Romulus

v1.3

Designers/Submitters (in alphabetical order):

Chun Guo[1], Tetsu Iwata[2], Mustafa Khairallah[3], Kazuhiko Minematsu[4], Thomas Peyrin[3]

[1] Shandong University, China
chun.guo@sdu.edu.cn

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

[3] Nanyang Technological University, Singapore
mustafa.khairallah@ntu.edu.sg,
thomas.peyrin@ntu.edu.sg

[4] NEC Corporation, Japan
k-minematsu@nec.com

Webpage: https://romulusae.github.io/romulus/

Contents

1. Introduction

This document specifies Romulus, a submission to the NIST Lightweight Cryptography competition, with three authenticated encryption with associated data (AEAD) schemes and a hash function, all based on a tweakable block cipher (TBC) Skinny. More precisely, Romulus consists of a nonce-based AE (NAE) Romulus-N (our main variant), a nonce misuse-resistant AE (MRAE) Romulus-M, a leakage-resilient AE Romulus-T and a hash function Romulus-H[1].

A TBC was introduced by Liskov *et al.* at CRYPTO 2002 [53]. Since its inception, TBCs have been acknowledged as a powerful primitive in that it can be used to construct simple and highly secure NAE/MRAE schemes, including ΘCB3 [51] and SCT [64]. While these schemes are computationally efficient (in terms of the number of primitive calls) and have high security, lightweight applications are not the primal use cases of these schemes, and they are not particularly suitable for small devices. With this in mind, Romulus aims at lightweight, efficient, and highly-secure AE schemes, based on a TBC.

The overall structure of Romulus-N, our main variant, shares similarity in part with a (TBC-based variant of) block cipher mode COFB [28, 29], yet, we make numerous refinements to achieve our design goal. Romulus-N has been published at FSE 2020 [44] together with Romulus-M described below. Romulus-N generally requires a fewer number of TBC calls than ΘCB3 thanks to the faster MAC computation for associated data, while the hardware implementation is significantly smaller than ΘCB3 thanks to the reduced state size and inverse-freeness (*i.e.*, TBC inverse is not needed). In fact, Romulus-N's state size is essentially what is needed for evaluating the TBC. Moreover, it encrypts an n-bit plaintext block with just one call of the n-bit block TBC, hence there is no efficiency loss. Romulus-N is extremely efficient for small messages, which is particularly important in many lightweight applications, requiring for example only 2 TBC calls to handle one associated data block and one message block (in comparison, other designs like ΘCB3, OCB3, TAE, CCM require from 3 to 5 TBC to calls in the same situation). Romulus-N achieves these advantages without a security penalty, *i.e.*, Romulus-N guarantees the full n-bit security, which is a similar security bound to ΘCB3.

If we compare Romulus-N with other size-oriented and n-bit secure AE schemes, such as conventional permutation-based AEs using $3n$-bit permutation with n-bit rate, the state size is comparable ($3n$ to $3.5n$ bits). Our advantage is that the underlying cryptographic primitive is expected to be much more lightweight and/or faster because of smaller output size ($3n$ vs n bits). In addition, the n-bit security of Romulus-N is proved under the standard model, which provides a high-level assurance for security not only quantitatively but also qualitatively. To elaborate a bit more, with a security proof in the standard model, one can precisely connect the security status of the primitive to the overall security of the mode that uses this primitive. In our case, for Romulus-N and Romulus-M, the best attack on it implies a chosen-plaintext attack (CPA) in the single-key setting against Skinny, *i.e.*, unless Skinny is broken by CPA adversaries in the single-key setting, Romulus-N and Romulus-M indeed maintain the claimed bit security. Such a guarantee is

[1]The leakage-resilient AE Romulus-T and the hash function Romulus-H have been added to the Romulus family since the version 1.3 (May 2021) of this document.

not possible with non-standard models and it is often not easy to deduce the impact of a found "flaw" of the primitive to the security of the mode. In a more general context, this gap between the proof and the actual security is best exemplified by "uninstantiable" Random Oracle-Model schemes [17, 26]. To evaluate the security of Romulus, with the standard model proof, we can focus on the security evaluation of Skinny, while this type of focus is not possible in schemes with proofs in non-standard models.

Another interesting feature of Romulus-N is that it can reduce area depending on the use cases, without harming security. If it is enough to have a relatively short nonce or a short counter (or both), which is common to low-power networks, we can save the area by truncating the tweak length. This is possible because Skinny allows to reduce area if a part of its tweak is never used. Note that this type of area reduction is not possible with conventional permutation-based AE schemes: it only offers a throughput/security trade-off. We present a detailed comparison of Romulus-N with other AE candidates in Section 6.

Romulus-M follows the general MRAE construction called SIV [69]. Romulus-M reuses the components of Romulus-N as much as possible, and Romulus-M is simply obtained by processing message twice by Romulus-N. This allows a faster and smaller operation than TBC-based MRAE SCT, yet, we maintain the strong security features of SCT. That is, Romulus-M achieves n-bit security against nonce-respecting adversaries and $n/2$-bit security against nonce-misusing adversaries. Moreover, Romulus-M enjoys a useful feature called graceful degradation introduced in [64]: it ensures that the full n-bit security is almost retained if the number of nonce repetitions during encryption is limited. The security under RUP model (Release of Unverified Plaintext, which means that the adversary is able to see the unverified plaintext in the decryption irrespective of the verification result) has been proved for Romulus-M. More specifically, the authenticity of Romulus-M remains even under RUP, with bit security ranging from n to $n/2$ depending on the number of nonce repetitions. For privacy under RUP, in the integer block cases, Romulus-M is birthday-secure under PA1, which is a variant of privacy notion under RUP. See [42] for details.

Thanks to the shared components, most of the advantages of Romulus-N mentioned above also hold for Romulus-M. We present a detailed comparison of Romulus-M with other MRAE candidates in Section 6.

Romulus-T is designed as a leakage-resilient mode following TEDT published at CHES 2020 [22]: it limits the exploitability of physical leakages via side-channel attacks, even if these leakages happen during every message encryption and decryption operations. Due to the longer tweak of Skinny enabling a rate 1[2] cryptographic hashing, Romulus-T has a slightly better rate than original TEDT, while retaining the same strong provable security bounds. Namely, it ensures $(n - \log_2(n))$-bit integrity and this holds even if everything but the keys of the key derivation/tag generation functions are leaked in side-channel attack. Moreover, it ensures $(n - \log_2(n))$-bit confidentiality. A nice additional feature is that when the nonces are repeated, these security guarantees have also been proven in the nonce misuse-resilience setting (in the sense of Ashur *et al.* [11]).

Finally, we also introduce Romulus-H which is the MDPH hashing scheme proposed by Naito at Latincrypt 2019 [60]. It is a variant of the popular double-block hashing scheme based on block cipher proposed by Hirose at FSE 2006 [39] and a domain extension scheme by Hirose et al. at Asiacrypt 2007 [40]. Romulus-H comes with a strong security guarantee, namely $(n - \log_2(n))$-bit indifferentiability [60] using an n-bit-block primitive.

As the underlying TBC for all our variants, we adopt Skinny proposed at CRYPTO 2016 [14], more precisely Skinny-128-384+ which is a reduced-round version of Skinny-128-384[3]. The security

[2]Rate is the number of input blocks processed per primitive call. For a more precise definition, please check Section 6.2

[3]From the version 1.3 (May 2021) of this document, we simplified the family members of Romulus so that all Romulus variants use the same 128-bit block and 384-bit tweakey TBC Skinny-128-384+. Note that due to the huge security margin offered by Skinny-128-384, the number of rounds has been decreased from 56 to 40 in Skinny-128-384+,

of this TBC has been extensively studied, and it has attractive implementation characteristics.

We emphasize that our main variant (Romulus-N) maintains its mode since the first proposal in 2019, and the change from the previous document (v1.2) is just the change in the number of rounds for Skinny (See Footnote 3 of this section and Changelog in Appendix B). This also holds for Romulus-M. This suggests their algorithmic maturity.

Organization of the document. In Section 2, we first introduce the basic notations and the notion of tweakable block cipher, followed by the list of parameters for Romulus, and the specification of the TBC Skinny-128-384+. In the last part of Section 2, we specify three AE variants of Romulus (Romulus-N, Romulus-M and Romulus-T) and a hash function Romulus-H. We present our security claims in Section 3 and show our security analysis including the provable security bounds and the status of computational security of Skinny in Section 4. In Section 5, we describe the desirable features of Romulus. The design rationale of our schemes, including some details of modes and choice of the TBC, is presented in Section 6. Finally, we show some implementation aspects of Romulus in Section 7.

which still maintains more than 30% security margin even for near-brute force complexity distinguishers. This directly offers a 40% boost on all software performance figures from the previous Romulus versions using Skinny-128-384 and a significant boost to hardware implementations.

2. Specification

2.1 Notations

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

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

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

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

where $\mathtt{len}_8(X)$ denotes the one-byte encoding of the byte-length of X. Here, $\mathtt{pad}_l(\varepsilon) = 0^l$. When $l = 128$, $\mathtt{len}_8(X)$ has 16 variations (*i.e.*, byte length 0 to 15), and we encode it to the last 4 bits of $\mathtt{len}_8(X)$ (for example, $\mathtt{len}_8(11) = \mathtt{00001011}$). We use $l = 128$ for Romulus-N and Romulus-M.

For Romulus-T and Romulus-H we use an injective variant. For notational convenience we define it on the whole byte strings. For any $X \in \{0,1\}^*$ of length multiple of 8 (*i.e.*, byte string), let

$$\mathtt{ipad}_l(X) = X \,\|\, 0^{l-(|X| \bmod l)-8} \,\|\, c,$$

where $c = \mathtt{len}_8(Z)$ for some Z of $(|X| \bmod l)$ bits. Here, Z is interpreted as the empty string when $|X| \bmod l = 0$, and otherwise the last partial block obtained by parsing X into l-bit blocks. We remark that X is a byte string. When $|X| \bmod l = 0$, $\mathtt{ipad}_l$ appends 0^l to X. As a special case, when $X = \varepsilon$, $\mathtt{ipad}_l(X) = 0^l$. When $|X| \bmod l \neq 0$ the padding is interpreted as applying $\mathtt{pad}_l$ to the last (partial) block. The injectivity is easily confirmed. The $\mathtt{ipad}^*_l(X)$ padding is exactly the same as $\mathtt{ipad}_l(X)$, except that an empty input returns an empty output $\mathtt{ipad}^*_l(\varepsilon) = \varepsilon$. We use $l = 128$ for Romulus-T and $l = 256$ for Romulus-H. The byte length encoding uses the last byte.

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[1]. 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 $\sum_{i=0}^{n-1} a_i 2^i$.

[1] In the previous specifications we also introduced a variant (alternating parsing) that parses a byte string into n and t-bit blocks in an alternating order. Thanks to the simplified members, the current specification does not need it.

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

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

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

where X_i denotes the $(i+1)$-st bit of X (for $i \in [\![n]\!]_0$). Therefore, when c is a multiple of 8 and X is a byte string, $\mathtt{msb}_c(X)$ and $\mathtt{lsb}_c(X)$ denote the last (rightmost) c bytes of X and the first (leftmost) c bytes of X, respectively. For example, $\mathtt{lsb}_{16}(X) = (X_7 X_6 \dots X_0 \,\|\, X_{15} X_{14} \dots X_8)$ and $\mathtt{msb}_8(X) = (X_{n-1} X_{n-2} \dots X_{n-8})$ with the above X. Since our specification is defined over byte strings, we only consider the above case for `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 $\widetilde{E}_K^{T_w}(M)$. The decryption routine is written as $(\widetilde{E}_K^{T_w})^{-1}(\cdot)$, where if $C = \widetilde{E}_K^{T_w}(M)$ holds for some (K, T_w, M) we have $M = (\widetilde{E}_K^{T_w})^{-1}(C)$. When $\mathcal{T}_\mathcal{W}$ is singleton, it is essentially a block cipher and is simply written as $E : \mathcal{K} \times \mathcal{M} \to \mathcal{M}$.

We also use a TBC as a keyless function $\widetilde{E} : \mathcal{K} \times \mathcal{T}_\mathcal{W} \times \mathcal{M} \to \mathcal{M}$, in which case we regard $\mathcal{K} \times \mathcal{T}_\mathcal{W}$ as the tweakey space $\mathcal{K}_\mathcal{T}$ and $\mathcal{M} = \{0,1\}^n$ is the message space. For any $(K, T_w) \in \mathcal{K}_\mathcal{T} (= \mathcal{K} \times \mathcal{T}_\mathcal{W})$, $\widetilde{E}(K, T_w, \cdot)$ is a permutation over $\mathcal{M}$. We write $\widetilde{E}^T(M)$ for $T = K \,\|\, T_w$.

