison is shown in Table 6.2 for Misuse-Resistant TBC-based AEAD modes. It shows that Romulus-M is very efficient. Not only the state size is smaller, but also it is faster. For example, Romulus-M1 is 25% faster (1.33x speed-up) than SCT for the same parameters, when $|A| = 0$, and it is even faster when $|A| > 0$.

Table 6.2: Features of Romulus-M members compared to other MRAE modes : λ is the bit security level of a mode. Here, (n, k)-BC is a block cipher of n-bit block and k-bit key, (n, t, k)-TBC is a TBC of n-bit block and k-bit key and t-bit tweak. Security is for Nonce-respecting adversary.

Scheme	Number of Primitive Calls	Primitive	Security (λ)	State Size (S)	Rate (R)	S/R	Inverse Free
Romulus-M1	$\lceil \frac{\lvert A\rvert+\lvert M\rvert-n}{2n} \rceil + \lceil \frac{\lvert M\rvert}{n} \rceil + 1$	$(n, 1.5n, k)$-TBC†, $n = k$	n	$n + 2.5k = 3.5\lambda$	1/2	7λ	Yes
Romulus-M2	$\lceil \frac{\lvert A\rvert+\lvert M\rvert-n}{1.75n} \rceil + \lceil \frac{\lvert M\rvert}{n} \rceil + 1$	$(n, 1.2n, k)$-TBC†, $n = k$	n	$n + 2.2k = 3.2\lambda$	1/2	6.4λ	Yes
Romulus-M3	$\lceil \frac{\lvert A\rvert+\lvert M\rvert-n}{1.75n} \rceil + \lceil \frac{\lvert M\rvert}{n} \rceil + 1$	(n, n, k)-TBC, $n = k$	n	$n + 2k = 3\lambda$	1/2	6λ	Yes
SCT ‡ [35]	$\lceil \frac{\lvert A\rvert+\lvert M\rvert}{n} \rceil + \lceil \frac{\lvert M\rvert}{n} \rceil + 1$	(n, n, k)-TBC, $n = k$	n	$4n = 4\lambda$	1/2	8λ	Yes
SUNDAE [1]	$\lceil \frac{\lvert A\rvert+\lvert M\rvert}{n} \rceil + \lceil \frac{\lvert M\rvert}{n} \rceil + 1$	(n, k)-BC, $n = k$	$n/2$	$2n = 4\lambda$	1/2	8λ	Yes
ZAE ♯ [21]	$\lceil \frac{\lvert A\rvert+\lvert M\rvert}{2n} \rceil + \lceil \frac{\lvert M\rvert}{n} \rceil + 6$	(n, n, k)-TBC, $n = k$	n	$7n = 7\lambda$	1/2	14λ	Yes

† Unused part of tweakey is not a part of state thus not considered;
‡ Tag is n bits;
♯ Tag is $2n$ bits;

Rationale of TBC. We chose some of the members of the Skinny family of tweakable block ciphers [4] as our internal TBC primitives. Skinny was published at CRYPTO 2016 and has received a lot of attention since its proposal. In particular, a lot of third party cryptanalysis has been provided (in part motivated by the organization of cryptanalysis competitions of Skinny by the designers) and this was a crucial point in our primitive choice. Besides, our mode requested a lightweight tweakable block cipher and Skinny is the main such primitive. It is very efficient and lightweight, while providing a very comfortable security margin. Provable constructions that turn a block cipher into a tweakable block cipher were considered, but they are usually not lightweight, not efficient, and often only guarantee birthday-bound security.

6.3 Hardware Implementations

General Architecture and Hardware Estimates. The goal of the design of Romulus is to have a very small area overhead over the underlying TBC, specially for the round-based implementations. In order to achieve this goal, we set two requirements:

1. There should be no extra Flip-Flops over what is already required by the TBC, since Flip-Flops are very costly ($4 \sim 7$ GEs per Flip-Flop).
2. The number of possible inputs to each Flip-Flop and outputs of the circuits have to be minimized. This is in order to reduce the number of multiplexers required, which is usually one of the cause of efficiency reduction between the specification and implementation.

