Remus

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

Remus aims at lightweight, efficient, and highly-secure NAE and MRAE schemes, based on a TBC. As the underlying TBC, we adopt Skinny proposed at CRYPTO 2016 [3]. The security of this TBC has been extensively studied, and it has attractive implementation characteristics. Remus shares the basic structure with Romulus [17], which is a set of TBC-based modes based on Skinny. Therefore, Remus inherits the overall implementation advantages of Romulus.

The biggest difference of Remus from Romulus is the way it instantiates a TBC. Specifically, Remus takes an approach of utilizing the whole tweakey state of Skinny as a function of the key and tweak, using a tweak-dependent key derivation. In contrast to this, in Romulus, Skinny is used in the standard keying setting (*i.e.*, tweakey state takes a persistent key material and a changing tweak). The tweak-dependent key derivation allows us to use a smaller variant of Skinny than those used by Romulus, and brings us better efficiency and comparable bit security. The downside is that the security proof of Remus is not based on the standard assumption of its cryptographic core (namely, the pseudorandomness of Skinny) as was done by Romulus. Instead, we can prove the security of Remus by assuming Skinny as an ideal-cipher (thus ideal-cipher model proof), or, assuming the pseudorandomness of another TBC built on Skinny, called ICE . The latter is a standard model proof but the assumption is still different from the pseudorandomness assumption of Skinny. This means that Remus is not a simple optimization of Romulus but is a product of tread-off between (qualitative) security and efficiency.

We specify a set of members for Remus that have different TBC (ICE) instantiations based on a block cipher (taking Skinny as a block cipher) in order to provide security-efficiency tread-offs.

As well as Romulus-N [17], the overall structure of Remus-N shares similarity in part with a (TBC-based variant of) block cipher mode COFB [10, 12], yet, we make numerous refinements to achieve our design goal. Consequently, as a mode of TBC ICE, Remus-N achieves a significantly smaller state size than ΘCB3 [25], the typical choice for TBC-based AE mode, while keeping the equivalent efficiency (*i.e.*, the same number of TBC calls). Also Remus-N is inverse-free (*i.e.*, no TBC decryption routine is needed) unlike ΘCB3. For security, it allows either classical $n/2$-bit security or full n-bit security depending on the variant of ICE, for $n = 128$ being the block size of Skinny. We also define a variant with 64-bit security based on $n = 64$-bit block version of Skinny. The difference in ICE gives a security-area tread off, and 128-bit secure variant (Remus-N2) has the bit security equivalent to ΘCB3.

To see the superior performance of Remus-N, let us compare n-bit secure Remus-N2 with other size-oriented and n-bit secure AE schemes, such as conventional permutation-based AEs using $3n$-bit permutation with n-bit rate. Both have $3n$ state bits and process n-bit message per primitive call. However, the cryptographic primitive for Remus-N2 is expected to be much more lightweight

and/or faster because of smaller output size ($3n$ vs n bits). Moreover, our primitive has only n-bit tweakey, hence it is even smaller than the members of Romulus; they are n-bit secure and using tweakey state of $2n$ or $3n$ bits. Both permutation-based schemes and Remus rely on non-standard models (random permutation or ideal-cipher), and we emphasize that the security of Skinny inside Remus has been comprehensively evaluated, not only for the single-key related-tweak setting but also related-tweakey setting, which suggests strong reliability to be used as the ideal-cipher. Besides, we did not weaken the algorithm of Skinny from the original, say by reducing the number of rounds. This is a sharp difference from the strategy often taken in permutation-based constructions: it makes the underlying permutation much weaker than the stand-alone version for which the random permutation model is assumed, in order to boost the throughput.

An additional feature of Remus is that it offers a very flexible security/size tread-off without changing the throughput. In more detail, Remus contains $n/2$-bit secure variants (Remus-N1 and Remus-M1) and n-bit secure variants (Remus-N2 and Remus-M2). Their difference is only in the existence of the second (block) mask, which increases the state size. If the latter is too big and n-bit security is overkill, it is possible to derive an intermediate variant by truncating the second mask to (say) $n/2$ bits. It will be $(n + n/2)/2 = 3n/4$-bit secure. For simplicity, we did not include such variants in the official members of Remus, however, this flexibility would be useful in practice.

Remus-M follows the general construction of MRAE called SIV [39]. Remus-M reuses the components of Remus-N as much as possible, and Remus-M is simply obtained by processing message twice by Remus-N. Remus-M has an efficiency advantage over misuse-resistant variants of Romulus (Romulus-M). In particular, the high-security variant (Remus-M2) achieves n-bit security against nonce-respecting adversaries and $n/2$-bit security against nonce-misusing adversaries, which shows an equivalent level of security of Romulus-M to SCT. Thanks to the shared components, most of the advantages of Remus-N mentioned above also hold for Remus-M.

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

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 Remus, the recommended parameter sets, and the specification of TBC Skinny. In the last part of Section 2, we specify two families of Remus, Remus-N and Remus-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 Remus. 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 Remus 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 be the string of i zero bits, and for instance we write 10^i for $1 \,\|\, 0^i$. We denote $\mathtt{msb}_x(X)$ (resp. $\mathtt{lsb}_x(X)$) the truncation of X to its x most (resp. least) significant bits. See"Endian" paragraph below. Bitwise XOR of two variables X and Y is denoted by $X \oplus Y$, where $|X| = |Y| = c$ for some integer c. By convention, if one of X or Y is represented as an integer in $[\![2^c]\!]_0$ we assume a standard integer-to-binary encoding: for example $X \oplus 1$ denotes $X \oplus 0^{c-1}1$.

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

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

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 $|\varepsilon|_n = 1$.

Galois Field. An element a in the Galois field $\mathrm{GF}(2^n)$ will be interchangeably represented as an n-bit string $a_{n-1}\ldots a_1 a_0$, 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

$$X_7X_6 \dots X_0 \,\|\, X_{15}X_{14} \dots X_8 \,\|\, \dots \,\|\, X_{n-1}X_{n-2} \dots X_{n-8},$$

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, $\mathtt{lsb}_{16}(X) = (X_7X_6 \dots X_0 \,\|\, X_{15}X_{14} \dots X_8)$ 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 `msb` and `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 $\widetilde{E} : \mathcal{K} \times \mathcal{T}_\mathcal{W} \times \mathcal{M} \to \mathcal{M}$, where $\mathcal{K}$ is the key space, $\mathcal{T}_\mathcal{W}$ is the tweak space, and $\mathcal{M} = \{0,1\}^n$ is the message space, such that for any $(K, T_w) \in \mathcal{K} \times \mathcal{T}_\mathcal{W}$, $\widetilde{E}(K, T_w, \cdot)$ 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 $\mathcal{T}_\mathcal{W}$ 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

Remus has the following parameters:

- Nonce length $nl \in \{96, 128\}$.
- Key length $k = 128$.
- Message and AD block length $n \in \{64, 128\}$.
- Mode to convert a block cipher into a TBC, $\mathtt{ICmode} \in \{\mathsf{ICE1}, \mathsf{ICE2}, \mathsf{ICE3}\}$.
- Block cipher $E : \mathcal{K} \times \mathcal{M} \to \mathcal{M}$ with $\mathcal{M} = \{0,1\}^n$ and $\mathcal{K} = \{0,1\}^k$. Here, E is either Skinny-128/128 or Skinny-64/128, by seeing the whole tweakey space as the key space, and assuming certain tweakey encodings specified in Section 2.4.
- Counter bit length d. *Counter* refers the part of the tweakey that changes after each TBC call, for the same (N, K) pair. Each variants of Remus has $2^d - 1$ possible counter values for each (N, K) pair.
- Tag length $\tau = n$.

While our submission fixes $\tau = n$, a tag for NAE schemes can be truncated if needed (not for MRAE), at the cost of decreased security against forgery. See Section 4.

NAE and MRAE families. Remus has two families, Remus-N and Remus-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 [39]. The name Remus stands for the set of two families.

2.3 Recommended Parameter Sets

We present our members (parameters) in Table 2.1. The primary member of our submission is Remus-N1. All members conform to the requirements of NIST call for proposal with respect to key length, nonce length, and maximum input length.

Table 2.1: Members of Remus.

Family	Name	E	ICmode	k	nl	n	d	τ
Remus-N	Remus-N1	Skinny-128/128	ICE1	128	128	128	128	128
	Remus-N2	Skinny-128/128	ICE2	128	128	128	128	128
	Remus-N3	Skinny-64/128	ICE3	128	96	64	120	64
Remus-M	Remus-M1	Skinny-128/128	ICE1	128	128	128	128	128
	Remus-M2	Skinny-128/128	ICE2	128	128	128	128	128

2.4 The Tweakable Block Cipher Skinny

In this section, we will recall the Skinny family of tweakable block ciphers [3]. In our submission, we will use three members of the family: Skinny-64/128, Skinny-128/128.

Skinny Versions.

The lightweight block ciphers of the Skinny family have 64-bit and 128-bit block versions. The internal state is viewed as a 4×4 square array of cells, where each cell is either a nibble (when $n = 64$) or a byte (when $n = 128$). We denote s the bit size of a cell and $IS_{i,j}$ 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 [21] 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 $t = n$ 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 $TK1$ when $z = 1$, $TK1$, $TK2$ 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 s-bit values (either nibbles or 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 s-bit cells. The initialization of the cipher's tweakey state is performed by simply setting for $0 \leq i \leq 15$: $TK1_i = tk_i$ when $z = 1$, $TK1_i = tk_i$, $TK2_i = tk_{16+i}$ 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-64/128 uses 36 rounds, while Skinny-128/128 has 40 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.

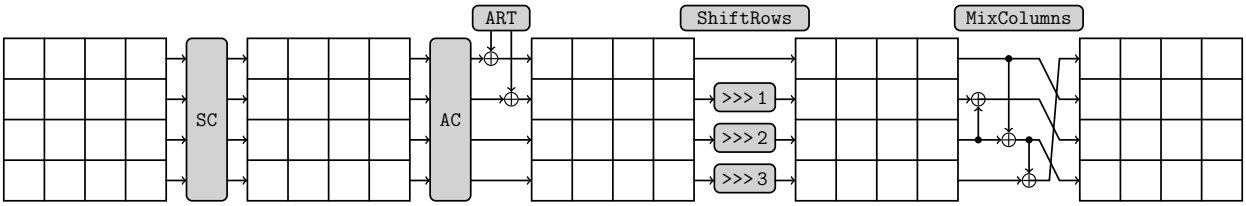


Figure 2.1: The Skinny round function applies five different transformations: `SubCells` (`SC`), `AddConstants` (`AC`), `AddRoundTweakey` (`ART`), `ShiftRows` (`SR`) and `MixColumns` (`MC`).

`SubCells`. A s-bit Sbox is applied to every cell of the cipher internal state. For $s = 4$, Skinny cipher uses a Sbox $\mathcal{S}_4$. The action of this Sbox in hexadecimal notation is given by the following Table 2.2.

Table 2.2: 4-bit Sbox $\mathcal{S}_4$ used in Skinny when $s = 4$.

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$\mathcal{S}_4[x]$	c	6	9	0	1	a	2	b	3	8	5	d	4	e	7	f
$\mathcal{S}_4^{-1}[x]$	3	4	6	8	c	a	1	e	9	2	5	7	0	b	d	f

Note that $\mathcal{S}_4$ can also be described with four NOR and four XOR operations, as depicted in Figure 2.2. If x_0, x_1, x_2 and x_3 represent the four inputs bits of the Sbox (x_0 being the least significant bit), one simply applies the following transformation:

$$(x_3, x_2, x_1, x_0) \rightarrow (x_3, x_2, x_1, x_0 \oplus (\overline{x_3 \vee x_2})),$$

followed by a left shift bit rotation. This process is repeated four times, except for the last iteration where the bit rotation is omitted.