2.2 Parameters

Members of Romulus. Romulus consists of Romulus-N, Romulus-M, Romulus-T and Romulus-H. Romulus-N implements nonce-based AE (NAE) secure against Nonce-respecting adversaries, and Romulus-M implements nonce Misuse-resistant AE (MRAE) introduced by Rogaway and Shrimpton [69]. Romulus-T implements an updated version of the leakage-resilient AE TEDT [22]. Romulus-H implements a Hash function with standard security properties, such as collision resistance and (2nd) preimage resistance. The name Romulus stands for the set of these members. The primary AEAD member of our submission is Romulus-N, and Romulus-H is our only hash function member.

Romulus-N, Romulus-M and Romulus-T have the following parameters:

- Nonce length $nl = 128$.
- Message block length $n = 128$.
- Key length $k = 128$.
- Counter bit length $d = 56$.
- Tag length $\tau = 128$.

The encryption algorithm of Romulus-N, Romulus-M and Romulus-T takes a key K, nonce N, associated data (AD) A and message M as input, and returns a ciphertext C and tag T. The decryption algorithm of Romulus-N, Romulus-M and Romulus-T takes (K, N, A, C, T) as input, and returns M or $\perp$ indicating rejection.

In Romulus-N, Romulus-M or Romulus-T, we set the maximum message plus associated data length for a single input for encryption to 2^{59} bytes[2], and the maximum ciphertext plus associated

[2]The counter can take $2^{56} - 1$ non-zero values, and this allows at least $2^{55} \times 16 = 2^{59}$ bytes.

data length for a single input for decryption to 2^{59} bytes. In other words, their encryption algorithms Romulus-N.Enc, Romulus-M.Enc and Romulus-T.Enc have the following input/output:

$$\text{Romulus-}\{\mathsf{N},\mathsf{M},\mathsf{T}\}.\mathsf{Enc}_K(N,A,M) = (C,T),$$

where $|K| = k$, $|N| = nl$, $|A| + |M|$ is at most 2^{59} bytes, $|C| = |M|$ and $|T| = \tau$ for $(k, nl, \tau) = (n, n, n)$ with $n = 128$. As for the decryption algorithms Romulus-N.Dec, Romulus-M.Dec and Romulus-T.Dec, they have the following input/output:

$$\text{Romulus-}\{\mathsf{N},\mathsf{M},\mathsf{T}\}.\mathsf{Dec}_K(N,A,C,T) = M \text{ or } \perp,$$

where $|K| = k$, $|N| = nl$, $|A|+|C|$ is at most 2^{59} bytes, $|T| = \tau$ and $|M| = |C|$ for $(k, nl, \tau) = (n, n, n)$ with $n = 128$, and $\perp$ indicates the reject symbol.

The total data limits in the encryption/decryption algorithms are implied by the security bounds (See Section 4 for details on the security bound) as well as the nonce and key sizes. The nonce size allows 2^{128} unique nonces to be processed per encryption key. Hence, the main limitation comes from the security bounds, with 128-bit security for Romulus-N and Romulus-M and 121-bit security for Romulus-T when the nonce is never repeated. These parameters cover all possible realistic data usages, even for the long-term future. This implies that when using Romulus, a user does not have to worry about changing the key after a certain period of time because a lot of data has been processed already and one is reaching the effective limits of the security bounds.

While our submission fixes $\tau = 128$, a tag for Romulus-N can be truncated if needed, at the cost of decreased security against forgery (see Section 4). Note that Romulus-M and Romulus-T do not support shorter tag due to the use of full length tag during decryption.

Romulus-N and Romulus-M use a TBC $\widetilde{E} : \mathcal{K} \times \overline{\mathcal{T}} \times \mathcal{M} \to \mathcal{M}$, where $\mathcal{K} = \{0,1\}^k$, $\mathcal{M} = \{0,1\}^n$ and $\overline{\mathcal{T}} = \mathcal{T} \times \mathcal{B} \times \mathcal{D}$. Here, $\mathcal{T} = \{0,1\}^{128}$, $\mathcal{D} = [\![2^d - 1]\!]_0$, and $\mathcal{B} = [\![256]\!]_0$ for parameter d, and $\mathcal{B}$ is also represented as a byte (see Section 2.4.1). For tweak $\overline{T} = (T, B, D) \in \overline{\mathcal{T}}$, T is always assumed to be a byte string. $\widetilde{E}$ is Skinny-128-384+ with appropriate tweakey encoding functions as described in Section 2.3. $\mathcal{T}$ is used to potentially process the nonce or an AD block, $\mathcal{D}$ is used for counter, and $\mathcal{B}$ is for domain separation, *i.e.*, deriving a small number of independent instances. The secret key is set at $\mathcal{K}$. Romulus-T uses the same TBC $\widetilde{E}$, with a domain separation $\mathcal{B}$ and a counter $\mathcal{D}$. The key space $\mathcal{K}$ takes either the secret key or temporal values that depend on nonces. Romulus-T also uses Romulus-H as a black-box subroutine.

Romulus-H has the following parameters:

- Message block length $2n$, where $n = 128$.
- Hash value length $2\tau = 256$.

The hash function Romulus-H can take a message of arbitrary length as input, i.e., it has the following input/output:

$$\text{Romulus-}\mathsf{H}(M) = T,$$

where $|M|$ is arbitrary and $|T| = 256$. Hence, the data limit of using the hash function is only implied by the provable security bound. See Section 4 for details on the security bound. It internally uses a TBC $\widetilde{E} : \mathcal{K} \times \overline{\mathcal{T}} \times \mathcal{M} \to \mathcal{M}$, where $\mathcal{K} = \{0,1\}^{128}$, $\mathcal{M} = \{0,1\}^{128}$, and $\overline{\mathcal{T}} = \{0,1\}^{256}$, as a keyless function with tweakey space $\mathcal{K}_{\mathcal{T}} = \mathcal{K} \times \overline{\mathcal{T}} = \{0,1\}^{384}$.

2.3 The Tweakable Block Cipher Skinny

In this section, we will recall the Skinny family of tweakable block ciphers [14]. We remark that the following specification is from [14] and for all variants of Romulus we will use Skinny-128-384+, the 40-round version of Skinny-128-384.

Basic Skinny Structure.

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

Skinny follows the TWEAKEY framework from [47] and thus takes a tweakey input instead of a key or a pair key/tweak. The family of lightweight block ciphers Skinny defines three main tweakey size versions, but we will use only one of them: for a block size n, we will use a version with tweakey size $t = 3n$. The tweakey state is also viewed as a collection of three 4×4 square arrays of cells. We denote these arrays $TK1$, $TK2$ and $TK3$. Moreover, we denote $TKz_{i,j}$ the cell of the tweakey state located at Row i and Column j of the z-th cell array. As for the internal state, we extend this notation to a vector view with a single subscript: $TK1_i$, $TK2_i$ and $TK3_i$. Moreover, we define the adversarial model **SK** (resp. **TK1**, **TK2** or **TK3**) where the attacker cannot (resp. can) introduce differences in the tweakey state.

Initialization.

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

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

This is the initial value of the cipher internal state and note that the state is loaded row-wise rather than in the column-wise fashion we have come to expect from the AES; this is a more hardware-friendly choice, as pointed out in [59].

The cipher receives a tweakey input $tk = tk_0 \| tk_1 \| \cdots \| tk_{46} \| tk_{47}$, where the tk_i are 8-bit cells. The initialization of the cipher's tweakey state is performed by simply setting for $0 \leq i \leq 15$: $TK1_i = tk_i$, $TK2_i = tk_{16+i}$ and $TK3_i = tk_{32+i}$. We note that the tweakey states are loaded row-wise.

The Round Function.

Skinny-128-384 has originally 56 rounds, but Skinny-128-384+ is reduced to 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.

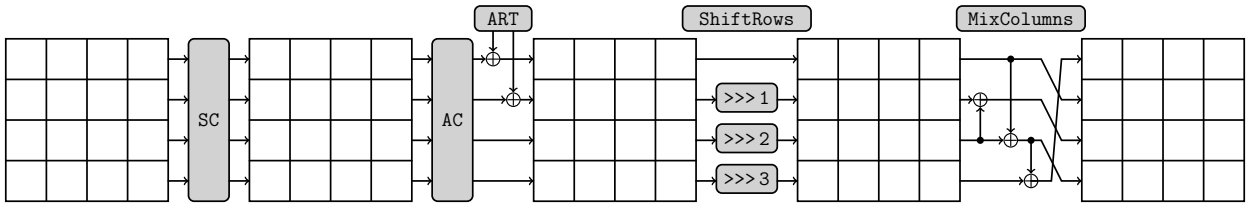


Figure 2.1: The Skinny round function applies five different transformations: `SubCells` (`SC`), `AddConstants` (`AC`), `AddRoundTweakey` (`ART`), `ShiftRows` (`SR`) and `MixColumns` (`MC`).

`SubCells`. An 8-bit Sbox is applied to every cell of the cipher internal state. The action of this Sbox is given in hexadecimal notation by the following Table 2.1.

Table 2.1: 8-bit Sbox $\mathcal{S}_8$ used in Skinny-128-384+.

```
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
};
```

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

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

followed by the bit permutation:

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

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

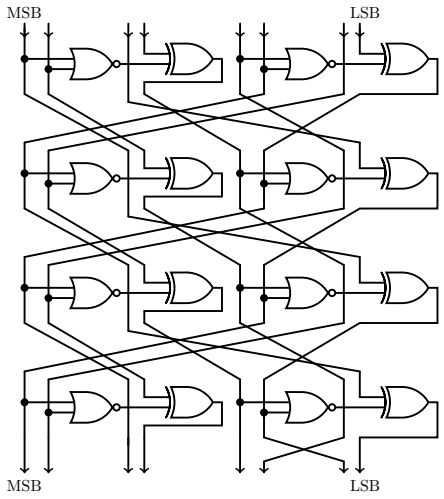


Figure 2.2: Construction of the Sbox $\mathcal{S}_8$.

`AddConstants`. A 6-bit affine LFSR, whose state is denoted (rc_5, rc_4, rc_3, rc_2, rc_1, 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 $(c_0, c_1) = (0\|0\|0\|0\|\mathrm{rc}_3\|\mathrm{rc}_2\|\mathrm{rc}_1\|\mathrm{rc}_0,\ 0\|0\|0\|0\|0\|0\|\mathrm{rc}_5\|\mathrm{rc}_4)$.

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 - 40	`31,23,06,0D,1B,36,2D,1A`

`AddRoundTweakey`. The first and second rows of all tweakey arrays are extracted and bitwise exclusive-ored to the cipher internal state, respecting the array positioning. More formally, for $i = \{0, 1\}$ and $j = \{0, 1, 2, 3\}$, we have $IS_{i,j} = IS_{i,j} \oplus TK1_{i,j} \oplus TK2_{i,j} \oplus TK3_{i,j}$.

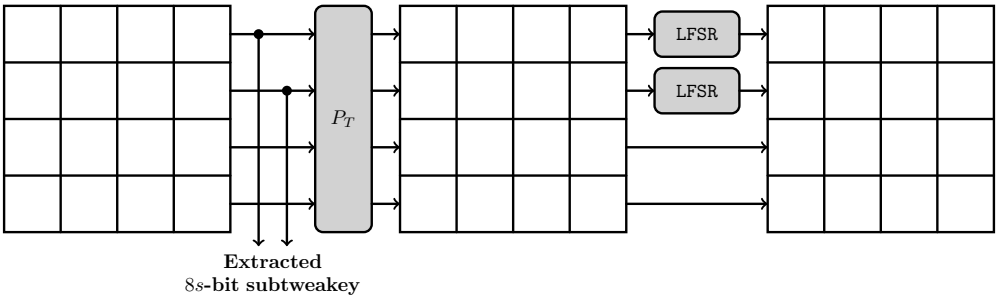


Figure 2.3: The tweakey schedule in Skinny. Each tweakey word $TK1$, $TK2$ and $TK3$ follows a similar transformation update, except that no LFSR is applied to $TK1$.

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

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

and similarly for $TK2$ and $TK3$. 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 2.2 (x_0 stands for the LSB of the cell).

Table 2.2: The LFSRs used in Skinny to generate the round constants. The TK parameter gives the number of tweakey words in the cipher.

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

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

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

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

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

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

```
/* Skinny-128-384+ */
Key:        df889548cfc7ea52d296339301797449
            ab588a34a47f1ab2dfe9c8293fbea9a5
            ab1afac2611012cd8cef952618c3ebe8
Plaintext:  a3994b66ad85a3459f44e92b08f550cb
Ciphertext: ff38d1d24c864c4352a853690fe36e5e
```

2.4 The Authenticated Encryption and Hash Function Romulus

2.4.1 The Tweakey Encoding

LFSR. We use a 56-bit LFSR for counter. $\texttt{lfsr}_{56}$ is a one-to-one mapping $\texttt{lfsr}_{56} : [\![2^{56} - 1]\!]_0 \rightarrow \{0,1\}^{56} \setminus \{0^{56}\}$ defined as follows. Let $F_{56}(\mathtt{x})$ be the lexicographically-first polynomial among the the irreducible degree 56 polynomials of a minimum number of coefficients. Specifically $F_{56}(\mathtt{x}) = \mathtt{x}^{56} + \mathtt{x}^7 + \mathtt{x}^4 + \mathtt{x}^2 + 1$ and $\texttt{lfsr}_{56}(D) = 2^D \bmod F_{56}(\mathtt{x})$.