One of the advantages of Skinny as a lightweight TBC is that it has a very simple datapath, consisting of a simple state register followed by a low-area combinational circuit, where the same circuit is used for all the rounds, so the only multiplexer required is to select between the initial input for the first round and the round output afterwards (Figure 6.1(a)), and it has been shown that this multiplexer can even have lower cost than a normal multiplexer if it is combined with the Flip-Flops by using Scan-Flops (Figure 6.1(b)) [22]. However, when used inside an AEAD mode, challenges arise, such as how to store the key and nonce, as the key scheduling algorithm will change these values after each block encryption. The same goes for the block counter. In order to avoid duplicating the storage elements for these values; one set to be used to execute the TBC and one set to be used by the mode to maintain the current value, we studied the relation between the original and final value of the tweakey. Since the key scheduling algorithm of Skinny is fully linear and has very low area (most of the algorithm is just routing and renaming of different bytes), the full algorithm can be inverted using a very small circuit that costs 64 XOR gates. Moreover, the LFSR computation required between blocks can be implemented on top of this circuit, costing 3 extra XOR gates. This operation can be computed in parallel to ρ, such that when the state is updated for the next block, the tweakey key required is also ready. This costs only ~ 67 XOR gates as opposed to ~ 320 Flip-Flops that will, otherwise, be needed to maintain the tweakey value. Hence, the mode was designed with the architecture in Figure 6.1(b) in mind, where only a full-width state-register is used, carrying the TBC state and tweakey values, and every cycle, it is either kept without change, updated with the TBC round output (which includes a single round of the key scheduling algorithm) or the output of a simple linear transformation, which consists of ρ/ρ^{-1}, the unrolled inverse key schedule and the block counter. In order estimate the hardware cost of Romulus-N1 the mode we consider the round based implementation with an $n/4$-bit input/output bus:

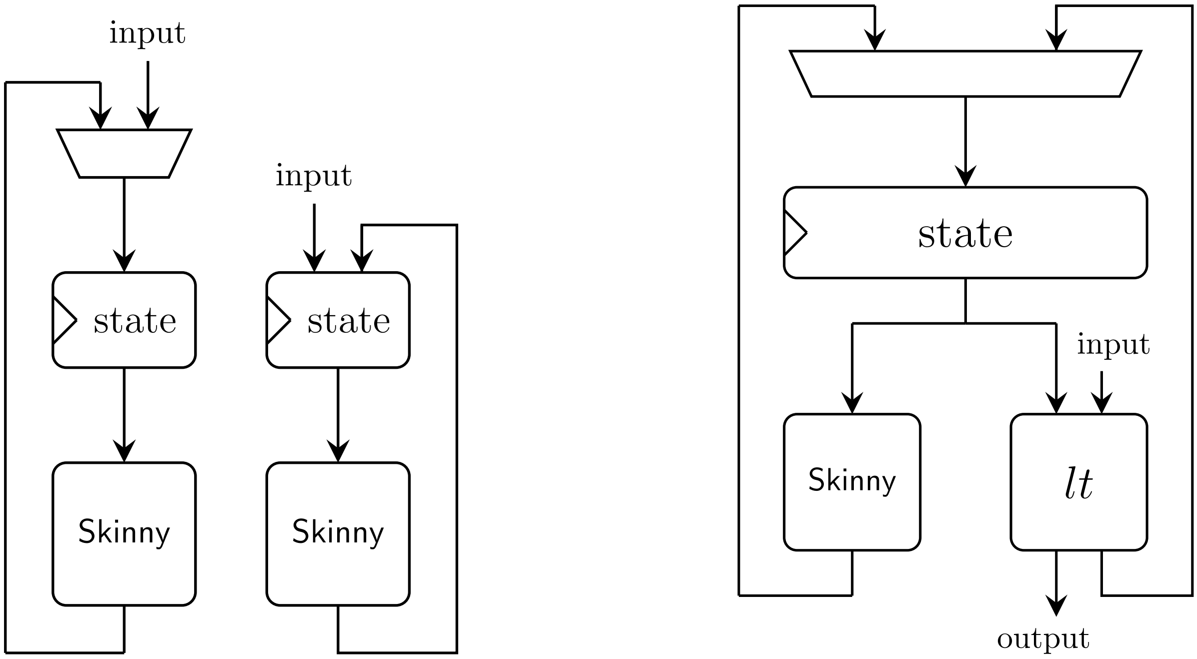
- 4 XOR gates for computing G.
- 64 XOR gates for computing ρ.
- 67 XOR gates for the correction of the tweakey and counting.
- 56 multiplexers to select whether to choose to increment the counter or not.
- 320 multiplexers to select between the output of the Skinny round and lt.