For the case $s = 8$, Skinny uses an 8-bit Sbox $\mathcal{S}_8$ that is built in a similar manner as for the 4-bit Sbox $\mathcal{S}_4$ described above. The construction is simple and is depicted in Figure 2.3. If $x_0, \ldots, x_7$ 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:

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

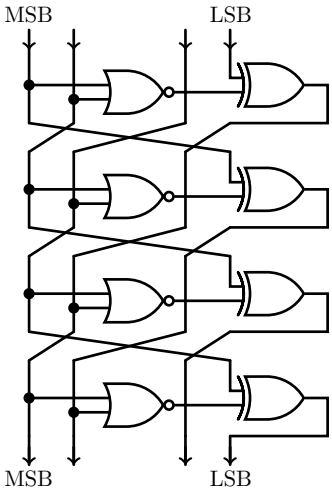


Figure 2.2: Construction of the Sbox $\mathcal{S}_4$.

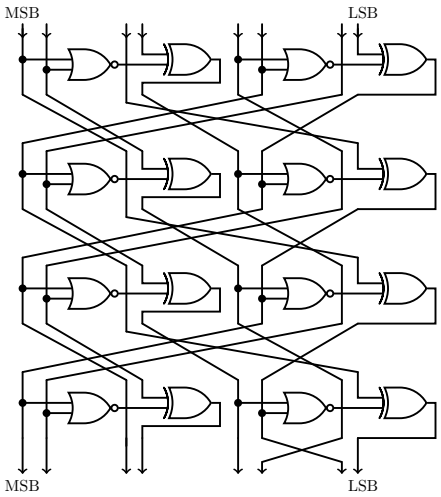


Figure 2.3: Construction of the Sbox $\mathcal{S}_8$.

Table 2.3: 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
};
```

repeating this process four times, except for the last iteration where there is just a bit swap between x_1 and x_2. Besides, we provide in Table 2.3 the table of Sbox $\mathcal{S}_8$ in hexadecimal notations.

`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:

- $IS_{i,j} = IS_{i,j} \oplus TK1_{i,j}$ when $z = 1$,
- $IS_{i,j} = IS_{i,j} \oplus TK1_{i,j} \oplus TK2_{i,j}$ when $z = 2$.
- $IS_{i,j} = IS_{i,j} \oplus TK1_{i,j} \oplus TK2_{i,j} \oplus TK3_{i,j}$ when $z = 3$.

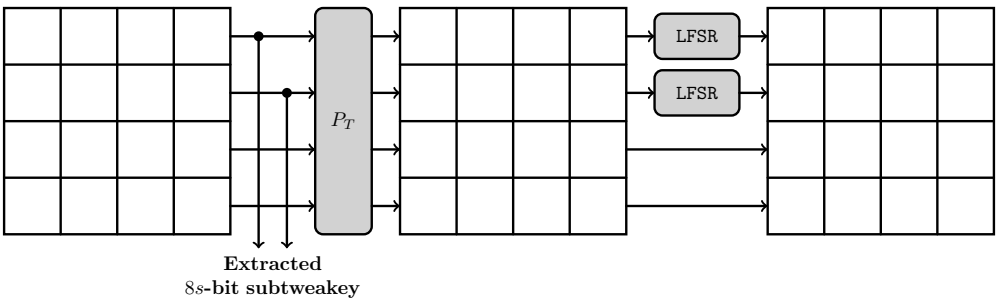


Figure 2.4: The tweakey schedule in Skinny. Each tweakey word $TK1$, $TK2$ and $TK3$ (if any) follows a similar transformation update, except that no LFSR is applied to $TK1$.

Then, the tweakey arrays are updated as follows (this tweakey schedule is illustrated in Figure 2.4). 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 2.4 (x_0 stands for the LSB of the cell).

Table 2.4: The LFSRs used in Skinny to generate the round constants. The TK parameter gives the number of tweakey words in the cipher, and the s parameter gives the size of cell in bits.

TK	s	**LFSR**
$TK2$	4	$(x_3\|\|x_2\|\|x_1\|\|x_0) \rightarrow (x_2\|\|x_1\|\|x_0\|\|x_3 \oplus x_2)$
	8	$(x_7\|\|x_6\|\|x_5\|\|x_4\|\|x_3\|\|x_2\|\|x_1\|\|x_0) \rightarrow (x_6\|\|x_5\|\|x_4\|\|x_3\|\|x_2\|\|x_1\|\|x_0\|\|x_7 \oplus x_5)$
$TK3$	4	$(x_3\|\|x_2\|\|x_1\|\|x_0) \rightarrow (x_0 \oplus x_3\|\|x_3\|\|x_2\|\|x_1)$
	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-64/128 and Skinny-128/128 are provided below.

```
/* Skinny-64-128 */
Key:        9eb93640d088da63
            76a39d1c8bea71e1
Plaintext:  cf16cfe8fd0f98aa
Ciphertext: 6ceda1f43de92b9e