Note that we use $\texttt{lfsr}_{56}(D)$ as a block counter, so most of the time D changes incrementally with a step of 1, and this enables $\texttt{lfsr}_{56}(D)$ to generate a sequence of $2^{56} - 1$ pairwise-distinct values. From an implementation point of view, it should be implemented in the sequence form, $\mathtt{x}_{i+1} = 2 \cdot \mathtt{x}_i \bmod F_{56}(\mathtt{x})$.

Let $(z_{55} \,\|\, z_{54} \,\|\, \ldots \,\|\, z_1 \,\|\, z_0)$ denote the state of the 56-bit LFSR. In our modes, the LFSR is initialized to 1 mod $F_{56}(\mathtt{x})$, *i.e.*, $(0^7 1 \,\|\, 0^{48})$, in little-endian format. Incrementation of the LFSR is

defined as follows:

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

Domain separation for Romulus-N, Romulus-M and Romulus-T. We will use a domain separation byte B to ensure appropriate independence between the tweakable block cipher calls in the various AE versions of Romulus. Note that no specific domain separation is used for Romulus-H (as well as for the Romulus-H call inside Romulus-T) since it is the only hash function scheme in Romulus.

Let $B = (b_7\|b_6\|b_5\|b_4\|b_3\|b_2\|b_1\|b_0)$ be the bitwise representation of this byte, where b_7 is the MSB and b_0 is the LSB. The bits $b_7b_6b_5$ are dedicated to separate the various AE schemes, namely, $b_7b_6b_5 = \texttt{000}$ for Romulus-N, $\texttt{001}$ for Romulus-M and $\texttt{010}$ for Romulus-T.

For Romulus-N and Romulus-M we then have (see Figure 2.4):

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

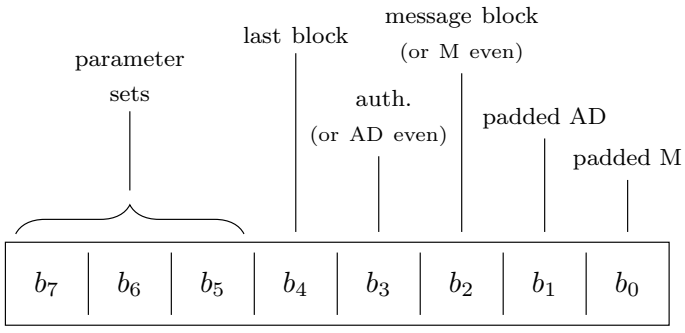


Figure 2.4: Domain separation when using the tweakable block cipher for Romulus-N and Romulus-M.

For Romulus-T we have:

- b_3, b_4 and b_5 are all always set to $\texttt{0}$.
- b_2 is set to $\texttt{1}$ for the Tag Generation Function (TGF), i.e. the TBC call that generates the tag, to $\texttt{0}$ otherwise.

- b_1 is set to `1` for the Key-Derivation Function (KDF), i.e. the TBC call that generates the session key value, to `0` otherwise.
- b_0 is used to differentiate the two parallel TBCs calls when handling the message blocks. b_0 is set to `1` for the TBC call that generates the new state value S for each new message block. It is set to `0` otherwise.

The reader can refer to Table A.1 in the Appendix to obtain the exact specifications of the domain separation values depending on the various cases.

Tweakey Encoding for Romulus-N, Romulus-M and Romulus-T. We specify the following tweakey encoding function for implementing TBC $\widetilde{E} : \mathcal{K} \times \overline{\mathcal{T}} \times \mathcal{M} \to \mathcal{M}$ using Skinny-128-384+ in Romulus-N, Romulus-M and in the message pass of Romulus-T (this encoding is not used in Romulus-H and in the Romulus-H call inside Romulus-T). The tweakey encoding is a function

$$\texttt{encode} : \mathcal{K} \times \overline{\mathcal{T}} \to \mathcal{K}_{\mathcal{T}},$$

where $\mathcal{K}_{\mathcal{T}} = \{0,1\}^{384}$ is the tweakey space for Skinny-128-384+. As defined earlier, $\overline{\mathcal{T}} = \mathcal{T} \times \mathcal{B} \times \mathcal{D}$, $\mathcal{K} = \{0,1\}^{128}$ and $\mathcal{T} = \{0,1\}^{128}$, $\mathcal{D} = [\![2^{56} - 1]\!]_0$, $\mathcal{B} = [\![256]\!]_0$.

For Romulus-N, Romulus-M and Romulus-T, the `encode` function is defined as follows:

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

For plaintext $M \in \{0,1\}^n$ and tweak $\overline{\mathcal{T}} = (T, B, D) \in \mathcal{T} \times \mathcal{B} \times \mathcal{D}$, $\widetilde{E}_K^{(T,B,D)}(M)$ denotes encryption of M with 384-bit tweakey state $\texttt{encode}(K, T, B, D)$. Tweakey encode is always implicitly applied, hence the counter D is never arithmetic in the tweakey state, unless we explicitly state otherwise. To avoid confusion, we may write $\overline{D}$ (in particular when it appears in a part of tweak) in order to emphasize that this is indeed an LFSR counter. One can interpret $\overline{D}$ as a state of LFSR when clocked D times (but in that case it is a part of tweakey state and *not* a part of input of `encode`).

Tweakey Encoding for Romulus-H. In Romulus-H including the one used in Romulus-T, for given input $R \in \{0,1\}^n, M \in \{0,1\}^{2n}$ and $L \in \{0,1\}^n$ for $n = 128$, we let

$$\begin{aligned} T &\leftarrow R \,\|\, M, \\ L' &\leftarrow \widetilde{E}^T(L), \end{aligned}$$

which is the encryption of L under the tweakey $R \,\|\, M \in \mathcal{K}_{\mathcal{T}} = \{0,1\}^{384}$.

2.4.2 State Update Function for Romulus-N and Romulus-M

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.

2.4.3 Romulus-N Nonce-Based AE Mode

The specification of Romulus-N is shown in Figure 2.5. Figure 2.6 shows the encryption of Romulus-N. For completeness, the definition of ρ is also included.

Algorithm Romulus-N.Enc$_K(N, A, M)$

1. $S \leftarrow 0^n$
2. $(A[1], \ldots, A[a]) \xleftarrow{n} A$
3. **if** $|A[a]| < n$ **then** $w_A \leftarrow 26$ **else** 24
4. $A[a] \leftarrow \mathtt{pad}_n(A[a])$
5. **for** $i = 1$ **to** $\lfloor a/2 \rfloor$
6. $\quad (S, \eta) \leftarrow \rho(S, A[2i-1])$
7. $\quad S \leftarrow \widetilde{E}_K^{(A[2i],8,\overline{2i-1})}(S)$
8. **end for**
9. **if** $a \bmod 2 = 0$ **then** $V \leftarrow 0^n$ **else** $A[a]$
10. $(S, \eta) \leftarrow \rho(S, V)$
11. $S \leftarrow \widetilde{E}_K^{(N,w_A,\overline{a})}(S)$
12. $(M[1], \ldots, M[m]) \xleftarrow{n} M$
13. **if** $|M[m]| < n$ **then** $w_M \leftarrow 21$ **else** 20
14. **for** $i = 1$ **to** $m - 1$
15. $\quad (S, C[i]) \leftarrow \rho(S, M[i])$
16. $\quad S \leftarrow \widetilde{E}_K^{(N,4,\overline{i})}(S)$
17. **end for**
18. $M'[m] \leftarrow \mathtt{pad}_n(M[m])$
19. $(S, C'[m]) \leftarrow \rho(S, M'[m])$
20. $C[m] \leftarrow \mathtt{lsb}_{|M[m]|}(C'[m])$
21. $S \leftarrow \widetilde{E}_K^{(N,w_M,\overline{m})}(S)$
22. $(\eta, T) \leftarrow \rho(S, 0^n)$
23. $C \leftarrow C[1] \,\|\, \ldots \,\|\, C[m-1] \,\|\, C[m]$
24. **return** (C, T)

Algorithm Romulus-N.Dec$_K(N, A, C, T)$

1. $S \leftarrow 0^n$
2. $(A[1], \ldots, A[a]) \xleftarrow{n} A$
3. **if** $|A[a]| < n$ **then** $w_A \leftarrow 26$ **else** 24
4. $A[a] \leftarrow \mathtt{pad}_n(A[a])$
5. **for** $i = 1$ **to** $\lfloor a/2 \rfloor$
6. $\quad (S, \eta) \leftarrow \rho(S, A[2i-1])$
7. $\quad S \leftarrow \widetilde{E}_K^{(A[2i],8,\overline{2i-1})}(S)$
8. **end for**
9. **if** $a \bmod 2 = 0$ **then** $V \leftarrow 0^n$ **else** $A[a]$
10. $(S, \eta) \leftarrow \rho(S, V)$
11. $S \leftarrow \widetilde{E}_K^{(N,w_A,\overline{a})}(S)$
12. $(C[1], \ldots, C[m]) \xleftarrow{n} C$
13. **if** $|C[m]| < n$ **then** $w_C \leftarrow 21$ **else** 20
14. **for** $i = 1$ **to** $m - 1$
15. $\quad (S, M[i]) \leftarrow \rho^{-1}(S, C[i])$
16. $\quad S \leftarrow \widetilde{E}_K^{(N,4,\overline{i})}(S)$
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. $S \leftarrow \widetilde{E}_K^{(N,w_C,\overline{m})}(S)$
23. $(\eta, T^*) \leftarrow \rho(S, 0^n)$
24. $M \leftarrow M[1] \,\|\, \ldots \,\|\, M[m-1] \,\|\, M[m]$
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 2.5: The Romulus-N nonce-based AE mode. Lines of [**if (statement) then** $X \leftarrow x$ **else** x'] are shorthand for [**if (statement) then** $X \leftarrow x$ **else** $X \leftarrow x'$]. The dummy variable η is always discarded.

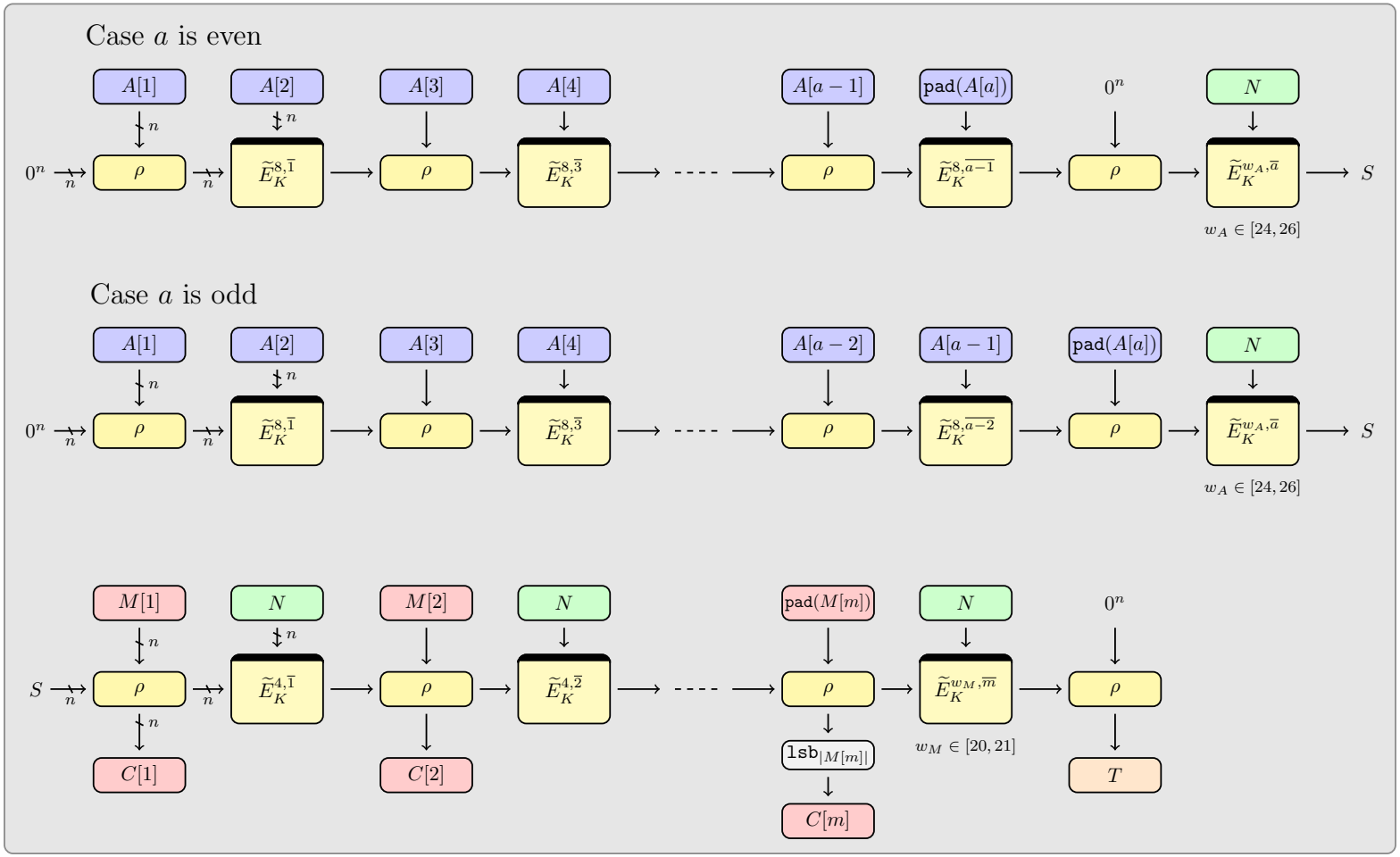


Figure 2.6: The Romulus-N nonce-based AE mode. (Top) process of AD with even AD blocks (Middle) process of AD with odd AD blocks (Bottom) Encryption.