This adds up to 135 XOR gates and 376 multiplexers. For estimation purposes assume an XOR gate costs 2.25 GEs and a multiplexer costs 2.75 GEs, which adds up to 1337.75 GEs. In the original Skinny paper [2], the authors reported that Skinny-128-384 requires $4,268$ GEs, which adds up to $\sim 5,605$ GEs. This is ~ 1.4 KGEs smaller than the round based implementation of Ascon [18]. Moreover, a smart design can make use of the fact that 64 bits of the tweakey of Skinny-128-384 are not used, replacing 64 Flip-Flops by 64 multiplexers reducing an extra ~ 200 GEs. In order to design a combined encryption/decryption circuit, we show below that the decryption costs only extra 32 multiplexers and ~ 32 OR gates, or ~ 100 GEs. Similar analysis is done for Romulus-N2 and Romulus-N3, estmating that they would cost $1,217$ and $1,073$ GEs, respectively, on top of there corresponding Skinny variant, or $5,485$ and $4,385$ GEs, respectively.

These estimations show that Romulus-N is not just competitive theoretically but it can be a very attractive option practically for low area applications. For example, the 8-bit implementation of ACORN, the smallest implementation publicly available for all the round 3 candidates of the CAESAR competition, costs $5,900$ GEs, as shown in [28]. If we assume around $\sim 1,000$ GEs as the cost of the CAESAR Hardware API included in that design, as reported in [18], then Romulus-N3 is still smaller than that. Besides, we believe the area can be even lower using Serial Implementations of Skinny, which cost $\sim 3,000$ GE for Skinny-128-384 and $\sim 2,000$ GEs for Skinny-128-256, a gain of more than $1,000$ GEs compared to the round based implementation.

Another possible optimization is to consider the fact that most of the area of Skinny comes from the storage elements, hence, we can speed up Romulus to almost double the speed by using a simple two-round unrolling, which costs $\sim 1,000$ GEs, as only the logic part of Skinny needs replication, which is only $< 20\%$ increase in terms of area.

Romulus-M is estimated to have almost the same area as Romulus-N, except for an additional set of multiplexers in order to use the tag as an initial vector for the encryption part. This indicates that it can be a very lightweight choice for high security applications.



(a) Overview of the round based architecture of Skinny.

(b) Overview of the round based architecture of Romulus. *lt*: The linear transformation that includes ρ, block counter and inverse key schedule.

Figure 6.1: Expected architectures for Skinny and Romulus

6.4 Software Implementations

We refer to Skinny document for discussions on software implementations of the various Skinny versions. The Romulus mode will have little impact on the global performance of Skinny in software as long as serial implementations are used. We expect very little increase in ROM or RAM when compared to Skinny benchmarks. The very performant micro-controller implementations reported in the Skinny document were benchmarked without assuming parallel cipher calls, and without any pre-processing. Therefore, Romulus will present a very similar performance profile as the numbers reported on micro-controllers. Generally, using little amount of RAM, Skinny is easy and efficient to implement using simple table-based approach.

For high-end platforms, such as latest Intel processors, very efficient highly-parallel bitsliced implementations of Skinny using SSE, AVX, AVX2 instructions on XMM/YMM registers will not be directly applicable as our Romulus mode is serial in nature. However, in the classical case of a server communicating with many lightweight devices, we note that it would be possible to consider bitslicing the key schedule [8] of Skinny (being relatively simple to compute) or using scheduling strategies [10]. Classical table-based implementation of Skinny will ensure acceptable performance on even legacy platforms, while Vector Permute (`vperm`) might lead to better results on medium range platforms by parallelizing the computation of the Sbox.