/* Skinny-128-128 */
Key:        4f55cfb0520cac52fd92c15f37073e93
Plaintext:  f20adb0eb08b648a3b2eeed1f0adda14
Ciphertext: 22ff30d498ea62d7e45b476e33675b74
```

2.5 The Authenticated Encryption Remus

2.5.1 Block Counters and Domain Separation

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 Remus. 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 2.5). Then, we have the following:

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

  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 0.
- b_3 is set to 1 once we have handled the last block of data (AD and message chains are treated separately), to 0 otherwise.
- b_2 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_1 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_0 is set to 1 when we are handling a padded block (associated data or message), 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.

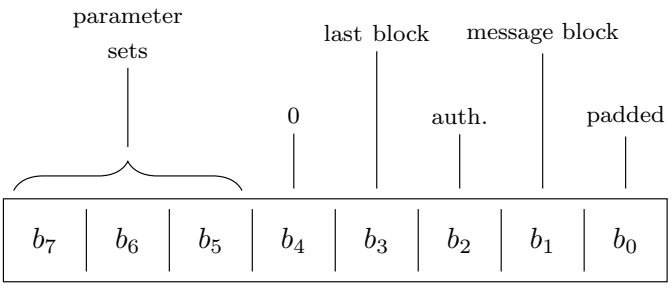


Figure 2.5: Domain separation when using the tweakable block cipher

Doubling over a Finite Field. For any positive integer c, we assume GF(2^c) is defined over the lexicographically-first polynomial among the irreducible degree c polynomials of a minimum number of coefficients. We use two fields: GF(2^c) for $c \in \{128, 120\}$. The primitive polynomials are:

$$\mathtt{x}^{128} + \mathtt{x}^7 + \mathtt{x}^2 + \mathtt{x} + 1 \text{ for } c = 128,$$
$$\mathtt{x}^{120} + \mathtt{x}^4 + \mathtt{x}^3 + \mathtt{x} + 1 \text{ for } c = 120.$$

Let $Z = (z_{c-1}z_{c-2}\dots z_1z_0)$ for $z_i \in \{0,1\}$, $i \in [\![c]\!]_0$ be an element of GF(2^c). A multiplication of Z by the generator (polynomial $\mathtt{x}$) is called *doubling* and written as $2Z$ [37]. An i-times doubling

of Z is written as 2^iZ, and is efficiently computed from $2^{i-1}Z$ (see below). Here, $2^0Z = Z$ for any Z. When $Z = 0^n$, *i.e.*, zero entity in the field, then $2^iZ = 0^n$ for any $i \geq 0$.

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 a doubling-based counter, *i.e.*, 2^DX for some key-dependent variable X. One can interpret $\overline{D}$ as 2^D (but in that case it is a part of tweakey state or a coefficient of mask, and *not* a part of input of ICE).

On bit-level, doubling $Z \to 2Z$ over $\mathrm{GF}(2^c)$ for $c = 128$ is defined as

$$\begin{aligned}
&z_i \leftarrow z_{i-1} \text{ for } i \in [\![128]\!]_0 \setminus \{7, 2, 1, 0\},\\
&z_7 \leftarrow z_6 \oplus z_{127},\\
&z_2 \leftarrow z_1 \oplus z_{127},\\
&z_1 \leftarrow z_0 \oplus z_{127},\\
&z_0 \leftarrow z_{127}.
\end{aligned}$$

Similarly for $\mathrm{GF}(2^{120})$, we have

$$\begin{aligned}
&z_i \leftarrow z_{i-1} \text{ for } i \in [\![120]\!]_0 \setminus \{4, 3, 1, 0\},\\
&z_4 \leftarrow z_3 \oplus z_{119},\\
&z_3 \leftarrow z_2 \oplus z_{119},\\
&z_1 \leftarrow z_0 \oplus z_{119},\\
&z_0 \leftarrow z_{119}.
\end{aligned}$$

2.5.2 The TBC ICE

In the specification of Remus, Skinny is not directly used as a TBC. Instead, we use Skinny : $\mathcal{K} \times \mathcal{M} \to \mathcal{M}$ as a building block (as a block cipher rather than TBC) to build another TBC : $\mathcal{K} \times \mathcal{T} \times \mathcal{M} \to \mathcal{M}$, where $\mathcal{K} = \{0,1\}^k$ is the key space, $\mathcal{M} = \{0,1\}^n$ is the message space, and $\mathcal{T} = \mathcal{N} \times \mathcal{D} \times \mathcal{B}$ is the tweak space. The tweak space $\mathcal{T}$ consists of the nonce space $\mathcal{N} = \{0,1\}^{nl}$, the counter space $\mathcal{D} = [\![2^d - 1]\!]$, and the domain separation byte $\mathcal{B} = [\![256]\!]_0$ as described in Section 2.5.1. We call this TBC ICE (for Ideal-Cipher Encryption). There are 3 variants, ICE1, ICE2 and ICE3.

Each variant consists of two main components, the key derivation function $\mathsf{KDF} : \mathcal{K} \times \mathcal{N} \to \mathcal{L} \times \mathcal{V}$, and the "core" encryption function $\mathsf{ICEnc} : (\mathcal{L} \times \mathcal{V}) \times (\mathcal{D} \times \mathcal{B}) \times \mathcal{M} \to \mathcal{M}$. Here, $\mathcal{L} = \mathcal{K} = \{0,1\}^k$ and $\mathcal{V} = \mathcal{M} = \{0,1\}^n$. The algorithm of ICEnc is as shown in Figure 2.6 for all variants. In addition, there is a tweakey encode function $\mathtt{encode} : \mathcal{L} \times \mathcal{D} \times \mathcal{B} \to \mathcal{K}$ inside ICEnc. For convenience, KDF for ICE1 may also be referred as KDF1. KDF2 and KDF3 are defined analogously.

An encryption of ICE is performed as follows. Given a tweak $T = (N, D, B) \in \mathcal{T}$, key $K \in \mathcal{K}$, and plaintext $M \in \mathcal{M}$, first, $\mathsf{KDF}(N, K) \to (L, V)$ derives the nonce-dependent mask values (L, V), and then ICEnc encrypts M as $\mathsf{ICEnc}(L, V, D, B, M) \to C$, using the key of the internal E determined by $\mathtt{encode}(L, D, B)$. Here, $\mathsf{ICEnc}(L, V, D, B, *)$ is a permutation over $\mathcal{M}$ for any (L, V, D, B).

Each variant is defined as follows. The matrix G is defined at Section 2.5.3. For all variants, $k = 128$.

1. ICE1: $n = 128$, $nl = 128$, $d = 128$ and it uses Skinny-128/128 as its building block E.
   (a) $\mathsf{KDF}(N, K) = (L, V)$ where $L = G(E_K(N))$, $V = 0^n$.
   (b) $\mathtt{encode}(L, D, B) = 2^DL \oplus (0^{120} \,\|\, B)$.
2. ICE2: $n = 128$, $nl = 128$, $d = 128$ and it uses Skinny-128/128 as its building block E.

Algorithm $\mathsf{ICEnc}^{D,B}_{L,V}(M)$

1. $S \leftarrow 2^D V \oplus M$
2. $T_K \leftarrow \mathtt{encode}(L, D, B)$
3. $S \leftarrow E_{T_K}(S)$
4. $C \leftarrow 2^D V \oplus S$
5. **return** C

Figure 2.6: Definition of ICEnc, the core encryption routine of ICE. ICEnc is common to all three variants of ICE, ICE1 and ICE2 and ICE3 except the definition of `encode`. Note that ICE1 and ICE3 fix $V = 0^n$, hence effectively $S \leftarrow M$ (line 1) and $C \leftarrow S$ (line 4). Variables L and V are assumed to be derived from the corresponding KDF taking (N, K), as a pre-processing.

   (a) $\mathsf{KDF}(N, K) = (L, V)$ where $L' = E_K(N), V' = E_{K\oplus 1}(L')$, and $L = G(L')$ and $V = G(V')$.

   (b) $\mathtt{encode}(L, D, B) = 2^D L \oplus (0^{120} \,\|\, B)$.

3. ICE3: $n = 64$, $nl = 96$, $d = 120$ and it uses Skinny-64/128 as its building block E.

   (a) $\mathsf{KDF}(N, K) = (L, V)$ where $L \leftarrow (N \,\|\, 0^{32}) \oplus K, V \leftarrow 0^n$.

   (b) $\mathtt{encode}(L, D, B) = (2^D L[1]) \,\|\, (B \oplus L[2])$, where $(L[1], L[2]) \xleftarrow{120} L$ and the multiplication is over $\mathrm{GF}(2^{120})$ and applied to $L[1]$. Note that $|L[1]| = 120$ and $|L[2]| = 8$.

Note that ICE1 and ICE2 are only different in the second mask V derived by their KDFs.

When ICE is working inside Remus, the corresponding KDF is performed only once as an initialization. For ICE1 or ICE2, KDF involves one or two calls of E and matrix multiplications by G (see above). For ICE3, KDF is just a linear operation of (N, K). For each input block, ICE applies doubling to the derived mask values. Since doubling is a sequential operation, computing $\mathsf{ICEnc}^{D+1,B}_{L,V}(M)$ after $\mathsf{ICEnc}^{D,B'}_{L,V}(M')$ is easy and does not need any additional memory.

2.5.3 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

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

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 $(X[0], \ldots, X[n/8-1]) \xleftarrow{8} X$ and $(X[i][0], \ldots, X[i][7]) \xleftarrow{1} X[i]$, 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.

2.5.4 Remus-N nonce-based AE mode

The specification of Remus-N is shown in Figure 2.7. Figures 2.8, 2.9, 2.10 show encryption of Remus-N. For completeness, the definition of ρ is also included.

2.5.5 Remus-M misuse-resistant AE mode

The specification of Remus-M is shown in Figure 2.11. Figures 2.12, 2.13 show encryption of Remus-M. For completeness, the definition of ρ is also included.

Algorithm $\mathsf{Remus\text{-}N.Enc}_K(N, A, M)$

1. $(L, V) \leftarrow \mathsf{KDF}(N, K)$
2. $S \leftarrow 0^n$
3. $(A[1], \ldots, A[a]) \xleftarrow{n} A$
4. $(M[1], \ldots, M[m]) \xleftarrow{n} M$
5. **if** $|A[a]| < n$ **then** $w_A \leftarrow 13$ **else** 12
6. **if** $|M[m]| < n$ **then** $w_M \leftarrow 11$ **else** 10
7. $A[a] \leftarrow \mathtt{pad}_n(A[a])$
8. **for** $i = 1$ **to** $a - 1$
9. $\quad (S, \eta) \leftarrow \rho(S, A[i])$
10. $\quad S \leftarrow \mathsf{ICEnc}^{\bar{i},4}_{L,V}(S)$
11. **end for**
12. $(S, \eta) \leftarrow \rho(S, A[a])$
13. $S \leftarrow \mathsf{ICEnc}^{\bar{a},w_A}_{L,V}(S)$
14. **for** $i = 1$ **to** $m - 1$
15. $\quad (S, C[i]) \leftarrow \rho(S, M[i])$