2.4.4 Romulus-M Misuse-Resistant AE Mode

The specification of Romulus-M is shown in Figure 2.7. Figure 2.8 shows the encryption of Romulus-M.

Algorithm Romulus-M.Enc$_K(N, A, M)$

1. $S \leftarrow 0^n$
2. $(X[1], \ldots, X[a]) \xleftarrow{n} A$
3. $(X[a+1], \ldots, X[a+m]) \xleftarrow{n} M$
4. $z \leftarrow |X[a+m]|$
5. $w \leftarrow 48$
6. **if** $|X[a]| < n$ **then** $w \leftarrow w \oplus 2$
7. **if** $|X[a+m]| < n$ **then** $w \leftarrow w \oplus 1$
8. **if** $a \bmod 2 = 0$ **then** $w \leftarrow w \oplus 8$
9. **if** $m \bmod 2 = 0$ **then** $w \leftarrow w \oplus 4$
10. $X[a] \leftarrow \mathtt{pad}_n(X[a])$
11. $X[a+m] \leftarrow \mathtt{pad}_n(X[a+m])$
12. $x \leftarrow 40$
13. **for** $i = 1$ **to** $\lfloor (a+m)/2 \rfloor$
14. $\quad (S, \eta) \leftarrow \rho(S, X[2i-1])$
15. $\quad$ **if** $i = \lfloor a/2 \rfloor + 1$ **then** $x \leftarrow x \oplus 4$
16. $\quad S \leftarrow \widetilde{E}_K^{(X[2i], x, \overline{2i-1})}(S)$
17. **end for**
18. **if** $a \bmod 2 = m \bmod 2$ **then**
19. $\quad (S, \eta) \leftarrow \rho(S, 0^n)$
20. **else**
21. $\quad (S, \eta) \leftarrow \rho(S, X[a+m])$
22. $S \leftarrow \widetilde{E}_K^{(N, w, \overline{a+m})}(S)$
23. $(\eta, T) \leftarrow \rho(S, 0^n)$
24. **if** $M = \epsilon$ **then return** (ϵ, T)
25. $S \leftarrow T$
26. **for** $i = 1$ **to** m
27. $\quad S \leftarrow \widetilde{E}_K^{(N, 36, \overline{i-1})}(S)$
28. $\quad (S, C[i]) \leftarrow \rho(S, X[a+i])$
29. **end for**
30. $C[m] \leftarrow \mathtt{lsb}_z(C[m])$
31. $C \leftarrow C[1] \,\|\, \ldots \,\|\, C[m-1] \,\|\, C[m]$
32. **return** (C, T)

Algorithm Romulus-M.Dec$_K(N, A, C, T)$

1. **if** $C = \epsilon$ **then** $M \leftarrow \epsilon$
2. **else**
3. $\quad S \leftarrow T$
4. $\quad (C[1], \ldots, C[m]) \xleftarrow{n} C$
5. $\quad z \leftarrow |C[m]|$
6. $\quad C[m] \leftarrow \mathtt{pad}_n(C[m])$
7. $\quad$ **for** $i = 1$ **to** m
8. $\quad\quad S \leftarrow \widetilde{E}_K^{(N, 36, \overline{i-1})}(S)$
9. $\quad\quad (S, M[i]) \leftarrow \rho^{-1}(S, C[i])$
10. $\quad$ **end for**
11. $\quad M[m] \leftarrow \mathtt{lsb}_z(M[m])$
12. $\quad M \leftarrow M[1] \,\|\, \ldots \,\|\, M[m-1] \,\|\, M[m]$
13. $S \leftarrow 0^n$
14. $(X[1], \ldots, X[a]) \xleftarrow{n} A$
15. $(X[a+1], \ldots, X[a+m]) \xleftarrow{n} M$
16. $w \leftarrow 48$
17. **if** $|X[a]| < n$ **then** $w \leftarrow w \oplus 2$
18. **if** $|X[a+m]| < n$ **then** $w \leftarrow w \oplus 1$
19. **if** $a \bmod 2 = 0$ **then** $w \leftarrow w \oplus 8$
20. **if** $m \bmod 2 = 0$ **then** $w \leftarrow w \oplus 4$
21. $X[a] \leftarrow \mathtt{pad}_n(X[a])$
22. $X[a+m] \leftarrow \mathtt{pad}_n(X[a+m])$
23. $x \leftarrow 40$
24. **for** $i = 1$ **to** $\lfloor (a+m)/2 \rfloor$
25. $\quad (S, \eta) \leftarrow \rho(S, X[2i-1])$
26. $\quad$ **if** $i = \lfloor a/2 \rfloor + 1$ **then** $x \leftarrow x \oplus 4$
27. $\quad S \leftarrow \widetilde{E}_K^{(X[2i], x, \overline{2i-1})}(S)$
28. **end for**
29. **if** $a \bmod 2 = m \bmod 2$ **then**
30. $\quad (S, \eta) \leftarrow \rho(S, 0^n)$
31. **else**
32. $\quad (S, \eta) \leftarrow \rho(S, X[a+m])$
33. $S \leftarrow \widetilde{E}_K^{(N, w, \overline{a+m})}(S)$
34. $(\eta, T^*) \leftarrow \rho(S, 0^n)$
35. **if** $T^* = T$ **then return** M **else** $\perp$

Figure 2.7: The Romulus-M misuse-resistant AE mode. The ρ function is the same as Romulus-N. The dummy variable η is always discarded. Note that in the case of empty message, no encryption call has to be performed in the encryption part.

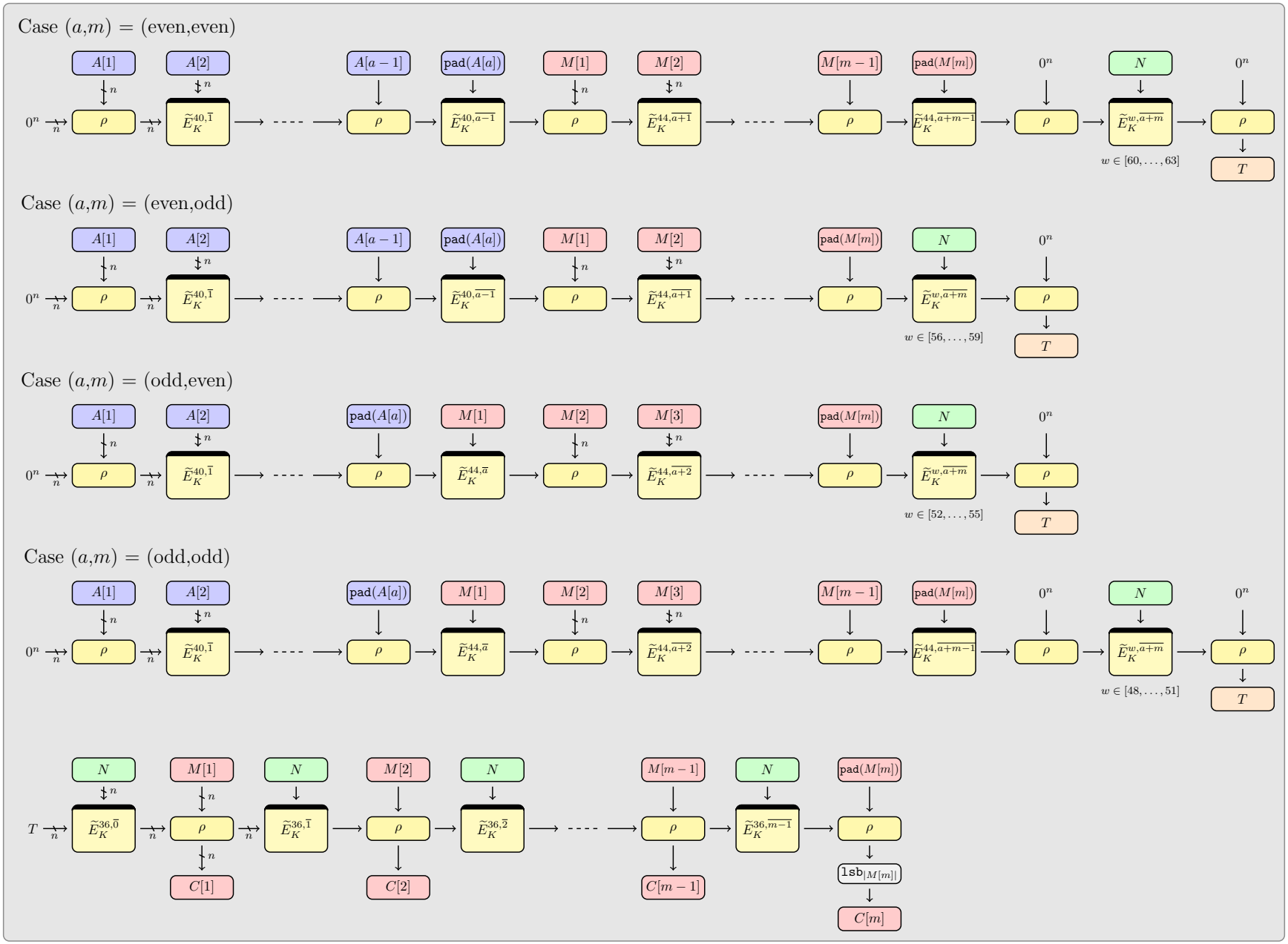


Figure 2.8: The Romulus-M misuse-resistant AE mode. (Top) process of AD with even/even, even/odd, odd/even, odd/odd AD and M blocks respectively (Bottom) Encryption. Note that $(A[1],\ldots,A[a]) \xleftarrow{n} A$ and $(M[1],\ldots,M[m]) \xleftarrow{n} M$.

2.4.5 Romulus-T Leakage-Resilient AE Mode

The specification of Romulus-T is shown in Figure 2.9. Figure 2.10 shows the encryption of Romulus-T. It internally evaluates the Romulus-H hashing, whose description can be found in Section 2.4.6.

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

1. **if** $M = \epsilon$ **then**
2. $\quad C \leftarrow \epsilon$
3. $\quad m \leftarrow 0$
4. **else**
5. $\quad (M[1], \ldots, M[m]) \xleftarrow{n} M$
6. $\quad S \leftarrow \widetilde{E}_K^{(0^n, 66, 0)}(N)$
7. $\quad$ **for** $i = 1$ **to** $m - 1$
8. $\quad\quad C[i] \leftarrow M[i] \oplus \widetilde{E}_S^{(0^n, 64, \overline{i-1})}(N)$
9. $\quad\quad S \leftarrow \widetilde{E}_S^{(0^n, 65, \overline{i-1})}(N)$
10. $\quad$ **end for**
11. $\quad Z \leftarrow \widetilde{E}_S^{(0^n, 64, \overline{m-1})}(N)$
12. $\quad C[m] \leftarrow M[m] \oplus \mathtt{lsb}_{|M[m]|}(Z)$
13. $\quad C \leftarrow C[1] \,\|\, \ldots \,\|\, C[m]$
14. $U \leftarrow \mathtt{ipad}_n^*(A) \| \mathtt{ipad}_n^*(C) \| N \| \overline{m}$
15. $H \leftarrow$ Romulus-H(U)
16. $(L, R) \xleftarrow{n} H$
17. $T \leftarrow \widetilde{E}_K^{(R, 68, 0)}(L)$
18. **return** (C, T)