6.5 Primitives Choices

LFSR-Based Counters. The NIST call for lightweight AEAD algorithms requires that such algorithms must allow encrypting messages of length at least 2^{50} bytes while still maintaining their security claims. This means that using a TBC whose block size is 128 bits, we need a block counter of a period of at least 2^{46}. While this can be achieved by a simple arithmetic counter of 46 bits, arithmetic counters can be costly both in terms of area ($3 \sim 5$ GEs/bit) and performance (due to the long carry chains which limit the frequency of the circuit). In order to avoid this, we decided to use LFSR-based counters, which can be implemented using a handful of XOR gates (3 XORs $\approx 6 \sim 9$ GEs). This, in addition to the architecture described above, makes the cost of counter almost negligible.

Counter Separation (Romulus-N2 and Romulus-M2). For Romulus-N2 and Romulus-M2, we used two LFSRs instead of one, such that the second LFSR is updated only when the first LFSR performs a full period. The rationale for that is to provide an even more lightweight variant of Romulus-N1 and Romulus-M1. Since one can argue that the limitation imposed by the NIST for the number of bytes is impractical for some lightweight applications, the designer can choose to support only up to $\sim 2^{28}$ bytes instead, and not use the second LFSR. This saves $64 \sim 128$ Flip-Flops.

Smaller $\mathcal{D}$ for Romulus-N3. The goal of Romulus-N3 is to fit the Romulus algorithm in Skinny-128-256, which is faster and smaller than Skinny-128-384. In most lightweight applications, the amount of data to be sent under the same key is small. Hence, Romulus-N3 represents a variant targeted at such applications that is faster and smaller than the other variants and can encrypt up to ~ 256 MBs of data.

Tag Generation. Considering hardware simplicity, the tag is the final output state (*i.e.*, the same way as the ciphertext blocks), as opposed to the final state S of the TBC. In order to avoid branching when it comes to the output of the circuit, the tag is generated as $G(S)$ instead of S. In hardware, this can be implemented as $\rho(S, 0^n)$, *i.e.*, similar to the encryption of a zero vector. Consequently, the output bus is always connected to the output of ρ and a multiplexer is avoided.

Padding. The padding function used in Romulus is chosen so that the padding information is always inserted in the most significant byte of the last block of the message/AD. Hence, it reduces the number of decisions for each byte to only two decisions (either the input byte or a zero byte, except the most significant byte which is either the input byte or the byte length of that block). Besides, it is also the case when the input is treated as a string of words (16-, 32-, 64- or 128-bit words). This is much simpler than the classical 10^* padding approach, where every word has a lot of different possibilities when it comes to the location of the padding string. Besides, usually implementations maintain the length of the message in a local variable/register, which means that the padding information is already available, just a matter of placing it in the right place in the message, as opposed to the decoder required to convert the message length into 10^* padding.

Padding Circuit for Decryption. One of the main features of Romulus is that it is inverse free and both the encryption and decryption algorithms are almost the same. However, it can be tricky to understand the behavior of decryption when the last ciphertext block has length $< n$. In order to understand padding in the decryption algorithm, we look at the ρ and ρ^{-1} functions when the input plaintext/ciphertext is partial. The ρ function applied on a partial plaintext block is shown in Equation (6.1). If ρ^{-1} is directly applied to $\mathtt{pad}_n(C)$, the corresponding output will be incorrect, due to the truncation of the last ciphertext block. Hence, before applying ρ^{-1} we need to regenerate the truncated bits. It can be verified that $C' = \mathtt{pad}_n(C) \oplus \mathtt{msb}_{n-|C|}(G(S))$. Once C' is regenerated, ρ^{-1} can be computed as shown in Equation (6.2):

$$\begin{bmatrix} S' \\ C' \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ G & 1 \end{bmatrix} \begin{bmatrix} S \\ \mathtt{pad}_n(M) \end{bmatrix} \quad \text{and} \quad C = \mathtt{lsb}_{|M|}(C'). \tag{6.1}$$

$$C' = \mathtt{pad}_n(C) \oplus \mathtt{msb}_{n-|C|}(G(S)) \quad \text{and} \quad \begin{bmatrix} S' \\ M \end{bmatrix} = \begin{bmatrix} 1 \oplus G & 1 \\ G & 1 \end{bmatrix} \begin{bmatrix} S \\ C' \end{bmatrix}. \tag{6.2}$$

While this looks like a special padding function, in practice it is simple. First of all, $G(S)$ needs to be calculated anyway. Besides, the whole operation can be implemented in two steps:

$$M = C \oplus \mathtt{lsb}_{|C|}(G(s)),$$
$$S' = \mathtt{pad}_n(M) \oplus S$$

which can have a very simple hardware implementation, as discussed in the next paragraph.