16. $\quad S \leftarrow \mathsf{ICEnc}^{\overline{a+i},2}_{L,V}(S)$
17. **end for**
18. $M'[m] \leftarrow \mathtt{pad}_n(M[m])$
19. $(S, C'[m]) \leftarrow \rho(S, M'[m])$
20. $S \leftarrow \mathsf{ICEnc}^{\overline{a+m},w_M}_{L,V}(S)$
21. $C[m] \leftarrow \mathtt{lsb}_{|M[m]|}(C'[m])$
22. $(\eta, T) \leftarrow \rho(S, 0^n)$
23. $C \leftarrow C[1] \,\|\, C[2] \,\|\, \ldots \,\|\, C[m]$
24. **return** (C, T)

Algorithm $\mathsf{Remus\text{-}N.Dec}_K(N, A, C, T)$

1. $(L, V) \leftarrow \mathsf{KDF}(N, K)$
2. $S \leftarrow 0^n$
3. $(A[1], \ldots, A[a]) \xleftarrow{n} A$
4. $(C[1], \ldots, C[m]) \xleftarrow{n} C$
5. **if** $|A[a]| < n$ **then** $w_A \leftarrow 13$ **else** 12
6. **if** $|C[m]| < n$ **then** $w_C \leftarrow 11$ **else** 10
7. $A[a] \leftarrow \mathtt{pad}_n(A[a])$
8. **for** $i = 1$ **to** $a - 1$
9. $\quad (S, \eta) \leftarrow \rho(S, A[i])$
10. $\quad S \leftarrow \mathsf{ICEnc}^{\bar{i},4}_{L,V}(S)$
11. **end for**
12. $(S, \eta) \leftarrow \rho(S, A[a])$
13. $S \leftarrow \mathsf{ICEnc}^{\bar{a},w_A}_{L,V}(S)$
14. **for** $i = 1$ **to** $m - 1$
15. $\quad (S, M[i]) \leftarrow \rho^{-1}(S, C[i])$
16. $\quad S \leftarrow \mathsf{ICEnc}^{\overline{a+i},2}_{L,V}(S)$
17. **end for**
18. $\widetilde{S} \leftarrow (0^{|C[m]|} \,\|\, \mathtt{msb}_{n-|C[m]|}(G(S)))$
19. $C'[m] \leftarrow \mathtt{pad}_n(C[m]) \oplus \widetilde{S}$
20. $(S, M'[m]) \leftarrow \rho^{-1}(S, C'[m])$
21. $M[m] \leftarrow \mathtt{lsb}_{|C[m]|}(M'[m])$
22. $S \leftarrow \mathsf{ICEnc}^{\overline{a+m},w_C}_{L,V}(S)$
23. $(\eta, T^*) \leftarrow \rho(S, 0^n)$
24. $M \leftarrow M[1] \,\|\, M[2] \,\|\, \ldots \,\|\, M[m]$
25. **if** $T^* = T$ **then return** M **else** $\perp$

Algorithm $\rho(S, M)$

1. $C \leftarrow M \oplus G(S)$
2. $S' \leftarrow S \oplus M$
3. **return** (S', C)

Algorithm $\rho^{-1}(S, C)$

1. $M \leftarrow C \oplus G(S)$
2. $S' \leftarrow S \oplus M$
3. **return** (S', M)

Figure 2.7: Encryption and decryption of Remus-N. It uses TBC ICE consisting of KDF and ICEnc. 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. Remus-N1 is used as a working example. For other Remus-N versions, the values of the bits b_7 and b_6 in the domain separation need to be adapted accordingly, alongside with using the appropriate ICE variants.

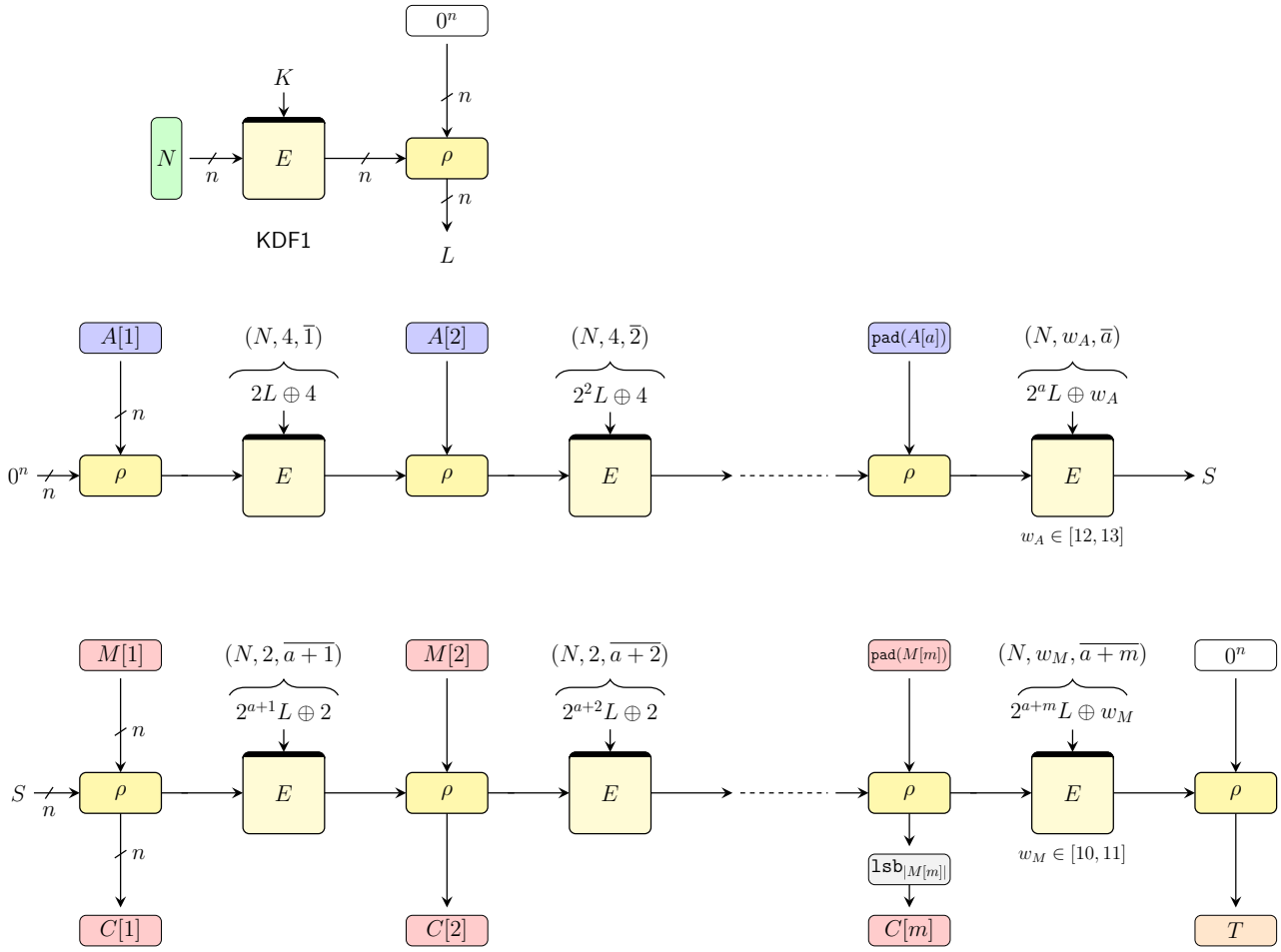


Figure 2.8: Remus-N with ICE1 (Remus-N1). (Top) Key derivation. (Middle) Processing of AD (Bottom) Encryption. The domain separation B being of 8 bits only, $\oplus B$ is to be interpreted as $\oplus\, 0^{120}||B$.

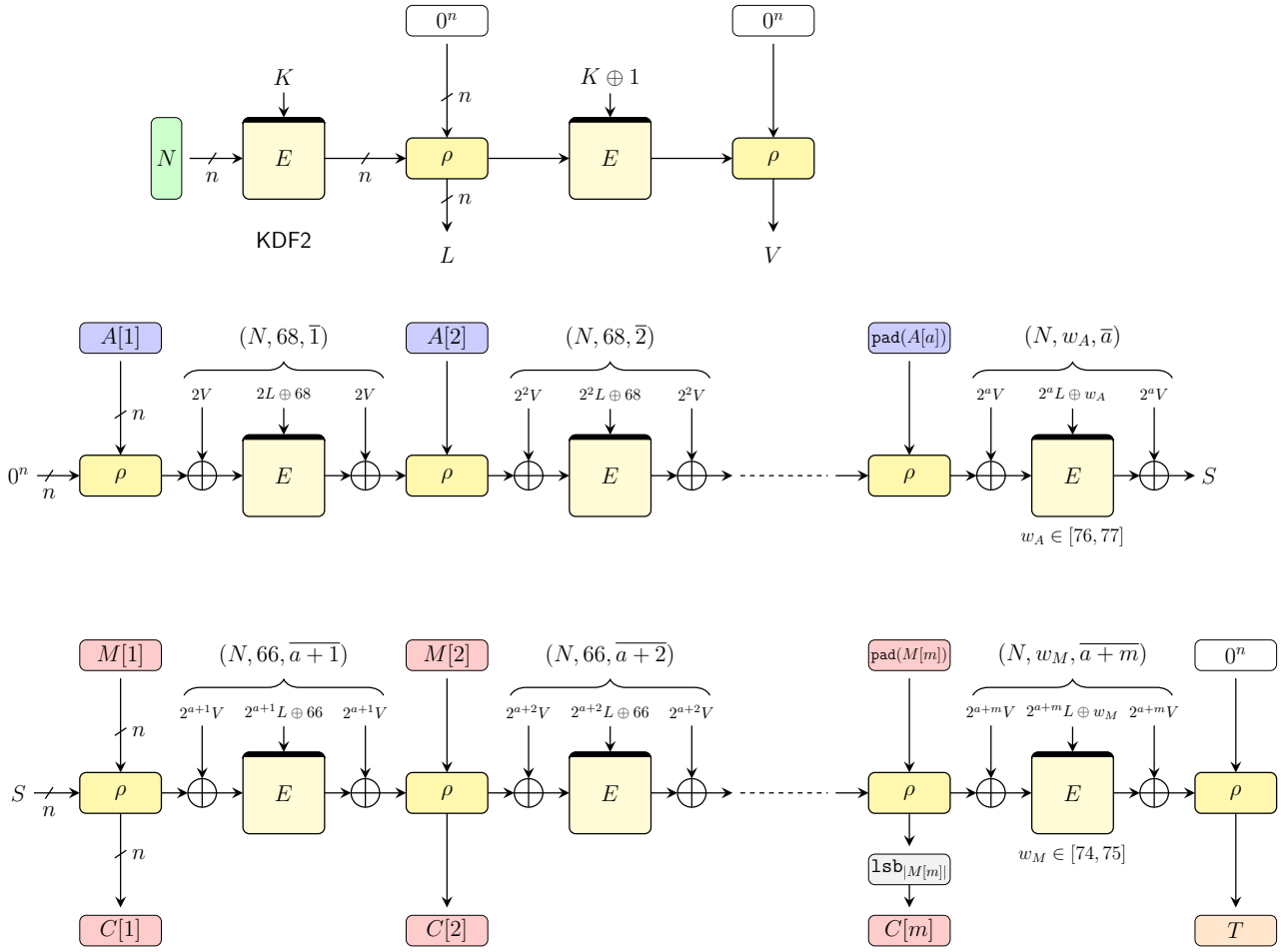


Figure 2.9: Remus-N with ICE2 (Remus-N2). (Top) Key derivation. (Middle) Processing of AD (Bottom) Encryption. The domain separation B being of 8 bits only, $\oplus B$ is to be interpreted as $\oplus\, 0^{120}||B$.

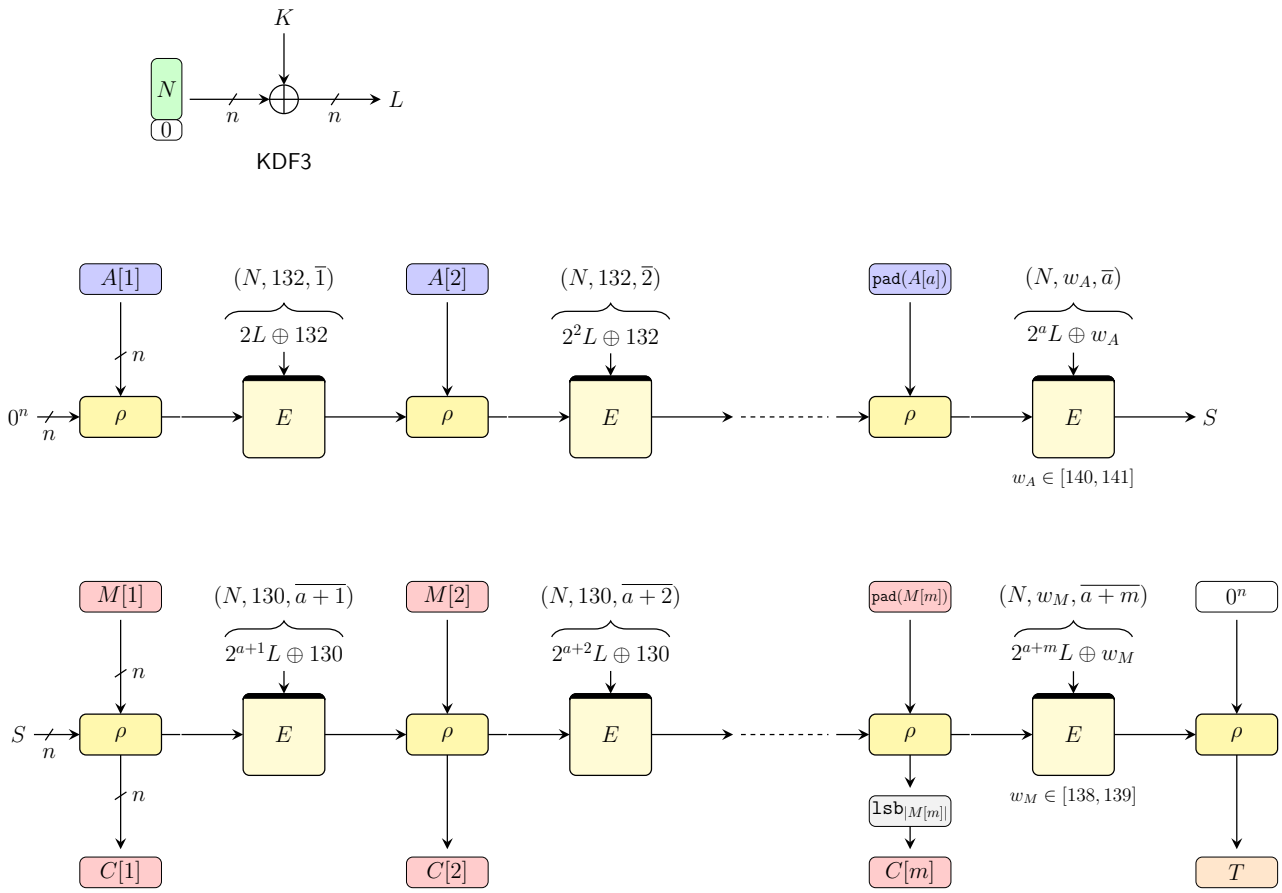


Figure 2.10: Remus-N with ICE3 (Remus-N3). (Top) Key derivation (computing $(2^D L[1])||(B \oplus L[2])$). (Middle) Processing of AD (Bottom) Encryption. The domain separation B being of 8 bits only, $\oplus B$ is to be interpreted as $\oplus\, 0^{120}||B$.

Algorithm $\mathsf{Remus\text{-}M.Enc}_K(N, A, M)$

1. $(L, V) \leftarrow \mathsf{KDF}(N, K)$
2. $S \leftarrow 0^n$
3. $(A[1], \ldots, A[a]) \xleftarrow{n} A$
4. $(M[1], \ldots, M[m]) \xleftarrow{n} M$
5. **if** $|A[a]| < n$ **then** $w_A \leftarrow 45$ **else** 44
6. **if** $|M[m]| < n$ **then** $w_M \leftarrow 47$ **else** 46
7. $A[a] \leftarrow \mathtt{pad}_n(A[a])$
8. **for** $i = 1$ **to** $a - 1$
9. $\quad (S, \eta) \leftarrow \rho(S, A[i])$
10. $\quad S \leftarrow \mathsf{ICEnc}_{L,V}^{\bar{i},36}(S)$
11. **end for**
12. $(S, \eta) \leftarrow \rho(S, A[a])$
13. $S \leftarrow \mathsf{ICEnc}_{L,V}^{\bar{a},w_A}(S)$
14. **for** $i = 1$ **to** $m - 1$
15. $\quad (S, \eta) \leftarrow \rho(S, M[i])$
16. $\quad S \leftarrow \mathsf{ICEnc}_{L,V}^{\overline{a+i},38}(S)$
17. **end for**
18. $M'[m] \leftarrow \mathtt{pad}_n(M[m])$
19. $(S, \eta) \leftarrow \rho(S, M'[m])$
20. $S \leftarrow \mathsf{ICEnc}_{L,V}^{\overline{a+m},w_M}(S)$
21. $(\eta, T) \leftarrow \rho(S, 0^n)$
22. **if** $M = \epsilon$ **then return** (ϵ, T)
23. $S \leftarrow T$
24. **for** $i = 1$ **to** $\underline{m} - 1$
25. $\quad S \leftarrow \mathsf{ICEnc}_{L,V}^{\overline{i-1},34}(S)$
26. $\quad (S, C[i]) \leftarrow \rho(S, M[i])$
27. **end for**
28. $S \leftarrow \mathsf{ICEnc}_{L,V}^{\overline{m-1},34}(S)$
29. $(\eta, C'[m]) \leftarrow \rho(S, M'[m])$
30. $C[m] \leftarrow \mathtt{lsb}_{|M[m]|}(C'[m])$
31. $C \leftarrow C[1] \,\|\, C[2] \,\|\, \ldots \,\|\, C[m]$
32. **return** (C, T)

Algorithm $\mathsf{Remus\text{-}M.Dec}_K(N, A, C, T)$

1. $(L, V) \leftarrow \mathsf{KDF}(N, K)$
2. **if** $C = \epsilon$ **then** $M \leftarrow \epsilon$
3. **else**
4. $\quad S \leftarrow T$
5. $\quad (C[1], \ldots, C[m]) \xleftarrow{n} C$
6. $\quad z \leftarrow |C[m]|$
7. $\quad C[m] \leftarrow \mathtt{pad}_n(C[m])$
8. $\quad$ **for** $i = 1$ **to** $\underline{m}$
9. $\quad\quad S \leftarrow \mathsf{ICEnc}_{L,V}^{\overline{i-1},34}(S)$
10. $\quad\quad (S, M[i]) \leftarrow \rho^{-1}(S, C[i])$
11. $\quad$ **end for**
12. $\quad M[m] \leftarrow \mathtt{lsb}_z(M[m])$
13. $\quad M \leftarrow M[1] \,\|\, \ldots \,\|\, M[m]$
14. $S \leftarrow 0^n$
15. $(A[1], \ldots, A[a]) \xleftarrow{n} A$
16. **if** $|A[a]| < n$ **then** $w_A \leftarrow 45$ **else** 44
17. **if** $|M[m]| < n$ **then** $w_M \leftarrow 47$ **else** 46
18. $A[a] \leftarrow \mathtt{pad}_n(A[a])$
19. **for** $i = 1$ **to** $a - 1$
20. $\quad (S, \eta) \leftarrow \rho(S, A[i])$
21. $\quad S \leftarrow \mathsf{ICEnc}_{L,V}^{\bar{i},36}(S)$
22. **end for**
23. $(S, \eta) \leftarrow \rho(S, A[a])$
24. $S \leftarrow \mathsf{ICEnc}_{L,V}^{\bar{a},w_A}(S)$
25. **for** $i = 1$ **to** $m - 1$
26. $\quad (S, \eta) \leftarrow \rho(S, M[i])$
27. $\quad S \leftarrow \mathsf{ICEnc}_{L,V}^{\overline{a+i},38}(S)$
28. **end for**
29. $M'[m] \leftarrow \mathtt{pad}_n(M[m])$
30. $(S, \eta) \leftarrow \rho(S, M'[m])$
31. $S \leftarrow \mathsf{ICEnc}_{L,V}^{\overline{a+m},w_M}(S)$
32. $(\eta, T^*) \leftarrow \rho(S, 0^n)$
33. **if** $T^* = T$ **then return** M **else** $\perp$

Algorithm $\rho(S, M)$

1. $C \leftarrow M \oplus G(S)$
2. $S' \leftarrow S \oplus M$
3. **return** (S', C)

Algorithm $\rho^{-1}(S, C)$

1. $M \leftarrow C \oplus G(S)$
2. $S' \leftarrow S \oplus M$
3. **return** (S', M)

Figure 2.11: Encryption and decryption of Remus-M. It uses TBC ICE consisting of KDF and ICEnc. Remus-M1 is used as a working example.

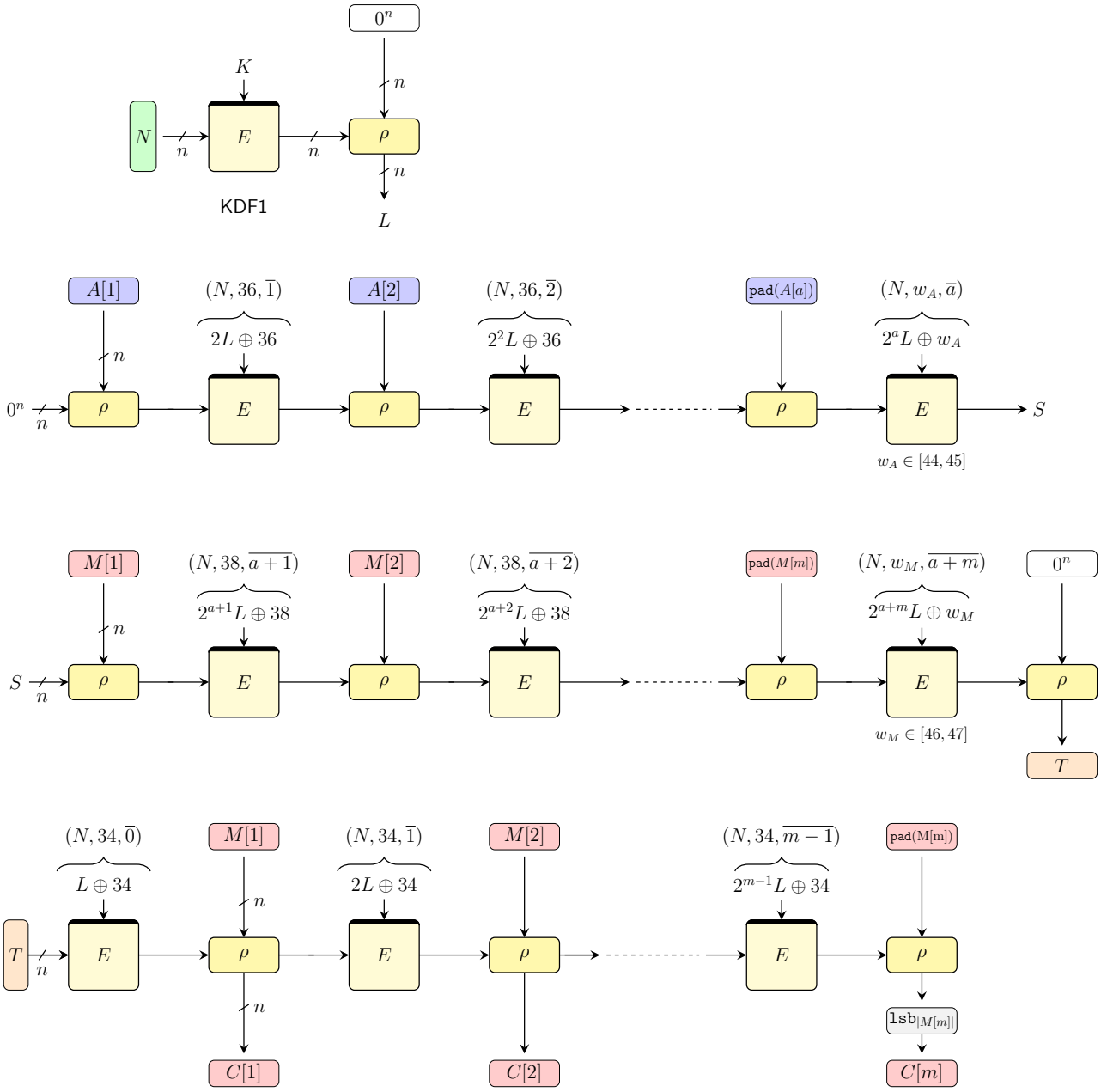


Figure 2.12: Remus-M with ICE1 (Remus-M1). (Top) Key derivation. (Middle-Top) Processing of AD (Middle-Bottom) Processing of M authentication (Bottom) Encryption. The domain separation B being of 8 bits only, $\oplus B$ is to be interpreted as $\oplus\, 0^{120}||B$.

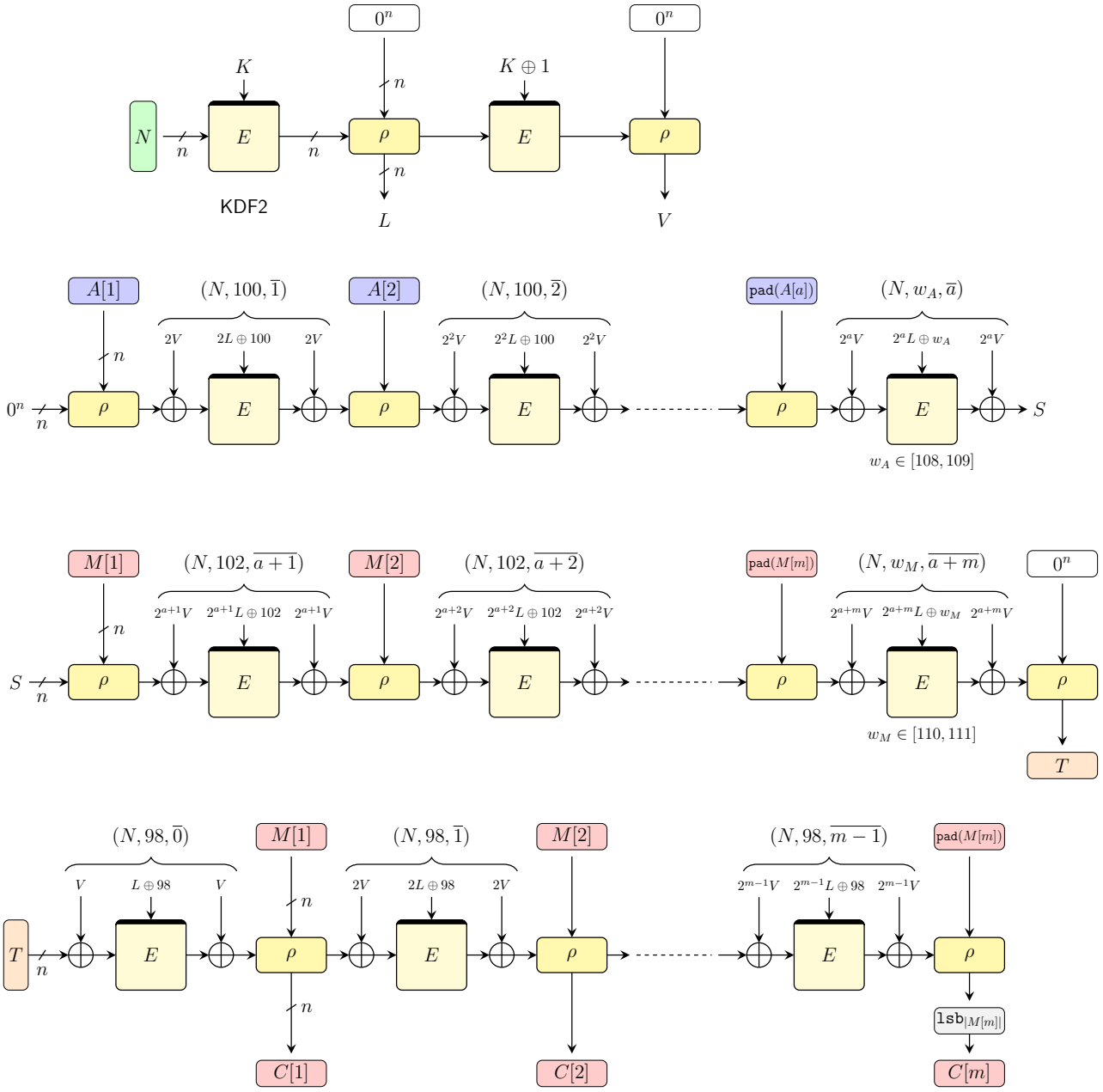


Figure 2.13: Remus-M with ICE2 (Remus-M2). (Top) Key derivation. (Middle-Top) Processing of AD (Middle-Bottom) Processing of M authentication (Bottom) Encryption. The domain separation B being of 8 bits only, $\oplus B$ is to be interpreted as $\oplus\, 0^{120}||B$.

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 online and offline query complexities (corresponding to data complexity and time complexity), of an adversary to break the cipher, in logarithm base 2. In more detail, an integer x in the table means an attack possibly breaks the scheme with online query complexity Q_{online} and offline query complexity Q_{offline} if $\max\{Q_{\text{online}}, Q_{\text{offline}}\} = 2^x$. 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 under the Ideal-Cipher Model (ICM), that is, the model that assumes Skinny is sampled uniformly over all the ciphers (see Section 4 for the definition). Intuitively, this corresponds to model that Skinny behaves ideally, though ICM cannot be instantiated (as a key of E is also a part of queries to ICM, and outputs must be random). ICM has been acknowledged as a meaningful security proof model, especially in the field of hash function constructions.

As described in Introduction, we can also obtain standard model proofs for Remus, by assuming the intermediate TBC ICE as a keyed primitive called tweakable pseudorandom permutation (TPRP). The standard model proofs are actually a part of ICM proofs. Specifically, the security bounds under this standard model will appear in the hybrid argument of ICM proofs. A similar technique appeared in some permutation-based schemes [13, 31, 34], where the keyed primitive is a variant of Even-Mansour cipher. We warn that assuming ICE as TPRP does not imply its perfect security: there are generic attacks which work even if Skinny is an ideal-cipher. Therefore, the expected bit security levels are identical for both ICM and this standard model analyses. Therefore, Section 4 only presents the security bounds under ICM.