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

1. **if** $C = \epsilon$ **then** $m \leftarrow 0$
2. **else** $(C[1], \ldots, C[m]) \xleftarrow{n} C$
3. $U \leftarrow \mathtt{ipad}_n^*(A) \| \mathtt{ipad}_n^*(C) \| N \| \overline{m}$
4. $H \leftarrow$ Romulus-H(U)
5. $(L, R) \xleftarrow{n} H$
6. $\underbrace{T^* \leftarrow \widetilde{E}_K^{(R, 68, 0)}(L)}_{L^* \leftarrow \left(\widetilde{E}_K^{(R, 68, 0)}\right)^{-1}(T)}$
7. $\underbrace{\textbf{if } T \neq T^* \textbf{ then return } \perp}_{\textbf{if } L \neq L^* \textbf{ then return } \perp}$
8. **else if** $C = \epsilon$ **then return** ϵ
9. **else**
10. $\quad S \leftarrow \widetilde{E}_K^{(0^n, 66, 0)}(N)$
11. $\quad$ **for** $i = 1$ **to** $m - 1$
12. $\quad\quad M[i] \leftarrow C[i] \oplus \widetilde{E}_S^{(0^n, 64, \overline{i-1})}(N)$
13. $\quad\quad S \leftarrow \widetilde{E}_S^{(0^n, 65, \overline{i-1})}(N)$
14. $\quad$ **end for**
15. $\quad Z \leftarrow \widetilde{E}_S^{(0^n, 64, \overline{m-1})}(N)$
16. $\quad M[m] \leftarrow C[m] \oplus \mathtt{lsb}_{|C[m]|}(Z)$
17. $\quad M \leftarrow M[1] \,\|\, \ldots \,\|\, M[m]$
18. **return** M

Figure 2.9: The Romulus-T leakage-resilient AE mode. Note that in the case of empty message, no encryption call has to be performed in the encryption part. Moreover, the value 0 in the tweak input of the Key-Derivation Function (KDF) and Tag Generation Function (TGF) (lines 6 and 17 in Romulus-T.$\mathsf{Enc}_K(N, A, M)$ and lines 6 and 10 in Romulus-T.$\mathsf{Dec}_K(N, A, C, T)$) is to be understood as 0^{56}, not as $\overline{0}$. For lines 6 and 7 in Romulus-T.$\mathsf{Dec}_K(N, A, C, T)$ with underbraces: normal implementations of these lines are recommended to use the upper statements, while side-channel resistant implementations of these lines shall follow the lower statements. The $\overline{m}$ in the padding denotes the result of 56-bit LFSR counter (See Section 2.4.1). Note that when $M = \varepsilon$ (for encryption) we have $m = 0$ hence $m \neq |M|_n$ for this case.

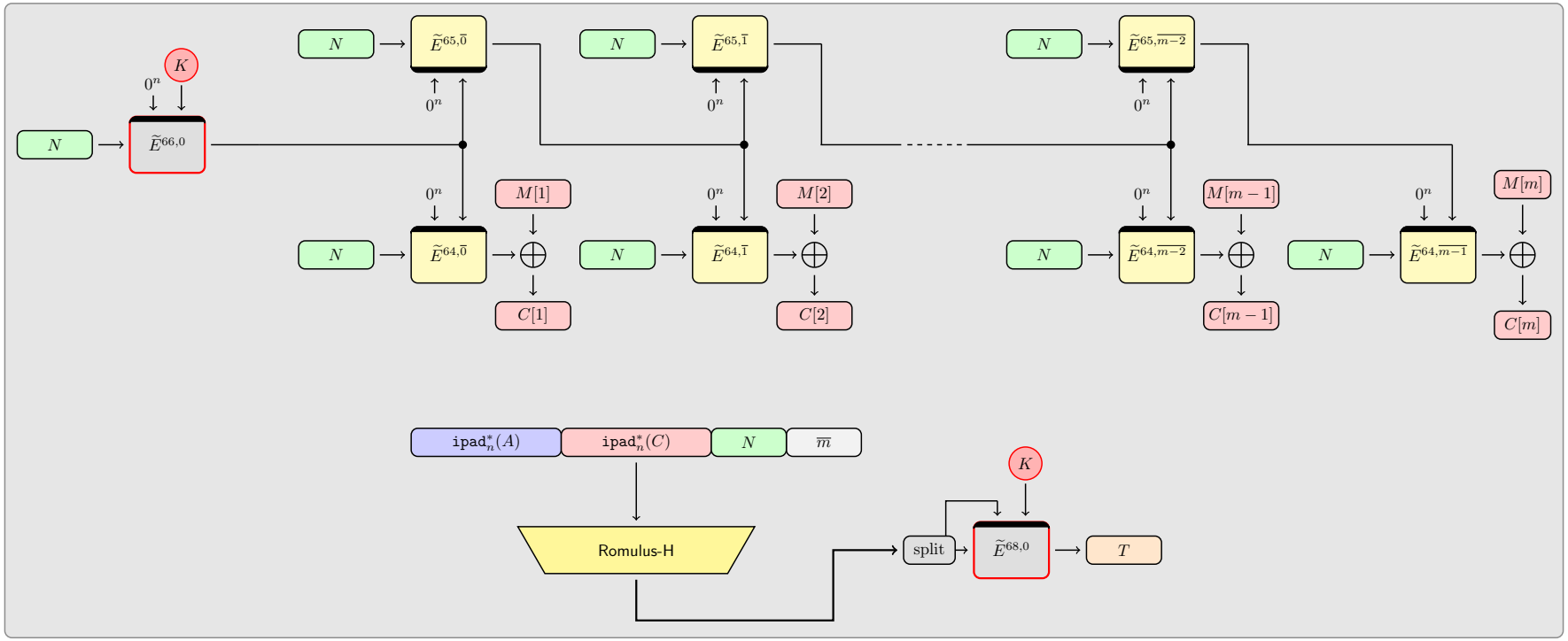


Figure 2.10: The Romulus-T leakage-resilient AE mode. The red-circled TBC calls are the Key-Derivation Function (KDF) and Tag Generation Function (TGF): for side-channel security they need heavy protection to be "leak-free", while the other TBC calls can be leaking. Note that the value 0 in the tweak input of the TGF and KDF is to be understood as 0^{56}, not as $\overline{0}$. This shows the encryption when the last message block has full n bits, otherwise we chop the TBC output. See Figure 2.9.

2.4.6 Romulus-H Hash Function

The specification of Romulus-H is shown in Figure 2.11 and it is illustrated in Figure 2.12. As described earlier, Romulus-H is an instantiation of MDPH [60] using Skinny-128-384+. MDPH is a combination of the well-known Hirose's DBL compression function [39] and the Merkle-Damgård with Permutation (MDP) domain extender [40].

Algorithm Romulus-H(M)

1. $L \leftarrow 0^n$, $R \leftarrow 0^n$
2. $(M[1], \ldots, M[m]) \xleftarrow{2n} \texttt{ipad}_{2n}(M)$
3. **for** $i = 1$ **to** $m - 1$
4. $\quad (L, R) \leftarrow \mathsf{CF}(L, R, M[i])$
5. $(L, R) \leftarrow \mathsf{CF}(L \oplus 2, R, M[m])$
6. $Y \leftarrow L \,\|\, R$
7. **return** Y

Algorithm CF(L, R, M)

1. $T \leftarrow R \,\|\, M$
2. $L' \leftarrow \widetilde{E}^T(L) \oplus L$
3. $R' \leftarrow \widetilde{E}^T(L \oplus 1) \oplus L \oplus 1$
4. **return** (L', R')

Figure 2.11: The Romulus-H hash function. Here, the constants 1 and 2 that are XORed to L denote $0^7 \,\|\, 1 \,\|\, 0^{n-8}$ and $0^6 \,\|\, 1 \,\|\, 0 \,\|\, 0^{n-8}$ respectively.

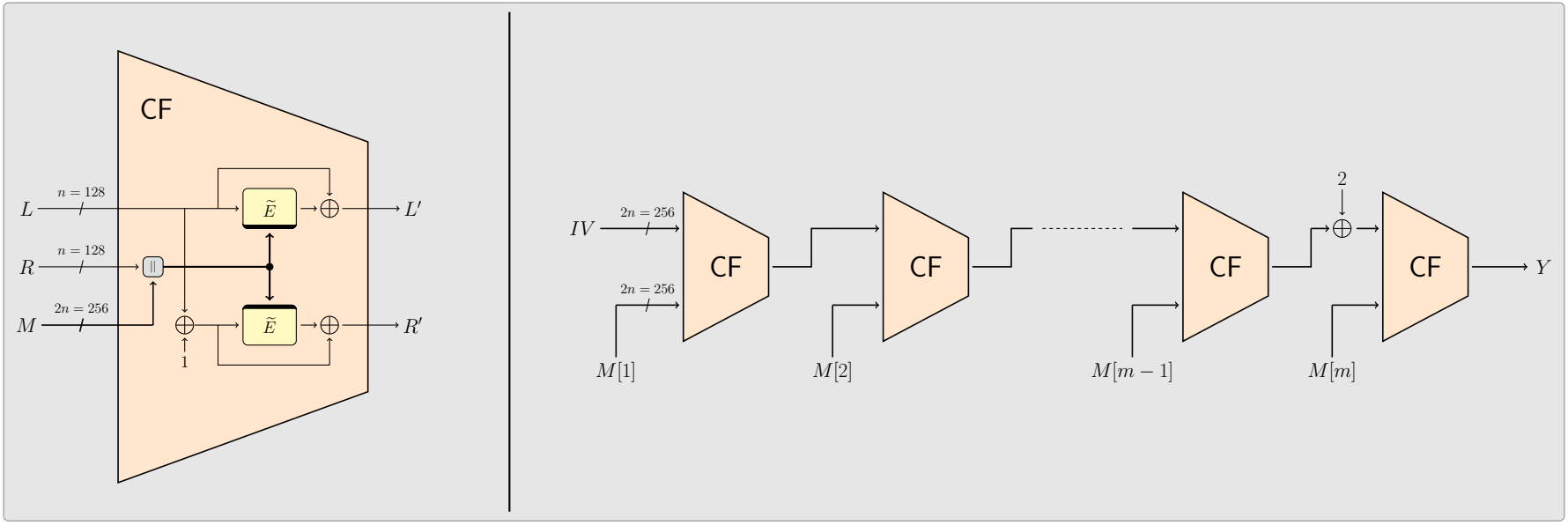


Figure 2.12: The Romulus-H hash function (compression function on the left, operating mode on the right). Here, the constant 1 and 2 denote $0^7 \| 1 \| 0^{n-8}$ and $0^6 \| 1 \| 0 \| 0^{n-8}$. The $2n$-bit chaining value in the right part is denoted by (L, R) in the code and $\oplus\, 2$ is applied to the L part.

3. Security Claims

3.1 Security Claims for Authenticated Encryption

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. For Romulus-N and Romulus-M, the variables in the table denote the required workload, in terms of data complexity, of an adversary to break the cipher, in logarithm base 2. The data complexity of attacker consists of the number of queries and the total amount of processed message blocks. If it reaches the suggested number, then there is no security guarantee anymore, and the cipher can be broken. For Romulus-T, the variables denote the overall complexity as suggested by the security bounds, namely the total (effective) queried blocks and the number of primitive queries to the ideal cipher. 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 the numbers for Romulus-N and Romulus-M hold as long as Skinny is a tweakable pseudorandom permutation, that is, it is computationally hard to distinguish Skinny from the set of uniform random permutations (URP) indexed by the tweak (a tweakable URP or TURP), using chosen-plaintext queries in the single-key setting. On the other hand, the numbers for Romulus-T hold assuming Skinny "behaves as an ideal cipher", that is, it is hard to distinguish Skinny from a random block cipher.

Table 3.1: Security claims of Romulus-N, Romulus-M and Romulus-T. NR denotes Nonce-Respecting adversary and NM denotes Nonce-Misusing adversary. In the table, $n = 128$ and small constant factors are neglected. See the main texts for the interpretations of these numbers.

Member	NR-Priv	NR-Auth	NM-Priv	NM-Auth
Romulus-N	n	n	–	–
Romulus-M	n	n	$n/2 \sim n$	$n/2 \sim n$
Romulus-T	$n - \log_2 n$	$n - \log_2 n$	–	$n - \log_2 n$

[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 $n = 128$, Table 3.1 shows 128-bit security of Romulus-N for privacy and authenticity against NR adversary. For Romulus-M, Table 3.1 shows 128-bit security for privacy and authenticity against NR adversary and in addition, 64-bit security for privacy and authenticity against NM adversary. The 64-bit security assumes that the NM adversary has full control over the nonce, but in practice, the nonce repetition can happen accidentally, and it is conceivable that the nonce is repeated only a few times. As we present in Section 4, the security bounds of Romulus-M show the notable property of graceful security degradation with respect to the number of nonce repetitions [64]. This property is similar to SCT, and if the number of nonce repetitions is limited, the actual security bound is close to the full 128-bit security.

For Romulus-T, Table 3.1 shows 121-bit security for privacy and authenticity against NR adversary, ignoring the constants as mentioned. Moreover, authenticity is fully preserved against NM adversary. On the other hand, the privacy of the encrypted messages is kept against nonce reuse, as long as the nonce used for encrypting those messages are never reused. This security notion is called nonce-misuse resilience, and is a weaker variant of NM notions in this document. The details on this notion can be found in [11]. Moreover, as we explain in Section 4, Romulus-T retains strong security even in the presence of leakage.