Encryption-Decryption Combined Circuit. One of the goals of Romulus is to be efficient for implementations that require a combine encryption-decryption datapath. Hence, we made sure that the algorithm is inverse free, *i.e.*, it does not used the inverse function of Skinny or $G(S)$. Moreover, ρ and ρ^{-1} can be implemented and combined using only one multiplexer, whose size depends on the size of the input/output bus. The same circuit can be used to solve the padding issue in decryption, by padding M instead of C. The tag verification operation simply checks if $\rho(S, 0^n)$ equals to T, which can be serialized depending on the implementation of ρ.

Choice of the G Matrix. We chose the position of G so that it is applied to the output state. This removes the need of G for AD processing, which improves software performance. In Section 6.2, we listed the security condition for G, and we choose our matrix G so that it meets these conditions and suits well for various hardware and software.

We noticed that for lightweight applications, most implementations use an input/output bus of width ≤ 32. Hence, we expect the implementation of ρ to be serialized depending on the bus size. Consequently, the matrix used in iCOFB can be inefficient as it needs a feedback operation over 4 bytes, which requires up to 32 extra Flip-Flops in order to be serialized, something we are trying to avoid in Romulus. Moreover, the serial operation of ρ is different for byte, which requires additional multiplexers.

However, we observed that if the input block is interpreted in a different order, both problems can be avoided. First, it is impossible to satisfy the security requirements of G without any feedback signals, *i.e.*, G is a bit permutation.

- If G is a bit permutation with at least one bit going to itself, then there is at least one non-zero value on the diagonal, so $I + G$ has at least 1 row that is all 0s.
- If G is a bit permutation without any bit going to itself, then every column in $I + G$ has exactly two 1's. The sum of all rows in such matrix is the 0 vector, which means the rows are linearly dependent. Hence, $I + G$ is not invertible.

However, the number of feedback signals can be adjusted to our requirements, starting from only 1 feedback signal. Second, we noticed that the input block/state of length n bits can be treated as several independent sub-blocks of size n/w each. Hence, it is enough to design a matrix G_s of size $w \times w$ bits and apply it independently n/w times to each sub-block. The operation applied on each sub-block in this case is the same (*i.e.*, as we can distribute the feedback bits evenly across the input block). Unfortunately, the choice of w and G_s that provides the optimal results depends on the implementation architecture. However, we found out that the best trade-off/balance across different architectures is when $w = 8$ and G_s uses a single bit feedback.

In order to verify our observations, we generated a family of matrices with different values of w and G_s, and measured the cost of implementing each of them on different architectures.

Implementations

7.1 Software Performances

7.1.1 Software implementations

The Skinny article presents extremely fast software implementations on various recent Intel processors, some as low as 2.37 c/B. However, these bitslice implementations are heavily relying on the parallelism offered by some operating modes. In our case, this parallelism is not present as Romulus is not a parallel mode. Therefore, the performance of Romulus on high-end servers will be closer to 20 c/B than 2 c/B.

However, in practice several easy solutions are possible to overcome this performance limitation. One solution is to let the two communicating entities to use short sessions, which would re-enable the server side to parallelise the encryption/decryption of the various sessions. Another possible solution to still use these very fast bitslice implementations is to let the server to communicate with several clients in parallel. This is in fact very probably what will happen in practice (a server communicating with many clients is the main reason why fast software implementations are interesting). Even in the case where the arriving data is always from new clients, bitslicing remains possible by bitslicing the key schedule part of Skinny as well.

7.1.2 Micro-controller implementations

The Skinny article also reports very efficient micro-controllers implementations of the Skinny-128-128, with various tradeoffs. Since these implementations do not require any parallelism, they can directly be applied in Romulus. However, since Romulus uses bigger versions of Skinny, the higher number of rounds will naturally reduce the performances accordingly. We expect the effect of the Romulus mode to be minor on the performances since it is rate 1.