For Remus-N1, Table 3.1 shows $n/2$-bit security for both privacy and authenticity against NR adversary. For Remus-N2, Table 3.1 shows full n-bit security for both privacy and authenticity against NR adversary. For Remus-N3, Table 3.1 shows $(n/2 - 4)$-bit security for both privacy and authenticity against NR adversary, where -4 comes from the $(n - 8)$-bit counter state (bit security is a half of it).

[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.

For Remus-M1, Table 3.1 shows $n/2$-bit security for both privacy and authenticity against NR and NM adversaries. For Remus-M2, Table 3.1 shows n-bit security for both privacy and authenticity against NR adversary and in addition, $n/2$-bit security for both 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 Remus-M2 show the notable property of graceful security degradation with respect to the number of nonce repetition [35]. 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: Security claims of Remus. NR denotes Nonce-Respecting adversary and NM denotes Nonce-Misusing adversary.

Parameter	NR-Priv	NR-Auth	NM-Priv	NM-Auth
Remus-N1	64	64	–	–
Remus-N2	128	128	–	–
Remus-N3	60	60	–	–
Remus-M1	64	64	64	64
Remus-M2	128	128	$64 \sim 128$	$64 \sim 128$

Key Recovery Security. For key recovery, the adversary needs to find the $k = 128$-bit key used in KDF with 2^k offline queries (computations). See Table 3.2. Therefore, all members of Remus have $k = 128$-bit security against key recovery under the single-key setting.

Table 3.2: Security claims of Remus against key recovery.

Parameter	Key Recovery
Remus-N1	128
Remus-N2	128
Remus-N3	128
Remus-M1	128
Remus-M2	128

Security Analysis

4.1 Security Notions

Security Notions for NAE. We consider the standard security notions for nonce-based AE [6, 7, 38]. 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, $\perp$. We assume Π is based on the ideal block cipher $E : \mathcal{K} \times \mathcal{M} \to \mathcal{M}$, which is uniformly distributed over all block ciphers of key space $\mathcal{K}$ and message space $\mathcal{M}$, and we allow the adversary to query E while attacking Π. Specifically, the adversary can query $Y \leftarrow E(K', X)$ for any $(K', X) \in \mathcal{K} \times \mathcal{M}$ or $X \leftarrow E^{-1}(K', Y)$ for any $(K', Y) \in \mathcal{K} \times \mathcal{M}$. We remark that K' here is a part of query and not the secret key $K \xleftarrow{\$} \mathcal{K}$. Such a query is called an offline query or a primitive query. In contrast, a query to $\Pi.\mathcal{E}_K$ or $\Pi.\mathcal{D}_K$ is called an online query or a construction query.

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), with access to the ideal cipher E for both worlds (\$ oracle and E oracle are independent). The adversary is assumed to be nonce-respecting. We define the privacy advantage as

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

which measures the hardness of breaking the 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. As in the case of the privacy notion, the adversary has access to E. We define the authenticity advantage as

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

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 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

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

and the authenticity advantage as

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

We note that while adversaries can repeat nonces, we without loss of generality assume that they do not repeat the same query. See also [39] for reference.

4.2 Security of Remus-N

For $A \in \{0,1\}^*$, we say A has a AD blocks if $|A|_n = a$. Similarly for plaintext $M \in \{0,1\}^*$ we say M has m message blocks if $|M|_n = m$. The same holds for the 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 $a + m$, which is called the number of *effective blocks* of a query.

Let $\mathcal{A}$ be a nonce-respecting adversary against Remus-N using q_c encryption (online/construction) queries and q_p offline/primitive queries with total number of effective blocks in encryption queries σ_{priv}. We have the following privacy bounds:

$$\begin{aligned}
\mathbf{Adv}^{\mathtt{priv}}_{\mathsf{Remus\text{-}N1}}(\mathcal{A}) &\le \frac{9\sigma_{\mathrm{priv}}^2 + 4\sigma_{\mathrm{priv}} \cdot q_p}{2^n} + \frac{2q_p}{2^n},\\
\mathbf{Adv}^{\mathtt{priv}}_{\mathsf{Remus\text{-}N2}}(\mathcal{A}) &\le \frac{9\sigma_{\mathrm{priv}}^2 + 4\sigma_{\mathrm{priv}} \cdot q_p}{2^{2n}} + \frac{2q_p}{2^n},\\
\mathbf{Adv}^{\mathtt{priv}}_{\mathsf{Remus\text{-}N3}}(\mathcal{A}) &\le \frac{q_p \cdot \sigma_{\mathrm{priv}}}{2^{k-8}}.
\end{aligned}$$

For authenticity bounds, let $\mathcal{B}$ be a nonce-respecting adversary using q_c encryption queries and q_d decryption queries (both are online/construction queries), with total number of effective blocks for encryption and decryption queries σ_{auth}, and q_p offline/primitive queries. Also we define ℓ as the maximum effective block length of a plaintext among q_c encryption queries. Then we have

$$\begin{aligned}
\mathbf{Adv}^{\mathtt{auth}}_{\mathsf{Remus\text{-}N1}}(\mathcal{B}) &\le \frac{9\sigma_{\mathrm{auth}}^2 + 4\sigma_{\mathrm{auth}} \cdot q_p}{2^n} + \frac{2q_p}{2^n} + \frac{2\ell q_d}{2^n} + \frac{2q_d}{2^\tau},\\
\mathbf{Adv}^{\mathtt{auth}}_{\mathsf{Remus\text{-}N2}}(\mathcal{B}) &\le \frac{9\sigma_{\mathrm{auth}}^2 + 4\sigma_{\mathrm{auth}} \cdot q_p}{2^{2n}} + \frac{2q_p}{2^n} + \frac{2\ell q_d}{2^n} + \frac{2q_d}{2^\tau},\\
\mathbf{Adv}^{\mathtt{auth}}_{\mathsf{Remus\text{-}N3}}(\mathcal{B}) &\le \frac{q_p \cdot \sigma_{\mathrm{auth}}}{2^{k-8}} + \frac{2\ell q_d}{2^n} + \frac{2q_d}{2^\tau}.
\end{aligned}$$

Note that tag length τ is set to n for all members of Remus-N, however, if $1 \le \tau < n$ (which is not a part of our submission), it still maintains n-bit privacy and τ-bit authenticity. While the term $\frac{2\ell q_d}{2^n}$ is present, this term is very likely not tight and can be improved to $\sigma_{\mathrm{dec}}/2^n$ with total effective blocks in decryption queries σ_{dec}.

The security of Remus-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 $X \in \{0,1\}^n$, $G^{(i)}(X) = \mathtt{lsb}_i(G(X))\|0^{n-i}$ for all $i = 0, 8, 16, \ldots, n$; note that all variables are byte strings, and $\mathtt{lsb}_i(X)$ 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 $i = 8, 16, \ldots, n$. The above security bounds hold as long as G is sound. The proofs are similar to those for iCOFB [12]. 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 Remus-M

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 $a + 2m$, which is called the number of *effective blocks* of a query.

Let $\mathcal{A}$ be an adversary against Remus-N using q_c encryption (online/construction) queries and q_p offline/primitive queries with total number of effective blocks in encryption queries σ_{priv}. In the NR case, we have the following privacy bounds:

$$\mathbf{Adv}^{\texttt{priv}}_{\text{Remus-M1}}(\mathcal{A}) \leq \frac{9\sigma_{\text{priv}}^2 + 4\sigma_{\text{priv}} \cdot q_p}{2^n} + \frac{2q_p}{2^n},$$
$$\mathbf{Adv}^{\texttt{priv}}_{\text{Remus-M2}}(\mathcal{A}) \leq \frac{9\sigma_{\text{priv}}^2 + 4\sigma_{\text{priv}} \cdot q_p}{2^{2n}} + \frac{2q_p}{2^n}.$$

In the NM case, we have the following privacy bounds:

$$\mathbf{Adv}^{\texttt{nm-priv}}_{\text{Remus-M1}}(\mathcal{A}) \leq \frac{9\sigma_{\text{priv}}^2 + 4\sigma_{\text{priv}} \cdot q_p}{2^n} + \frac{2q_p}{2^n} + \frac{4r\sigma_{\text{priv}}}{2^n},$$
$$\mathbf{Adv}^{\texttt{nm-priv}}_{\text{Remus-M2}}(\mathcal{A}) \leq \frac{9\sigma_{\text{priv}}^2 + 4\sigma_{\text{priv}} \cdot q_p}{2^{2n}} + \frac{2q_p}{2^n} + \frac{4r\sigma_{\text{priv}}}{2^n}.$$

Here, the adversary can repeat a nonce at most r times.

For authenticity bounds, let $\mathcal{B}$ be an adversary using q_c encryption queries and q_d decryption queries (both are online/construction queries), with total number of effective blocks for encryption and decryption queries σ_{auth}, and q_p offline/primitive queries. Also we define ℓ as the maximum effective block length among all the encryption and decryption queries. Then in the NR case, we have