For key recovery attacks against Romulus-N, Romulus-M or Romulus-T that use Skinny with k-bit keys, we claim k-bit security of Skinny. That is, key recovery attacks against these modes are essentially the key recovery attack against Skinny, under the single-key setting. Note that all the members have $k = 128$.

3.2 Security Claims for Hashing Scheme

The security of MDPH was proved in [60] by Naito. It shows that MDPH, and hence Romulus-H, has indifferentiability bound of $O(2^n/n)$, where n denotes the block size of the underlying (tweakable) block cipher. Here, the (tweakable) block cipher is assumed to be an ideal cipher. This means Romulus-H could be safely used as a random oracle in most single-stage security games, and our claim on collision and (2nd) preimage security in Table 3.2 is based on the result. We remark that small constant factors are neglected in Table 3.2, and it shows the overall complexity required for collision, preimage, and 2nd preimage attacks: "overall" means data plus time, more precisely the total queried message blocks plus the number of primitive queries (i.e. offline queries to the ideal cipher). See Section 4.5 for more details on the indifferentiability bound.

Table 3.2: Security claims of Romulus-H. In the table, $n = 128$ and small constant factors are neglected.

Member	Collision	Preimage	2nd Preimage
Romulus-H	$n - \log_2 n$	$n - \log_2 n$	$n - \log_2 n$

We remark that MDPH is based on Hirose's DBL compression function, and this has been proved to achieve $n-2$ bit collision security [39] and roughly $2n-5$ bit (2nd) preimage security [10]. Our security claim in Table 3.2 is based on [60] that directly analyzes the indifferentiability of MDPH/Romulus-H.

4. Security Analysis

4.1 Security Notions

Security Notions for NAE. We consider the standard security notions for nonce-based AE [18, 19, 67]. Let Π denote an NAE scheme consisting of an encryption procedure $\Pi.\mathcal{E}_K$ and a decryption procedure $\Pi.\mathcal{D}_K$, for secret key K uniform over set $\mathcal{K}$ (denoted as $K \xleftarrow{\$} \mathcal{K}$). For plaintext M with nonce N and associated data A, $\Pi.\mathcal{E}_K$ takes (N, A, M) and returns ciphertext C (typically $|C| = |M|$) and tag T. For decryption, $\Pi.\mathcal{D}_K$ takes (N, A, C, T) and returns a decrypted plaintext M if authentication check is successful, and otherwise an error symbol, $\perp$.

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

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

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

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

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

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

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

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

Ideal Cipher Model. The ideal cipher model is a widespread model in which all parties (including the attackers) are granted access to a random cipher, i.e., a block cipher that is randomly picked from all block ciphers with consistent parameters. An ideal cipher with n-bit blocks and k-bit keys defines 2^k independent random n-bit permutations. Random ciphers do not exist in reality [26]. Though, when it's infeasible to exhibit "non-random" properties w.r.t. a secure block cipher, security analyses sometimes assume the block cipher as "behaving as an ideal cipher" and proceed in the ideal cipher model. This is in the same spirit as the random oracle model. The security analyses of Romulus-T and Romulus-H indeed rely on this idealized model, i.e., assuming Skinny behaving as an ideal cipher. In this case, the offline computations are captured by the queries to the ideal cipher.

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

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

and the authenticity advantage as

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

We note that while NM adversaries can repeat nonces, we without loss of generality assume that they do not repeat the same query. See also [69] for reference.

Security Notions for Hash. The indifferentiability [55] is an extended notion of classical indistinguishability in case the target scheme contains publicly-accessible primitives. It is particularly useful as a strong security notion for hashing schemes based on ideal primitives [32]. Let H be a function that accesses to the oracle of another ideal primitive $\mathcal{O}$, and let $\mathcal{R}$ be an ideal primitive that has the same I/O spaces as H. Let $\mathcal{S}$ be an algorithm that has the same interface as $\mathcal{O}$, which we call a simulator, and has an oracle access to $\mathcal{R}$. If H is implemented with $\mathcal{O}$ we write it as $H^{\mathcal{O}}$. The indifferentiability advantage of $H^{\mathcal{O}}$ (with ideal primitive $\mathcal{O}$) with respect to the simulator $\mathcal{S}$ is defined as

$$\mathbf{Adv}^{\mathtt{indiff}}_{(H^{\mathcal{O}},\mathcal{O}),(\mathcal{R},\mathcal{S})}(\mathcal{A}) \stackrel{\text{def}}{=} \Pr\left[\mathcal{A}^{H^{\mathcal{O}}(\cdot),\mathcal{O}(\cdot)} \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{\mathcal{R}(\cdot),\mathcal{S}(\cdot)} \Rightarrow 1\right],$$

and we say H is indifferentiable from $\mathcal{R}$ with attack complexity θ (such as the number of queries), if for any adversary $\mathcal{A}$ with complexity θ, there exists a simulator $\mathcal{S}$ such that $\mathbf{Adv}^{\mathtt{indiff}}_{(H^{\mathcal{O}},\mathcal{O}),(\mathcal{R},\mathcal{S})}(\mathcal{A})$ is negligible. As mentioned earlier, the indifferentiability notion covers the classical collision-resistance, preimage and 2nd preimage resistance notions [9]. For these classical notions, refer to [68].

4.2 Security of Romulus-N

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

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

$$\mathbf{Adv}^{\mathtt{priv}}_{\text{Romulus-N}}(\mathcal{A}) \le \mathbf{Adv}^{\mathtt{tprp}}_{\tilde{E}}(\mathcal{A}'),$$
$$\mathbf{Adv}^{\mathtt{auth}}_{\text{Romulus-N}}(\mathcal{B}) \le \mathbf{Adv}^{\mathtt{tprp}}_{\tilde{E}}(\mathcal{B}') + \frac{3q_d}{2^n} + \frac{2q_d}{2^\tau}$$

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

The security of Romulus-N crucially relies on the $n \times n$ matrix G defined over GF(2). Let $G^{(i)}$ be an $n \times n$ matrix that is equal to G except the $(i+1)$-st to n-th rows, which are set to all zero. Here, $G^{(0)}$ is the zero matrix and $G^{(n)} = G$, and for $X \in \{0,1\}^n$, $G^{(i)}(X) = \mathtt{lsb}_i(G(X)) \| 0^{n-i}$ 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 [29] and presented at [44]. We have verified the soundness of our G for $n = 128$, by a computer program.

4.3 Security of Romulus-M

Let $\mathcal{A}$ be an adversary against Romulus-M using q encryption queries with time complexity t_A and with total number of effective blocks σ_{priv}. Here, for an encryption query (N, A, M), we define the number of effective blocks as $\lfloor a/2 \rfloor + \lfloor m/2 \rfloor + m + 2$, where $|A|_n = a$ and $|M|_n = m$. In the NR case, we have

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

and in the NM case, we have

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

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

Let $\mathcal{B}$ be an adversary using q_e encryption queries and q_d decryption queries, with total number of effective blocks for encryption and decryption queries σ_{auth}, and time complexity t_B. Here, for a decryption query (N, A, C, T), the number of effective blocks is defined as $\lfloor a/2 \rfloor + \lfloor m/2 \rfloor + m + 2$, where $|A|_n = a$ and $|C|_n = m$.

Then in the NR case, we have

$$\mathbf{Adv}^{\mathtt{auth}}_{\mathsf{Romulus\text{-}M}}(\mathcal{B}) \leq \mathbf{Adv}^{\mathtt{tprp}}_{\widetilde{E}}(\mathcal{B}') + \frac{5q_d}{2^n}.$$

In the NM case, we have

$$\mathbf{Adv}^{\mathtt{nm\text{-}auth}}_{\mathsf{Romulus\text{-}M}}(\mathcal{B}) \leq \mathbf{Adv}^{\mathtt{tprp}}_{\widetilde{E}}(\mathcal{B}') + \frac{4rq_e}{2^n} + \frac{5rq_d}{2^n}.$$

Here, r is the maximum number of the repetition of a nonce in encryption queries, and $\mathcal{B}'$ uses σ_{auth} chosen-plaintext queries with time complexity $t_B + O(\sigma_{\mathrm{auth}})$. The proofs are shown at [44].

4.3.1 RUP Security

The release of unverified plaintext (RUP) is the attack model introduced by Andreeva et al. [7], where the adversary learns the unverified plaintext irrespective of the verification result. This can happen by the limited memory of the decryption device or simply misuse. We briefly describe the relevant security notions and the security of Romulus-M under RUP. For details refer to [42].

Authenticity under RUP. The authenticity under RUP, also known as INT-RUP, represents the probability of a successful forgery given three oracles, namely the encryption, the unverified decryption, and the verification oracles. We write $(q_e, q_d, q_v, t, \sigma)$-adversary to mean an INT-RUP adversary using q_e encryption queries, q_d unverified decryption queries, q_v verification queries, with total time complexity t and the total number of TBC calls σ. For any AE Π, two advantage functions are defined, $\mathbf{Adv}^{\mathtt{auth\text{-}rup}}_{\Pi}$ and $\mathbf{Adv}^{\mathtt{nm\text{-}auth\text{-}rup}}_{\Pi}$, in an analogous way as $\mathbf{Adv}^{\mathtt{auth}}_{\Pi}$ and $\mathbf{Adv}^{\mathtt{nm\text{-}auth}}_{\Pi}$ by removing the (normal) decryption oracle and attaching the unverified decryption and the verification

oracles. We have the following result. Let $\mathcal{A}$ be a nonce-respecting, $(q_e, q_d, q_v, t_A, \sigma_A)$-adversary, and let $\mathcal{B}$ be a nonce-misusing $(q_e, q_d, q_v, t_B, \sigma_B)$-adversary. Then we have

$$\mathbf{Adv}^{\texttt{auth-rup}}_{\textsf{Romulus-M}}(\mathcal{A}) \leq \mathbf{Adv}^{\texttt{tprp}}_{\widetilde{E}}(\mathcal{A}') + \frac{5q_v}{2^n},$$
$$\mathbf{Adv}^{\texttt{nm-auth-rup}}_{\textsf{Romulus-M}}(\mathcal{B}) \leq \mathbf{Adv}^{\texttt{tprp}}_{\widetilde{E}}(\mathcal{B}') + \frac{4rq_e + 5rq_v}{2^n}$$

for some $\mathcal{A}'$ using σ_A queries with time $t_A + O(\sigma_A)$, and some $\mathcal{B}'$ using σ_B queries with time $t_B + O(\sigma_B)$. These bounds are essentially the same as the regular authenticity bounds of Romulus-M shown above. The proofs are almost immediate from those of regular authenticity bounds of Romulus-M [44].

Privacy under RUP. The privacy notion in the RUP setting is called plaintext awareness [7]. Intuitively, it requires the existence of an extractor that can simulate the (unverified) decryption oracle without knowing the secret key. It has two versions, called PA1 and PA2, and the stronger notion of PA2 can be achieved only with a wide-block CCA-secure (tweakable) block cipher used in the (heavier) encode-then-encipher approach (e.g., AEZ [41] or various TESs (Tweakable Enciphering Schemes)). It is easy to see that Romulus-M does not achieve PA2.

It is known that the SIV construction is PA1 secure [8, Proposition 6]. The construction of Romulus-M is not covered by the SIV construction (due to the use of a nonce in the encryption part), however, it can be shown that Romulus-M is PA1 secure by following the proof of [8, Proposition 6], which proves that the scheme is PA1 secure if the MAC part is a PRF and the encryption part is PA1 secure. The MAC part of Romulus-M is a secure PRF, and the encryption part can be proved to be PA1 secure by following the proof of PA1 security of CBC mode and CTR mode [8, Proposition 12]. The detailed analysis for the integer blocks is given in [42], which shows that Romulus-M achieves PA1 security up to the birthday bound for this case.

4.4 Security of Romulus-T

Let $\mathcal{A}$ be an adversary against Romulus-T using q encryption queries and p ideal cipher queries and with total number of effective blocks σ_{priv}. Here, for an encryption query (N, A, M), we define the number of effective blocks as $3m + a + 5$, where $a = |A|_n$ and $m = |M|_n$. In the NR case, we have

$$\mathbf{Adv}^{\texttt{priv}}_{\textsf{Romulus-T}}(\mathcal{A}) \leq \frac{25n(\sigma_{\text{priv}} + p)}{2^n} + \frac{27(\sigma_{\text{priv}} + p)}{2^n} + \frac{1}{2^n},$$

where the underlying game allows $\mathcal{A}$ to query the ideal cipher, for both encrypt and decrypt directions, in both real and ideal worlds.

Let $\mathcal{B}$ be an adversary using q_e encryption queries, q_d decryption queries, and p ideal cipher queries, with total number of effective blocks for encryption and decryption queries σ_{auth}. Here, for a decryption query (N, A, C, T), the number of effective blocks is defined as $3m + a + 5$, where $a = |A|_n$ and $m = |C|_n$.