7.2 ASIC Performances

We have implemented the round-based and 4-round unrolled architectures of Romulus-N1. The figures are exepected to be similar for Romulus-N2 and about 550 GEs smaller, and 12∼18% faster for Romulus-N3. Morover, Romulus and Remus [20] share a similar structure and our experimental results show that it costs only around 200 GEs to convert the implementation to the misuse resistant variant.The implementations are compliant with teh CAESAR Hardware API, so a more lightweight interface can be designed. We added our estimations to the table labeled by '*'.

Table 7.1: ASIC Round-Based Implementations of Romulus-N1 using the TSMC 65nm standard cell library. Power and Energy are estimated at 10 Mhz. Energy is for 1 TBC call.

Variant	**Cycles**	**Area w/o interface** (GE)	**Area** (GE)	**Minimum Delay** (ns)	**Throughput** (Gbps)	**Power** (μW)	**Energy** (pJ)	**Thput/Area** (Gbps/kGE)	**NR Security**	**NM Security**
Basic Iterative	60	5514	6620	1	2.78	548	32.8	0.42	128	-
Unrolled x4†	18	8231	9286	1.5	6.18	1325	23	0.67	128	-
Unrolled x4‡	18	9632	10748	1	9.27	-	-	0.86	128	-
Basic Iterative*	60	5000	6000	1	2.78	-	-	0.46	128	-
Unrolled x4*	18	9000	9700	1	9.27	-	-	0.95	128	-

† Minimum Area;

‡ 1 GHz;

* Estimation.

Acknowledgments

The second and fourth authors are supported by the Temasek Labs grant (DSOCL16194).

Bibliography

[1] Banik, S., Bogdanov, A., Luykx, A., Tischhauser, E.: SUNDAE: Small Universal Deterministic Authenticated Encryption for the Internet of Things. IACR Trans. Symmetric Cryptol. **2018**(3) (2018) 1–35

[2] Beierle, C., Jean, J., Kölbl, S., Leander, G., Moradi, A., Peyrin, T., Sasaki, Y., Sasdrich, P., Sim, S.M.: The SKINNY Family of Block Ciphers and Its Low-Latency Variant MANTIS. In: CRYPTO (2). Volume 9815 of Lecture Notes in Computer Science., Springer (2016) 123–153

[3] Beierle, C., Jean, J., Kölbl, S., Leander, G., Moradi, A., Peyrin, T., Sasaki, Y., Sasdrich, P., Sim, S.M.: The SKINNY Family of Block Ciphers and its Low-Latency Variant MANTIS. IACR Cryptology ePrint Archive **2016** (2016) 660

[4] Beierle, C., Jean, J., Kölbl, S., Leander, G., Moradi, A., Peyrin, T., Sasaki, Y., Sasdrich, P., Sim, S.M.: The SKINNY family of block ciphers and its low-latency variant MANTIS. In: Annual International Cryptology Conference, Springer (2016) 123–153

[5] Bellare, M., Boldyreva, A., Palacio, A.: An Uninstantiable Random-Oracle-Model Scheme for a Hybrid-Encryption Problem. In: EUROCRYPT. Volume 3027 of Lecture Notes in Computer Science., Springer (2004) 171–188

[6] Bellare, M., Namprempre, C.: Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. J. Cryptology **21**(4) (2008) 469–491

[7] Bellare, M., Rogaway, P., Wagner, D.A.: The EAX Mode of Operation. In: FSE. Volume 3017 of Lecture Notes in Computer Science., Springer (2004) 389–407

[8] Benadjila, R., Guo, J., Lomné, V., Peyrin, T.: Implementing Lightweight Block Ciphers on x86 Architectures. In Lange, T., Lauter, K.E., Lisonek, P., eds.: Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers. Volume 8282 of Lecture Notes in Computer Science., Springer (2013) 324–351

[9] Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications. In: Selected Areas in Cryptography. Volume 7118 of Lecture Notes in Computer Science., Springer (2011) 320–337

[10] Bogdanov, A., Lauridsen, M.M., Tischhauser, E.: Comb to Pipeline: Fast Software Encryption Revisited. In Leander, G., ed.: Fast Software Encryption - 22nd International Workshop, FSE 2015, Istanbul, Turkey, March 8-11, 2015, Revised Selected Papers. Volume 9054 of Lecture Notes in Computer Science., Springer (2015) 150–171

[11] Canetti, R., Goldreich, O., Halevi, S.: The Random Oracle Methodology, Revisited (Preliminary Version). In: STOC, ACM (1998) 209–218