$$\mathbf{Adv}^{\texttt{auth}}_{\text{Remus-M1}}(\mathcal{B}) \leq \frac{9\sigma_{\text{auth}}^2 + 4\sigma_{\text{auth}} \cdot q_p}{2^n} + \frac{2q_p}{2^n} + \frac{2\ell q_d}{2^n},$$
$$\mathbf{Adv}^{\texttt{auth}}_{\text{Remus-M2}}(\mathcal{B}) \leq \frac{9\sigma_{\text{auth}}^2 + 4\sigma_{\text{auth}} \cdot q_p}{2^{2n}} + \frac{2q_p}{2^n} + \frac{2\ell q_d}{2^n}.$$

In the NM case, we have

$$\mathbf{Adv}^{\texttt{nm-auth}}_{\text{Remus-M1}}(\mathcal{B}) \leq \frac{9\sigma_{\text{auth}}^2 + 4\sigma_{\text{auth}} \cdot q_p}{2^n} + \frac{2q_p}{2^n} + \frac{2r\ell q_d}{2^n},$$
$$\mathbf{Adv}^{\texttt{nm-auth}}_{\text{Remus-M2}}(\mathcal{B}) \leq \frac{9\sigma_{\text{auth}}^2 + 4\sigma_{\text{auth}} \cdot q_p}{2^{2n}} + \frac{2q_p}{2^n} + \frac{2r\ell q_d}{2^n}.$$

In the above bounds, the adversary can repeat a nonce at most r times in encryption queries. The term $O(\frac{r\ell q_d}{2^n})$ is likely not tight and we expect that it can be improved to $O(r\sigma_{\text{dec}}/2^n)$ with total effective blocks in decryption queries σ_{dec}.

4.4 Security of Skinny

Skinny [3,4] 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 organisation 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 [1, 28, 40, 41]) can only reach 23 rounds of the 36 rounds in Skinny-64/128, and 19 rounds of the 40 rounds in Skinny-128/128, 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 (about 40%), 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 Remus is to provide a lightweight, yet highly-secure, highly-efficient AE based on a TBC. Remus 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 Remus still have about 40% 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 Remus setting due to data limitations, and to the fact that tweak inputs to the TBC in Remus are unknown and uncontrolled by the adversary.
- **Security proofs.** Both Remus-N and Remus-M have provable security in the Ideal-Cipher Model (ICM), where Skinny is modelled as the ideal-cipher. This is very important for high security confidence of Remus and allows us to rely on the security of Remus to that of Skinny, which has been extensively studied since the proposal in 2016. As described above, Skinny has strong security even under the related-key related-tweakey model, which is relevant to Remus. Our provable security results are relying on the model where Skinny behaves as the ideal-cipher. It tells us that unless we find a structural weakness of Skinny, Remus will be secure, as in the same way of reasoning for (public) permutation-based schemes, such as Sponge.
- **Beyond-birthday-bound security (Remus-N2 and Remus-M2).** The security bounds of Remus-N2 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, Remus-N2 and Remus-M2 (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. We note that the above comparison ignores the fact that ΘCB3 and SCT use a dedicated TBC while ours use an (ideal) block cipher. This is simply because of the lack of BBB-secure NAE/MRAE schemes based on the ideal-cipher: combining known ICM-to-TBC results [22, 30, 42] and TBC-based modes would be possible, however, the performance will be quite poor unless the tweak-dependent key derivation is very carefully considered, which is exactly the point we elaborated in our design.
- **Misuse resistance.** Remus-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 [16, 23] reveals its authentication key.

- **Performances.** Remus-N is efficient: it encrypts an n-bit block by just one call of n-bit-block primitive. Besides, it is smaller than ΘCB3 in that it does not need an additional state beyond the internal TBC and it requires a smaller TBC (smaller tweakey). Although Remus 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 [15, 24, 27]. Moreover, parallelizability is not considered crucial in lightweight applications, so it is a small price for a simple, small and fast design.

  In Remus-M, a plaintext is processed twice, once for generating a tag and once for encryption. Apart from this, Remus-M inherits the overall structure of Remus-N, and shares its general implementation characteristics.

- **Small messages.** The variants of Remus offer a trade-off between performance over short vs. long messages. For example, Remus-N1 and Remus-N2 require 3 and 4 TBC calls, respectively, for processing 1 block of AD and 1 block of message. Remus-N3 needs only 2 calls. However, the latter has a smaller block size and lower security.

- **Simplicity/Small footprint.** Remus has a quite small footprint. Especially for Remus-N, we essentially need what is needed to implement the TBC Skinny itself, and one-block mask exclusively used by Remus-N2 and Remus-M2. 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. We do not need the inverse circuit for Skinny which was needed for ΘCB3. A comparison in Section 6 (Table 6.1) shows that Remus-N is quite small and especially efficient in terms of a combined metric of size and speed, compared with other schemes.

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

- **Flexibility.** Remus has a large flexibility. Generally, it is defined as a generic mode for TBCs, and the security proofs under ICM contribute to a high confidence of the scheme when combined with a secure TBC under related-tweakey model. In fact, versions of Remus-N1, Remus-N2, Remus-M1 and Remus-M2 are used in the TGIF design [18] with a different underlying TBC.

- **Side channels.** Remus does not inherently guarantee security against Side Channel Analysis and Fault Attacks. However, standard countermeasures are easily adaptable for Remus, e.g. Fresh Rekeying [29], 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 Remus-N without violating the security model, *i.e.*, misusing the nonce or releasing unverified plaintexts. We plan to study the applicability/cost of such countermeasures, in addition to newly proposed countermeasures suited specifically for Remus in subsequent works.

Design Rationale

6.1 Overview

Remus 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. Use a small-footprint TBC instantiated by a block cipher with tweak-dependent key derivation.
4. Provable security based on the well-established ideal-cipher model.

6.2 Mode Design

Rationale of NAE Mode. By seeing Remus-N as a mode of TBC (ICE), Remus-N has a similar structure as a mode called iCOFB, which appeared in the full version of CHES 2017 paper [12]. Because it was introduced to show the feasibility of the main proposal of [10], 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, Remus-N becomes a much more advanced, sophisticated NAE mode based on a TBC. Assuming ICE is an ideally secure TBC, the security bound of Remus-N is essentially equivalent to ΘCB3, having full n-bit security. The remaining problem is how to efficiently instantiate TBC. We could use a dedicated TBC, or a conventional block cipher mode (such as XEX [37]), however, they have certain limitations on security and efficiency. To overcome such limitations, we choose to use a block cipher with tweak-dependent key/mask derivation. This approach, initiated by Mennink [30], enables the performance that cannot be achieved by the previous approaches, at the expense of the ideal-cipher model for security. Specifically, the TBC ICE has three variants, where ICE1 and ICE2 can be seen as a variant of XHX [22], and ICE3 is a more classical one combined with doubling mask [37]. Each variant has its own security level, namely, ICE1 has $n/2$-bit security, ICE2 has n-bit security, and ICE3 has $(n/2-4)$-bit security. They have different computation cost for key/mask derivations and have different state sizes. Given the n-bit security of *outer* TBC-based mode, the standard hybrid argument shows that the security of Remus-N is effectively determined by the security of the internal ICE.

Rationale of MRAE Mode. Remus-M is designed as an MRAE mode following the structure of SIV [39] and SCT [35]. Remus-M reuses the components of Remus-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. Remus-M needs an equivalent number of primitive calls as SCT. The difference is in the primitive: Remus-M uses an n-bit block cipher

while SCT uses an n-bit-block dedicated TBC. Moreover, Remus-M has a smaller state than SCT because of single-state encryption part taken from Remus-N (SCT employs a variant of counter mode). Similarly to Remus-N, the provable security of Remus-M is effectively determined by the internal ICE. For Remus-M2, thanks to n-bit security of ICE2, its security 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 decrease 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. For Remus-M1, the security is $n/2$ bits for both NR and NM adversaries due to the $n/2$-bit security of ICE1.