Then in the NR case, we have

$$\mathbf{Adv}^{\texttt{auth}}_{\textsf{Romulus-T}}(\mathcal{B}) \leq \frac{18n(\sigma_{\text{auth}} + p)}{2^n} + \frac{27(\sigma_{\text{auth}} + p)}{2^n},$$

where the underlying game allows $\mathcal{A}$ to query the ideal cipher as in the case of privacy advantage.

4.4.1 Side-channel security of Romulus-T

Berti et al. proved leakage-resilience for TEDT w.r.t. to certain definitions and assumptions [22]. Similar conclusions could be drawn on Romulus-T. To ease understanding, we eschew the complicated definitions and leakage models in favor of less formal claims and interpretations.

To ensure strong security against side-channel attacks, Romulus-T could be implemented in a "leveled" approach, i.e., two types of implementations of Skinny-128-384+ are used. One implementation has been added heavy protection, and is secure against side-channel attacks with high data complexities (e.g., differential power analysis). The other implementation is only weakly protected and secure against side-channel attacks with very low-data complexity (e.g., simple power analysis). In addition, the number of calls to heavily protected (and inefficient) implementations is minimized. Concretely,

- To implement Romulus-T encryption, only line 6 and line 17 in Figure 2.9 have to invoke the heavily protected Skinny-128-384+ function/modular. The other operations can simply invoke the weakly protected Skinny-128-384+;
- To implement Romulus-T decryption, only line 6 and line 10 in Figure 2.9 have to invoke the heavily protected Skinny-128-384+ function/modular. The other operations can simply invoke the weakly protected Skinny-128-384+. Since line 6 invokes the inverse $\left(\widetilde{E}_K^{(R,68,0)}\right)^{-1}$, this means for side-channel secure decryption we need the protected *decryption* Skinny-128-384+ function/modular.

In the face of side-channel leakages, such a leveled implementation of Romulus-T ensures security as follows:

- As long as the side-channel attacker has not recovered the key K, integrity is ensured up to $2^n/n$ computations, even if nonces are reused in arbitrary. Since a heavily protected Skinny-128-384+ modular is not expected to resist attacks with such high complexities, it determines the concrete side-channel security.
- As long as:
  - the side-channel attacker has not recovered the key K, and
  - the side-channel attacker has not recovered the internal state that appeared during encrypting the confidential messages,
  - nonces used for encrypting confidential messages are never reused,

  confidentiality is ensured. Assume that the heavily protected Skinny-128-384+ modular is secure against side-channel attacks with less than D data, and the weakly protected Skinny-128-384+ implementation is secure against side-channel attacks with very few data (e.g., SPA attacks with 4 data), then the leveled Romulus-T implementation could securely encrypt D messages with no more than $2^{n/2}$ blocks.

We stress again that, due to the informal nature of the above claims and the less immaturity of leakage-resilience, the concrete interpretations may not be fully accurate in practice.

4.5 Security of Romulus-H

The indifferentiability bound of MDPH, hence Romulus-H, has been proved in [60]. Let E be an ideal cipher, which allows accesses to its encryption and decryption oracles. Let $\mathcal{R}$ be the random oracle of $2n$-bit output. For any indifferentiability adversary $\mathcal{A}$ making q hash queries of total σ message blocks, and p primitive queries and runs in time t, there exists a simulator $\mathcal{S}$ such that[1]

$$\mathbf{Adv}^{\mathtt{indiff}}_{(\mathsf{MDPH}^E,E),(\mathcal{R},\mathcal{S})}(\mathcal{A}) \leq \frac{8\mu Q}{2^n - 2Q} + 3 \cdot 2^n \cdot \left(\frac{2eQ}{\mu(2^n - 2Q)}\right)^{\mu}.$$

[1]We remark that Naito's indifferentiability simulator in [60] has a quadratic running time, and the concrete security of Romulus-H as a random oracle shall take this into consideration.

The parameter Q denotes $\sigma + p$, and μ is any positive integer, and $e = 2.71828\ldots$ is the Napier constant. For $n = 128$, by putting $\mu = 15$ we have $\mathbf{Adv}^{\mathtt{indiff}}_{(\mathsf{MDPH}^E,E),(\mathcal{R},\mathcal{S})}(\mathcal{A}) \leq \frac{120Q}{2^n-2Q} + \left(\frac{148Q}{2^n-2Q}\right)^{15}$, and the bound is less than $1/2$ as long as $Q \leq 2^{119}$.

4.6 Security of Skinny

Skinny [14, 15] 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 (in order to avoid a very large collection of references, we only refer to the current best attacks in this document). This was also further motivated by the organization of cryptanalysis competitions of Skinny by the designers.

To the best of our knowledge, the cryptanalysis that can attack the highest number of rounds (related-tweakey impossible differential attack [54, 70]) can only reach 27 of the 56 rounds of Skinny-128-384 (or of the 40 rounds of Skinny-128-384+), with a very high data/memory/time complexity. Even in the hash function setting where the attacker has a lot of control, the best (preimage) cryptanalysis of Romulus-H can only reach 23 rounds [35], for a computational complexity of 2^{248}, thus way beyond our security claims anyway.

All in all, we can conclude that the version of Skinny that we use has a very large security margin (about 33% and actually more if we limit to 2^{128} complexity), even after numerous third party cryptanalysis. In addition, this reasoning is made without even targeting the primitive inside one of the modes, which would of course reduce the attacker's capabilities in practice. 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).

5. Features

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

- **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. Notably, the Skinny version that is used in Romulus, Skinny-128-384+ still has more than 30% security margin in the related-key related-tweakey model (even much more if complexity is limited to 2^{128}). Actually, the security margin rate is probably even higher as these attacks can't be directly applied to Skinny in the Romulus setting due to data limitations, limited tweak space, etc. Moreover, our security assumption on the internal primitive is only single-key, not related-key. We note that Skinny is being considered for ISO standardization (currently in DIS stage for ISO/IEC 18033-7) and that it has already been deployed in the French Covid tracing application after a recommendation from the French Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI).

- **Security proofs.** Romulus-N, Romulus-M and Romulus-T have provable security reductions to Skinny. The reductions are in the standard model for Romulus-N and Romulus-M, and in the ideal-cipher model for Romulus-T. See [22, 43] for the proofs. This is very important for high security confidence of Romulus and allows us to rely on the security of Romulus to that of Skinny, which has been extensively studied since the proposal in 2016.

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

- **Misuse resistance/resilience.** Romulus-M is an MRAE mode which is secure against misuse (repeat) of nonces in encryption queries. More formally, it provides the best-possible security against nonce repeat in that ciphertexts do not give any information as long as the uniqueness of the input tuple (N, A, M) is maintained. In contrast to this, popular nonce-based AE modes are often vulnerable against nonce repeat, even one repetition can be significant. For example, the famous nonce repeat attack against GCM [38, 48] reveals its authentication key.

  Romulus-T guarantees security for a weaker security notion, yet meaningful in practice,

regarding repetition of nonces (so-called nonce misuse-resilience [11]).

- **Performances.** Romulus-N is smaller than ΘCB3 in that it does not need an additional state beyond the internal TBC. Besides, while it is rate 1 for the message input, it is faster than ΘCB3 as it processes two AD blocks per TBC call. In general, it requires only $\lceil \frac{|A|-128}{256} \rceil + \lceil \frac{|M|}{128} \rceil + 1$ TBC calls, as opposed to ΘCB3, which requires $\lceil \frac{|A|}{128} \rceil + \lceil \frac{|M|}{128} \rceil + 1$. Although Romulus is serial in nature, *i.e.*, not parallelizable, it was shown during the CAESAR competition that parallelizability does not lead to significant performance gains in hardware performance, [36, 49, 52]. Moreover, parallelizability is not considered crucial in lightweight applications, so it is a small price for a simple, small and fast design.

  In Romulus-M, a plaintext is processed twice, once in the authentication part and once in the encryption part. Romulus-M inherits the overall design of Romulus-N and thanks to the highly efficient authentication part (rate 2) the efficiency loss is minimized. Romulus-M is about only 1.5 times slower than Romulus-N (thus rate 2/3) when associated data is empty, and becomes closer to Romulus-N (rate 2) for long associated data.

  In Romulus-T, a plaintext is processed once with two TBC calls per blocks on average, but the corresponding ciphertext is processed together with the associated data in the hash function. Since the hash function has rate 1, the overall rate of Romulus-T is 1/3 when associated data is empty, and becomes closer to rate 1 for long associated data.

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

  Romulus-M also has a small footprint due to the shared structure with Romulus-N. Romulus-T requires larger area, but of course for much stronger leakage-resilience security guarantees and thus should be compared with schemes with similar levels of side-channel protections.

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

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

- **Side channels and Fault Attacks.** Romulus-N or Romulus-M do not inherently guarantee security against side-channel analysis and fault attacks. However, standard countermeasures are easily adaptable to them, e.g. fresh rekeying [56], masking [57], etc. Moreover, powerful fault attacks that require a small number of faults and pairs of faulty and non-faulty ciphertexts, such as DFA, are not applicable to Romulus-N without violating the security model, *i.e.*, repeating the nonce or releasing unverified plaintexts.

  We also note that in Romulus-N, we do not require the full tweakey size of Skinny-128-384+, so a potential countermeasure to both SCA and DFA is to randomize the round keys by adding a random value to each TBC call. The downfall of this idea is that the randomness

needs to be synchronized for correct decryption, but this property is shared with most SCA and DFA randomized countermeasures.

Of course, Romulus-T on the other hand provides very strong side-channel protection with its natural leakage-resilience. It offers what is currently considered as the highest possible security notions in the presence of leakage, namely beyond birthday bound CIML2 and security against Chosen Ciphertext Attacks with nonce-misuse-resilience and Leakage using levelled implementations (CCAmL2).

We list the features of Romulus-H.

- **Security of the primitive.** As already described at Section 4.6, Skinny has strong security margins, and this also applies to the hash function setting as well.
- **Strong security notion.** A hashing scheme with indifferentiability means certain stronger guarantees than those with classical notions, and avoids pitfalls in composition up to some restrictions [33, 66]. We remark that this security claim is given under the ideal cipher model, thus something orthogonal to our reliance in the standard model. However, we think this is a reasonable trade-off in case of hash function: indifferentiability proof naturally requires a non-standard model such as ICM or (public) random permutation model. Most of existing block cipher-based hashing schemes assume ICM even for classical notions [24].
- **Performance.** Romulus-H leverages the relatively large tweakey size of Skinny-128-384+ to reach an excellent rate of 1. Besides, implementations can also leverage the nice properties of Skinny. For example, given a single Skinny-128-384+ hardware core, the implementation requires two invocations of that core, with an auxiliary storage of $2n$ bits. As in the case of Romulus-N, the tweakey value does not need to be stored in an auxiliary register as it can be easily extracted from the cipher, and both invocations use the same key. In total, this would require $6n$ flip-flops including those needed for the cipher itself. At the cost of $2.5n$ flip-flops, Romulus-H can be easily added to an existing implementation of Romulus-N, given they share similar primitives, *e.g.* padding. Another trade-off is to leverage the the internal parallelism of Romulus-H, and use two Skinny cores. This strategy also requires $6n$ flip-flops, but it requires the combinational part of Skinny to be implemented twice, *i.e.*, it requires slightly more area, but it is twice faster. This approach can be useful for stand-alone hash implementations. We note that it is typical for hash functions built from TBCs to requires extra storage compared to AEAD, *e.g.* Skinny-Hash [16] requires similar area and storage, for 1/3 of the speed. It is also typical for Sponge-based hash functions to need a storage of $3\times$ the block size. On the other hand, we note that while Romulus-H requires two invocations of Skinny to hash 1 message block, the block size is double that of Romulus-N. Hence, a combined implementation should have somewhat matching speed and energy consumption for both AEAD and the hash function. This is an interesting feature as many practical hash functions are slower than their corresponding AEAD schemes.

6. Design Rationale

6.1 Overview

Romulus is a family of algorithms designed to achieve lightweight hardware performance while maintaining competitive security guarantees and software performance. In particular, the following goals we placed in mind:

1. Have minimal overhead on top of the underlying primitive, which translates to very small area compared to TBC-based designs of similar parameters.
2. Have relatively high efficiency in general by using a small number of TBC calls and a TBC that offers a range of performance trade-offs.
3. Whenever possible, have smaller overhead and fewer TBC calls for the AD processing.
4. Base the security on the established security models of block ciphers and use the TBC as a black-box. In particular, our main variant (Romulus-N) achieves the standard model security.

6.2 Mode Design

Rationale of the NAE Mode. Romulus-N has a similar structure as a mode called iCOFB, which appeared in the full version of CHES 2017 paper [29]. Because it was introduced to show the feasibility of the main proposal of [28], block cipher mode COFB, it does not work as a full-fledged AE using conventional TBCs. Therefore, starting from iCOFB, we apply numerous changes for improving efficiency while achieving high security. As a result, Romulus-N becomes a much more advanced, sophisticated NAE mode based on a TBC. The security bound of Romulus-N is essentially equivalent to ΘCB3, having full n-bit security.