[12] Chakraborti, A., Datta, N., Nandi, M., Yasuda, K.: Beetle Family of Lightweight and Secure Authenticated Encryption Ciphers. IACR Transactions on Cryptographic Hardware and Embedded Systems (2018) 218–241

[13] Chakraborti, A., Iwata, T., Minematsu, K., Nandi, M.: Blockcipher-Based Authenticated Encryption: How Small Can We Go? In: CHES. Volume 10529 of Lecture Notes in Computer Science., Springer (2017) 277–298

[14] Chakraborti, A., Iwata, T., Minematsu, K., Nandi, M.: Blockcipher-based Authenticated Encryption: How Small Can We Go? In: International Conference on Cryptographic Hardware and Embedded Systems, Springer (2017) 277–298

[15] Chakraborti, A., Iwata, T., Minematsu, K., Nandi, M.: Blockcipher-based Authenticated Encryption: How Small Can We Go? (Full version of [13]). IACR Cryptology ePrint Archive **2017** (2017) 649

[16] Dobraunig, C., Eichlseder, M., Mendel, F., Schläffer, M.: Ascon v1. 2. Submission to the CAESAR Competition (2016)

[17] George Mason University: ATHENa: Automated Tools for Hardware EvaluatioN. https://cryptography.gmu.edu/athena/ (2017)

[18] Groß, H., Wenger, E., Dobraunig, C., Ehrenhöfer, C.: Suit up!–Made-to-Measure Hardware Implementations of ASCON. In: 2015 Euromicro Conference on Digital System Design, IEEE (2015) 645–652

[19] Handschuh, H., Preneel, B.: Key-Recovery Attacks on Universal Hash Function Based MAC Algorithms. In Wagner, D., ed.: Advances in Cryptology - CRYPTO 2008. Volume 5157 of LNCS., Springer (2008) 144–161

[20] Iwata, T., Khairallah, M., Minematsu, K., Peyrin, T.: Remus v1. Submission to NIST Lightweight Cryptography Project (2019)

[21] Iwata, T., Minematsu, K., Peyrin, T., Seurin, Y.: ZMAC: a fast tweakable block cipher mode for highly secure message authentication. In: Annual International Cryptology Conference, Springer (2017) 34–65

[22] Jean, J., Moradi, A., Peyrin, T., Sasdrich, P.: Bit-Sliding: A Generic Technique for Bit-Serial Implementations of SPN-based Primitives. In: International Conference on Cryptographic Hardware and Embedded Systems, Springer (2017) 687–707

[23] Jean, J., Nikolic, I., Peyrin, T.: Tweaks and Keys for Block Ciphers: The TWEAKEY Framework. In: ASIACRYPT (2). Volume 8874 of Lecture Notes in Computer Science., Springer (2014) 274–288

[24] Joux, A.: Authentication Failures in NIST Version of GCM. Comments submitted to NIST Modes of Operation Process (2006) Available at http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/comments/800-38_Series-Drafts/GCM/Joux_comments.pdf.

[25] Khairallah, M., Chattopadhyay, A., Peyrin, T.: Looting the LUTs: FPGA optimization of AES and AES-like ciphers for authenticated encryption. In: International Conference in Cryptology in India, Springer (2017) 282–301

[26] Krovetz, T., Rogaway, P.: The Software Performance of Authenticated-Encryption Modes. In: FSE. Volume 6733 of Lecture Notes in Computer Science., Springer (2011) 306–327

[27] Krovetz, T., Rogaway, P.: The software performance of authenticated-encryption modes. In: International Workshop on Fast Software Encryption, Springer (2011) 306–327

[28] Kumar, S., Haj-Yihia, J., Khairallah, M., Chattopadhyay, A.: A Comprehensive Performance Analysis of Hardware Implementations of CAESAR Candidates. IACR Cryptology ePrint Archive **2017** (2017) 1261

[29] Liskov, M., Rivest, R.L., Wagner, D.A.: Tweakable Block Ciphers. In: CRYPTO. Volume 2442 of Lecture Notes in Computer Science., Springer (2002) 31–46