ZAE [19] is another TBC-based MRAE. Although it is faster than SCT, the state size is much larger than SCT and Remus-M.

Efficiency Comparison. In Table 6.1, we compare Remus-N to ΘCB3 and a group of recently proposed lightweight AEAD modes. In the table, 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. ΘCB3 is a well-studied TBC-based AEAD mode. COFB is a BC-based lightweight AEAD mode. Beetle is a Sponge-based AEAD mode, but it holds a lot of resemblance to Remus-N. 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, there is 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 of these modes show that Remus-N achieves its goals, as Remus-N1 has $2n$ state, which is smaller than COFB and equal to Beetle. Remus-N1 and COFB both have birthday security, *i.e.*, $n/2$. Beetle achieves higher security, at the expense of using a $2n$-bit permutation. Our analysis also shows that among the considered AEAD modes, Remus-N2 achieves the lowest R/S ratio, with a state size of $3n$ but only an n-bit permutation. Since Remus-N3 uses a 64-bit block cipher, we manage to achieve very small area and more relaxed state size.

Similar comparison is shown in Table 6.2 for Misuse-Resistant BC- and TBC-based AEAD modes. It shows that Remus-M2 particularly is very efficient.

Rationale of TBC. We chose some of the members of the Skinny family of tweakable block ciphers [5] 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

Table 6.1: Features of Remus-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
Remus-N1	$\lceil\frac{\lvert A\rvert}{n}\rceil+\lceil\frac{\lvert M\rvert}{n}\rceil+1$	(n,k)-BC, $n=k$	$n/2$	$n+k=4\lambda^{\dagger}$	1	4λ	Yes
Remus-N2	$\lceil\frac{\lvert A\rvert}{n}\rceil+\lceil\frac{\lvert M\rvert}{n}\rceil+2$	(n,k)-BC, $n=k$	n	$2n+k=3\lambda$	1	3λ	Yes
Remus-N3	$\lceil\frac{\lvert A\rvert}{n}\rceil+\lceil\frac{\lvert M\rvert}{n}\rceil$	(n,k)-BC, $n=k/2$	$n-4$	$n+k=3\lambda+8$	1	$3\lambda+8$	Yes
COFB [11]	$\lceil\frac{\lvert A\rvert}{n}\rceil+\lceil\frac{\lvert M\rvert}{n}\rceil+1$	(n,k)-BC, $n=k$	$n/2-\log_2 n/2$	$1.5n+k=5.4\lambda^{\ddagger}$	1	5.4λ	Yes
ΘCB3 [26]	$\lceil\frac{\lvert A\rvert}{n}\rceil+\lceil\frac{\lvert M\rvert}{n}\rceil+1$	$(n,1.5n,k)$-TBC$^{\sharp}$, $n=k$	n	$2n+2.5k=4.5\lambda$	1	4.5λ	No
Beetle [9]	$\lceil\frac{\lvert A\rvert}{n}\rceil+\lceil\frac{\lvert M\rvert}{n}\rceil+2$	$2n$-Perm, $n=k$	$n-\log_2 n$	$2n=2.12\lambda$	1/2	4.24λ	Yes
Ascon-128 [14]	$\lceil\frac{\lvert A\rvert}{n}\rceil+\lceil\frac{\lvert M\rvert}{n}\rceil+1$	$5n$-Perm, $n=k/2$	$n/2$	$7n=3.5\lambda$	1/5	17.5λ	Yes
Ascon-128a [14]	$\lceil\frac{\lvert A\rvert}{n}\rceil+\lceil\frac{\lvert M\rvert}{n}\rceil+1$	$2.5n$-Perm, $n=k$	n	$3.5n=3.5\lambda$	1/2.5	8.75λ	Yes
SpongeAE $^{\flat}$ [8]	$\lceil\frac{\lvert A\rvert}{n}\rceil+\lceil\frac{\lvert M\rvert}{n}\rceil+1$	$3n$-Perm, $n=k$	n	$3n=3\lambda$	1/3	9λ	Yes

† Can possibly be enhanced to 3λ with a different KDF and block cipher with $2k$-bit key;
‡ 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.

Table 6.2: Features of Remus-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
Remus-M1	$\lceil\frac{\lvert A\rvert+\lvert M\rvert}{n}\rceil+\lceil\frac{\lvert M\rvert}{n}\rceil+1$	(n,k)-BC, $n=k$	$n/2$	$2n=4\lambda$	1/2	8λ	Yes
Remus-M2	$\lceil\frac{\lvert A\rvert+\lvert M\rvert}{n}\rceil+\lceil\frac{\lvert M\rvert}{n}\rceil+2$	(n,k)-BC, $n=k$	n	$3n=3\lambda$	1/2	6λ	Yes
SCT † [36]	$\lceil\frac{\lvert A\rvert+\lvert M\rvert}{n}\rceil+\lceil\frac{\lvert M\rvert}{n}\rceil+1$	(n,n,k)-TBC, $n=k$	n	$4n=4\lambda$	1/2	8λ	Yes
SUNDAE [2]	$\lceil\frac{\lvert A\rvert+\lvert M\rvert}{n}\rceil+\lceil\frac{\lvert M\rvert}{n}\rceil+1$	(n,k)-BC, $n=k$	$n/2$	$2n=4\lambda$	1/2	8λ	Yes
ZAE $^{\sharp}$ [19]	$\lceil\frac{\lvert A\rvert+\lvert M\rvert}{2n}\rceil+\lceil\frac{\lvert M\rvert}{n}\rceil+6$	(n,n,k)-TBC, $n=k$	n	$7n=7\lambda$	1/2	14λ	Yes

† Tag is n bits;
$^{\sharp}$ Tag is $2n$ bits;

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

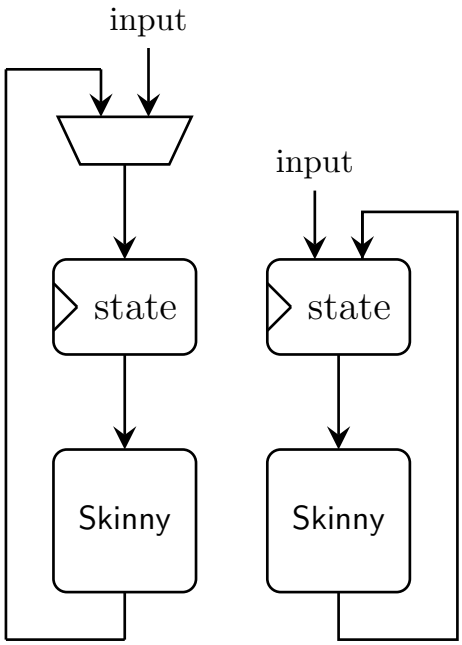
The goal of the design of Remus is to have a very small area overhead over the underlying TBC, specially for the round-based implementations. In order to achieve this goal, we set two requirements:

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.

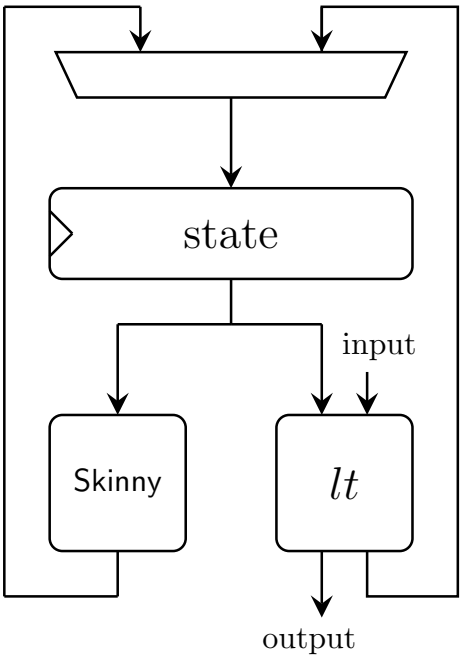
In this section, we describe various design choices that help achieve these two goals.

General Architecture and Hardware Estimates. 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-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(a-b)) [20]. 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. 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. 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.

Hardware Cost of Remus-N1. The overhead of Remus-N1 is mostly due to the doubling (3 XORs) and ρ operations (68 XORs, assuming the input/output bus has width of 32 bits). Moreover, we need 2 128-bit multiplexers to select the input to the tweakey out of four positive values: K, S (after applying the KDF function), lt, or the Skinny round key. We assume a multiplexer costs ~ 2.75 GEs and an XOR gate costs ~ 2.25 GEs. In total, this adds up to ~ 864 GEs on top of Skinny-128/128. In the original Skinny paper [3], the authors reported that the round-based implementation of Skinny-128/128 costs $\sim 2,391$ GEs. So we estimate that Remus-N1 should cost $\sim 3,255$ GEs, which is a very small figure, compared to not just TBC based AEAD modes, but in general. For example, ACORN, the smallest CAESAR candidate, costs $\sim 5,900$ GEs. Besides, two further optimizations are applicable. First, we can use the serial Skinny-128/128 implementation, which costs ~ 600 GEs less. The other direction is to unroll Skinny-128/128 to a 2- or 4-round implementation, reducing the number of cycles, at the cost of 1 KGEs per extra round. We do acknowledge that this huge gain in area comes at the cost of reduced security (birthday security).



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



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

Figure 6.1: Expected architectures for Skinny and Remus

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.

Hardware Cost of Remus-N2. Remus-N2 is similar to Remus-N1, with an additional mask V. Hence, the additional cost comes from the need to store and process this mask. The storage cost is simply 128 extra Flip-Flops. However, the processing cost can be tricky, especially since we adopt a serial concept for the implementation of ρ. Hence, we also adopt a serial concept for the processing of V. We assume that V will be updated in parallel to ρ and we merge the masking and ρ operations. Consequently, we need 64 XORs for the masking, 3 XORs for doubling, 5 XORs in order to correct the domain separation bits after each block (note that 3 bits are fixed), and 1 Flip-Flop for the serialization of doubling. Overall, we need ~ 800 GEs on top of Remus-N1, $\sim 4,055$ GEs overall, which is again smaller than almost all other AEAD designs (except other Remus variants), while achieving BBB security.

Hardware Cost for Remus-N3. Remus-N3 uses Skinny-64/128 as the TBC (via the mode ICE3). According to our estimations, it requires 224 multiplexers, 32 XOR gates for the KDF function, 68 XORs for ρ, 24 XOR gates for correcting the Tweakey and 3 XOR gates for the counting. This adds up to ~ 743 GEs on top of Skinny-64/128, which costs $1,399$ GEs. So we estimate that Remus-N3 costs $2,439$ GEs, which is a very small figure for any round-based implementation of an AEAD mode.

The arguments about serialization, unrolling and decryption are the same for all Remus-N variants. Thanks to the shared structure, these arguments also generally apply to Remus-M.

Algorithm KDF2(N, K)

1. $S \leftarrow 0^n$
2. $(S, \eta) \leftarrow \rho(S, N)$
3. $S \leftarrow E_K(S)$
4. $(S, L) \leftarrow \rho(S, 0^n)$
5. $S \leftarrow E_{K \oplus 1}(S)$
6. $(S, V) \leftarrow \rho(S, 0^n)$
7. $S \leftarrow 0^n$
8. **return** (L, V)

Figure 6.2: Alternative description of KDF2 (KDF for ICE2). KDF1 is obtained by substituting lines 5 to 7 with single line $[V \leftarrow 0^n]$.

6.4 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 TBCs whose block sizes are 128 and 64 bits, we need a block counter of a period of at least 2^{46} and 2^{47}, respectively. 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). These counters are either dedicated counter, in the case of Remus-N3, or consecutive doubling of the key L, which is equivalent to a Galois LFSR. This, in addition to the architecture described above, makes the cost of counter almost negligible.

Tag Generation. While Remus has a lot of similarities compared to iCOFB, the original iCOFB simply outputs the final chaining value as the tag. Considering hardware simplicity, we changed it so that the tag is the final output state (*i.e.*, the same way as the ciphertext blocks). 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.

Mask Generation in KDF. Similar to tag generation, we generate the masks by applying G to a standard n-bit keyed permutation. The reason for that is to be able to reuse the same circuit used for the normal opration of Remus for KDF. Moreover, it allows us to easily output and store the masks during the first pass of Remus-N, to be used during the second pass. Effectively, KDF2 is equivalent to the algorithm in Figure 6.2, where for KDF1 lines 5 to 7 will be substituted with a line $[V \leftarrow 0^n]$. This algorithm shows that the KDF has the same structure as the main encryption/decryption part of Remus itself and the same hardware circuit can be very easily reused with almost no overhead.

Padding. The padding function used in Remus 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 Remus 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 decryption, 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 $\texttt{pad}_n(C)$, 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 $C^{'} = \texttt{pad}_n(C) \oplus \texttt{msb}_{n-|C|}(G(S))$. Once $C^{'}$ is regenerated, ρ^{-1} can be computed as shown in Equation (6.2)

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

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

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:

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

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

Encryption-Decryption Combined Circuit. One of the goals of Remus 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 use 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 that 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 (e.g.) software performance. In Section 4, 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 Remus. 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 Remus is not a parallel mode. Therefore, the performance of Remus 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 Remus. We expect the effect of the Remus mode to be minor on the performances since it is rate 1.

7.2 ASIC Performances

We have implemented the round-based architecture of 4 variants of Remus, using a simple lightweight interface. The implementation uses the Skinny implementation published at the website of Skinny[1]. The results show that Remus is very lightweight as it requires about 3.5 KGE including a simple interface. Besides, Remus-M2 can be implemented in less than 5 KGE and provides 128-bit security, even in environments where randomness is not very reliable. As described in the rationale, Remus-N3 is estimated to acheive even smaller footprint.

[1] https://sites.google.com/site/skinnycipher/home

Table 7.1: ASIC Round-Based Implementations of Remus using the TSMC 65nm standard cell library. Power and Energy are estimated at 10 Mhz. Energy is for 1 TBC call for Remus-N members and 2 TBC calls for Remus-M members.

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
Remus-N1	44	3106	3611	0.98	2.96	218.5	961.4	0.82	64	-
Remus-N2	44	4230	4774	0.84	3.46	265.6	1168	0.72	128	-
Remus-M1	44(AD)/88(M)	3115	3805	1.01	2.16	278.5	2446	0.56	64	64
Remus-M2	44(AD)/88(M)	4295	4962	0.93	2.34	390.7	3440	0.47	128	64∼128

Acknowledgments

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

Bibliography

[1] Ankele, R., Banik, S., Chakraborti, A., List, E., Mendel, F., Sim, S.M., Wang, G.: Related-Key Impossible-Differential Attack on Reduced-Round Skinny. In: ACNS. Volume 10355 of Lecture Notes in Computer Science., Springer (2017) 208–228

[2] 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

[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. In: CRYPTO (2). Volume 9815 of Lecture Notes in Computer Science., Springer (2016) 123–153

[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. IACR Cryptology ePrint Archive **2016** (2016) 660

[5] 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

[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] 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

[9] 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

[10] 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

[11] 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

[12] Chakraborti, A., Iwata, T., Minematsu, K., Nandi, M.: Blockcipher-based Authenticated Encryption: How Small Can We Go? (Full version of [10]). IACR Cryptology ePrint Archive **2017** (2017) 649

[13] Chang, D., Dworkin, M., Hong, S., Kelsey, J., Nandi, M.: A Keyed Sponge Construction with Pseudo-randomness in the Standard Model. NIST SHA-3 2012 Workshop (2012)

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

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

[16] 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 Lecture Notes in Computer Science., Springer (2008) 144–161

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

[18] Iwata, T., Khairallah, M., Minematsu, K., Peyrin, T., Sasaki, Y., Sim, S.M., Sun, L.: TGIF v1. Submission to NIST Lightweight Cryptography Project (2019)

[19] 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

[20] 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

[21] 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

[22] Jha, A., List, E., Minematsu, K., Mishra, S., Nandi, M.: XHX - A Framework for Optimally Secure Tweakable Block Ciphers from Classical Block Ciphers and Universal Hashing. LATINCRYPT 2017 (2017) available at https://eprint.iacr.org/2017/1075.

[23] 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.

[24] 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

[25] 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

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

[27] 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

[28] 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

[29] 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

[30] Mennink, B.: Optimally Secure Tweakable Blockciphers. In: FSE. Volume 9054 of Lecture Notes in Computer Science., Springer (2015) 428–448

[31] Mennink, B., Reyhanitabar, R., Vizár, D.: Security of Full-State Keyed Sponge and Duplex: Applications to Authenticated Encryption. In: ASIACRYPT (2). Volume 9453 of Lecture Notes in Computer Science., Springer (2015) 465–489

[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] Mouha, N., Mennink, B., Herrewege, A.V., Watanabe, D., Preneel, B., Verbauwhede, I.: Chaskey: An Efficient MAC Algorithm for 32-bit Microcontrollers. In: Selected Areas in Cryptography. Volume 8781 of Lecture Notes in Computer Science., Springer (2014) 306–323

[35] 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

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

[37] 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

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

[39] 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

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

[41] Tolba, M., Abdelkhalek, A., Youssef, A.M.: Impossible Differential Cryptanalysis of Reduced-Round SKINNY. In: AFRICACRYPT. Volume 10239 of Lecture Notes in Computer Science. (2017) 117–134

[42] Wang, L., Guo, J., Zhang, G., Zhao, J., Gu, D.: How to Build Fully Secure Tweakable Blockciphers from Classical Blockciphers. In: ASIACRYPT (1). Volume 10031 of Lecture Notes in Computer Science. (2016) 455–483

Appendix

Table 8.1: Domain separation byte B of Remus. 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
Remus-N	-	-	0	0	0	1	0	0	4	A main
	-	-	0	0	1	1	0	0	12	A last unpadded
	-	-	0	0	1	1	0	1	13	A last padded
	-	-	0	0	0	0	1	0	2	M main
	-	-	0	0	1	0	1	0	10	M last unpadded
	-	-	0	0	1	0	1	1	11	M last padded
Remus-M	-	-	1	0	0	1	0	0	36	A main
	-	-	1	0	1	1	0	0	44	A last unpadded
	-	-	1	0	1	1	0	1	45	A last padded
	-	-	1	0	0	1	1	0	38	M auth main
	-	-	1	0	1	1	1	0	46	M auth last unpadded
	-	-	1	0	1	1	1	1	47	M auth last padded
	-	-	1	0	0	0	1	0	34	M enc main

Changelog

- 29-03-2019: version v1.0