[30] Liu, G., Ghosh, M., Song, L.: Security Analysis of SKINNY under Related-Tweakey Settings (Long Paper). IACR Trans. Symmetric Cryptol. **2017**(3) (2017) 37–72

[31] Medwed, M., Standaert, F.X., Großschädl, J., Regazzoni, F.: Fresh re-keying: Security against side-channel and fault attacks for low-cost devices. In: International Conference on Cryptology in Africa, Springer (2010) 279–296

[32] Messerges, T.S.: Securing the AES finalists against power analysis attacks. In: International Workshop on Fast Software Encryption, Springer (2000) 150–164

[33] Moradi, A., Poschmann, A., Ling, S., Paar, C., Wang, H.: Pushing the Limits: A Very Compact and a Threshold Implementation of AES. In: EUROCRYPT. Volume 6632 of Lecture Notes in Computer Science., Springer (2011) 69–88

[34] Peyrin, T., Seurin, Y.: Counter-in-Tweak: Authenticated Encryption Modes for Tweakable Block Ciphers. In: CRYPTO (1). Volume 9814 of Lecture Notes in Computer Science., Springer (2016) 33–63

[35] Peyrin, T., Seurin, Y.: Counter-in-tweak: authenticated encryption modes for tweakable block ciphers. In: Annual International Cryptology Conference, Springer (2016) 33–63

[36] Rogaway, P.: Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC. In: ASIACRYPT. Volume 3329 of Lecture Notes in Computer Science., Springer (2004) 16–31

[37] Rogaway, P.: Nonce-Based Symmetric Encryption. In: FSE. Volume 3017 of Lecture Notes in Computer Science., Springer (2004) 348–359

[38] Rogaway, P., Shrimpton, T.: A Provable-Security Treatment of the Key-Wrap Problem. In: EUROCRYPT. Volume 4004 of Lecture Notes in Computer Science., Springer (2006) 373–390

[39] Sadeghi, S., Mohammadi, T., Bagheri, N.: Cryptanalysis of Reduced round SKINNY Block Cipher. IACR Trans. Symmetric Cryptol. **2018**(3) (2018) 124–162

Appendix

Table 8.1: Domain separation byte B of Romulus. Bits b_7 and b_6 are to be set to the appropriate value according to the parameter sets.

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	int(B)	case
Romulus-N	-	-	0	0	1	0	0	0	8	A main
	-	-	0	1	1	0	0	0	24	A last unpadded
	-	-	0	1	1	0	1	0	26	A last padded
	-	-	0	0	0	1	0	0	4	M main
	-	-	0	1	0	1	0	0	20	M last unpadded
	-	-	0	1	0	1	0	1	21	M last padded
Romulus-M	-	-	1	0	1	0	0	0	40	A main
	-	-	1	0	1	1	0	0	44	M auth main
	-	-	1	1	1	1	1	1	63	w: (even,even,padded,padded)
	-	-	1	1	1	1	1	0	62	w: (even,even,padded,unpadded)
	-	-	1	1	1	1	0	1	61	w: (even,even,unpadded,padded)
	-	-	1	1	1	1	0	0	60	w: (even,even,unpadded,unpadded)
	-	-	1	1	1	0	1	1	59	w: (even,odd,padded,padded)
	-	-	1	1	1	0	1	0	58	w: (even,odd,padded,unpadded)
	-	-	1	1	1	0	0	1	57	w: (even,odd,unpadded,padded)
	-	-	1	1	1	0	0	0	56	w: (even,odd,unpadded,unpadded)
	-	-	1	1	0	1	1	1	55	w: (odd,even,padded,padded)
	-	-	1	1	0	1	1	0	54	w: (odd,even,padded,unpadded)
	-	-	1	1	0	1	0	1	53	w: (odd,even,unpadded,padded)
	-	-	1	1	0	1	0	0	52	w: (odd,even,unpadded,unpadded)
	-	-	1	1	0	0	1	1	51	w: (odd,odd,padded,padded)
	-	-	1	1	0	0	1	0	50	w: (odd,odd,padded,unpadded)
	-	-	1	1	0	0	0	1	49	w: (odd,odd,unpadded,padded)
	-	-	1	1	0	0	0	0	48	w: (odd,odd,unpadded,unpadded)
	-	-	1	0	0	1	0	0	36	M enc main

Changelog

- 29-03-2019: version v1.0