Rationale of the MRAE Mode. Romulus-M is designed as an MRAE mode following the structure of SIV [69] and SCT [64]. Romulus-M reuses the components of Romulus-N as much as possible to inherit its implementation advantages and the security. In fact, this brings us several advantages (not only for implementation aspects) over SIV/SCT. Compared with SCT, Romulus-M needs a fewer number of primitive calls thanks to the faster MAC part. In particular, while Romulus-M is a *"two-pass"* mode, it requires only 3 calls to the TBC for every $2n$ bits of the input, making it 50% faster than typical SIV based modes.

Moreover, Romulus-M has a smaller state than SCT because of single-state encryption part taken from Romulus-N (SCT employs a variant of counter mode). The provable security of Romulus-M is equivalent to SCT: the security depends on the maximum number of repetition of a nonce in encryption (r), and if $r = 1$ (*i.e.*, NR adversary) we have the full n-bit security. Security will gradually decreasing as r increases, also known as "graceful degradation", and even if r equals to the number of encryption queries, implying nonces are fixed, we maintain the birthday-bound, $n/2$-bit security. ZAE [45] is another TBC-based MRAE. Although it is faster than SCT, the state size is much larger than SCT and Romulus-M.

Rationale of the leakage-resilient Mode. Romulus-T is a variant of the leakage-resilient mode TEDT [22]. The differences are summarized as follows.

1. TEDT is built upon a TBC with tweak size the same as its block size n, while Romulus-T uses Skinny-128-384+ which enjoys a relatively long $2n$-bit tweak (or input). Benefiting from this,
   - Romulus-T could accept "full" n-bit nonces. In contrast, TEDT accepts nonces of $3n/4$ bits.
   - Romulus-T invokes the rate-1 hash Romulus-H in its second pass, and the involved Hirose's DBL compression function absorbs $2n$-bit blocks and achieves rate 1 (compared with rate 1/2 of TEDT's hashing). Due to this, the overall rate of Romulus-T is 1/3, which is slightly better than TEDT (1/4). Using Romulus-H in a black-box manner also simplifies the whole cipher suite.
2. Importantly, TEDT decryption invokes the inverse of the TBC by default even for unprotected implementations, while Romulus-T defines the unprotected implementations to be inverse-free to enjoy lighter implementations, and the protected implementation use decryption to maintain the side-channel immunity.
3. Romulus-T does not incorporate the "public key" mechanism in TEDT that was used to enhance multi-user security.
4. During encryption, TEDT uses different TBC inputs derived from an arithmetic counter, while Romulus-T separates every pair of TBC calls using different tweaks derived from the more efficient LFSR-based counter.
5. Romulus-T uses a different injective padding for the hashing, which employs an LFSR instead of two arithmetic counters (used to keep $|C|$ and $|A|$ in TEDT) and reduces the hardware cost of padding significantly.

Regarding security, Romulus-T enjoys similar proofs and security bounds as TEDT:

- It ensures $n - \log_2(n)$ bit integrity, and this holds in the nonce misuse setting.
- It ensures $n - \log_2(n)$ bit confidentiality, and this holds in the nonce misuse setting as long as *the nonces used for encrypting confidential messages are never reused* (nonce misuse-*resilience* in the sense of Ashur *et al.* [11]).
- Thanks to the leakage-resilience (or mode-level physical security), it ensures strong side-channel security with a relative low implementation cost. In detail, only the two TBC invocations using the main AE key are needed to be carefully protected against side-channel attacks (*leveled implementation* in the sense of Pereira *et al.* [20, 63]).

Our $n - \log_2(n)$ bit security is slightly better than Berti et al., which is only of $n - 2\log_2(n)$ bits [22]. The reason is that Berti et al.'s $n - 2\log_2(n)$ security is derived in the *multi-user setting*, and the additional $\log_2(n)$ security loss is due to the presence of multiple users. To justify, we have made a document containing the single-user security proof for Romulus-T, which will be made public soon.

We note that TBC-based leakage-resilient AEADs with better efficiency exist, namely TET [22] and AET-LR [30], but they only achieve weaker notions of leakage-resilience. In particular, neither TET nor AET-LR preserves confidentiality against decryption leakages. On the other hand, TEDT and Romulus-T achieved the so-called CIML2 leakage integrity (integrity against full nonce misuse and both encryption and decryption leakages) and CCAmL2 leakage confidentiality notions (confidentiality against both encryption and decryption leakages), which were ranked as the highest leakage security level [20].

Rationale for Hashing Scheme. Romulus-H is an instantiation of MDPH hash function [60], which is a variant of double-block-length (DBL) hashing mode based on the popular scheme [39]

having a dedicated finalization for improved efficiency and achieving indifferentiability notion. It achieves $(n - \log_2 n)$-bit collision and (2nd) preimage resistance thanks to the indifferentiability result. A detailed comparison with other indifferentiable hashing schemes is shown at [60, Table 1], which indicates MDPH achieves a very good balance in security and efficiency. Romulus-H has an excellent performance in terms of the rate – processes $2n$ bits of message per two Skinny calls – and these two calls can be parallel and share the computation of tweakey scheduling. These features are beneficial for both software and hardware, and in particular a 2-block parallel software implementation (via bitslice/fixslice [3, 5, 14]) could be used to boost the software performance.

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

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

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

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

Efficiency Comparison (Romulus-T). For leakage-resilient designs, it is hard to a have a similar fair comparision due to the different assumptions on the algorithms and primitives used. Romulus-T maintains competitive performance in this use-case, where processing one message block requires roughly 3 TBC calls, *i.e.*, it has a rate of 1/3. In particular, the number of TBC calls for an input (N, A, M) is

$$2 \times \lceil \frac{|M|}{n} \rceil + 2 \times \lceil \frac{|U|}{2n} \rceil + 2$$

where n is the block size of the underlying TBC and U is the input to the hash functions, *s.t.*

$$|U| = n \times (\lceil \frac{|A| + 8}{n} \rceil + \lceil \frac{|M| + 8}{n} \rceil) + 184$$

given $|A| \neq 0$ and $|M| \neq 0$. This estimation ignores the special cases of empty A or empty M. Romulus-T requires 4 calls to TBC that are independent of the message length, which are the KDF,

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

Scheme	Number of Primitive Calls	Primitive	Security (λ)	State Size (S)	Rate (R)	S/R	Inverse Free
Romulus-N	$\lceil \frac{\lvert A\rvert -n}{2n} \rceil + \lceil \frac{\lvert M\rvert}{n} \rceil + 1$	$(n, 1.5n, k)$-TBC†, $n = k$	n	$n + 2.5k = 3.5\lambda$	1	3.5λ	Yes
COFB [28]	$\lceil \frac{\lvert A\rvert}{n} \rceil + \lceil \frac{\lvert M\rvert}{n} \rceil + 1$	(n,k)-BC, $n = k$	$n/2 - \log_2 n/2$	$1.5n + k = 5.4\lambda^{\ddagger}$	1	5.4λ	Yes
ΘCB3 [51]	$\lceil \frac{\lvert A\rvert}{n} \rceil + \lceil \frac{\lvert M\rvert}{n} \rceil + 1$	$(n, 1.5n, k)$-TBC$^{\sharp}$, $n = k$	n	$2n + 2.5k = 4.5\lambda$	1	4.5λ	No
Beetle [27]	$\lceil \frac{\lvert A\rvert}{n} \rceil + \lceil \frac{\lvert M\rvert}{n} \rceil + 2$	$2n$-Perm, $n = k$	$n - \log_2 n$	$2n = 2.12\lambda$	1/2	4.24λ	Yes
Ascon-128 [34]	$\lceil \frac{\lvert A\rvert}{n} \rceil + \lceil \frac{\lvert M\rvert}{n} \rceil + 1$	$5n$-Perm, $n = k/2$	$n/2$	$7n = 3.5\lambda$	1/5	17.5λ	Yes
Ascon-128a [34]	$\lceil \frac{\lvert A\rvert}{n} \rceil + \lceil \frac{\lvert M\rvert}{n} \rceil + 1$	$2.5n$-Perm, $n = k$	n	$3.5n = 3.5\lambda$	1/2.5	8.75λ	Yes
SpongeAE $^{\flat}$ [23]	$\lceil \frac{\lvert A\rvert}{n} \rceil + \lceil \frac{\lvert M\rvert}{n} \rceil + 1$	$3n$-Perm, $n = k$	n	$3n = 3\lambda$	1/3	9λ	Yes

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

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

Scheme	Number of Primitive Calls	Primitive	Security (λ)	State Size (S)	Rate (R)	S/R	Inverse Free
Romulus-M	$\lceil \frac{\lvert A\rvert+\lvert M\rvert -n}{2n} \rceil + \lceil \frac{\lvert M\rvert}{n} \rceil + 1$	$(n, 1.5n, k)$-TBC†, $n = k$	n	$n + 2.5k = 3.5\lambda$	1/2	7λ	Yes
SCT ‡ [64]	$\lceil \frac{\lvert A\rvert+\lvert M\rvert}{n} \rceil + \lceil \frac{\lvert M\rvert}{n} \rceil + 1$	(n,n,k)-TBC, $n = k$	n	$4n = 4\lambda$	1/2	8λ	Yes
SUNDAE [12]	$\lceil \frac{\lvert A\rvert+\lvert M\rvert}{n} \rceil + \lceil \frac{\lvert M\rvert}{n} \rceil + 1$	(n,k)-BC, $n = k$	$n/2$	$2n = 4\lambda$	1/2	8λ	Yes
ZAE $^{\sharp}$ [45]	$\lceil \frac{\lvert A\rvert+\lvert M\rvert}{2n} \rceil + \lceil \frac{\lvert M\rvert}{n} \rceil + 6$	(n,n,k)-TBC, $n = k$	n	$7n = 7\lambda$	1/2	14λ	Yes

† Unused part of tweakey is not a part of state thus not considered;
‡ Tag is n bits;
$^{\sharp}$ Tag is $2n$ bits;

TGF and two TBC calls as part of the compression function and responsible for handling N and message length in the hash function. It is worth mentioning that the KDF and TGF are special calls in leakage-resilient implementations that should be implemented independently and are usually much more costly than normal TBC calls, but this is a standard practice in several leakage-resilient modes. In practice, the rate can be improved by leveraging the internal parallelism of the mode.

Rationale of TBC. We chose a member of the Skinny family of tweakable block ciphers [14] as our internal TBC primitives. Skinny was published at CRYPTO 2016 and has received a lot of attention since its proposal. In particular, a lot of third party cryptanalysis has been provided (in part motivated by the organization of cryptanalysis competitions of Skinny by the designers) and this was a crucial point in our primitive choice. Besides, our mode requested a lightweight tweakable block cipher and Skinny is the main such primitive. It is very efficient and lightweight, while providing a very comfortable security margin. Provable constructions that turn a block cipher into a tweakable block cipher were considered, but they are usually not lightweight, not efficient, and often only guarantee birthday-bound security.

6.3 Hardware Implementations

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

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

One of the advantages of Skinny as a lightweight TBC is that it has a very simple datapath, consisting of a simple state register followed by a low-area combinational circuit, where the same circuit is used for all the rounds, so the only multiplexer required is to select between the initial input for the first round and the round output afterwards (Figure 6.1(a)), and it has been shown that this multiplexer can even have lower cost than a normal multiplexer if it is combined with the Flip-Flops by using Scan-Flops (Figure 6.1(b)) [46]. 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 small circuit that costs 320 XOR gates. Moreover, the LFSR computation required between blocks can be implemented on top of this circuit, costing 3 extra XOR gates. This operation can be computed in parallel to ρ, such that when the state is updated for the next block, the tweakey key required is also ready. This costs only ~ 387 XOR gates as opposed to ~ 384 Flip-Flops that will, otherwise, be needed to maintain the tweakey value. Hence, the mode was designed with the architecture in Figure 6.1(b) in mind, where only a full-width state-register is used, carrying the TBC state and tweakey values, and every cycle, it is either kept without change, updated with the TBC round output (which includes a single round of the key scheduling algorithm) or the output of a simple linear transformation, which consists of ρ/ρ^{-1}, the unrolled inverse key schedule and the block counter. In order estimate the hardware cost of Romulus-N the mode we consider the round based implementation with an $n/4$-bit input/output bus:

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

This adds up to 455 XOR gates and 376 multiplexers. For estimation purposes assume an XOR gate costs 2.25 GEs and a multiplexer costs 2.75 GEs, which adds up to $2,057.75$ GEs. In the original Skinny paper [14], the authors reported that Skinny-128-384 requires $4,268$ GEs, which adds up to $\sim 6,325$ GEs. This is ~ 1 KGEs smaller than the round based implementation of Ascon [37]. Moreover, a smart design can make use of the fact that 64 bits of the tweakey of Skinny-128-384 are not used, replacing 64 Flip-Flops by 64 multiplexers reducing an extra ~ 200 GEs. In order to design a combined encryption/decryption circuit, we show below that the decryption costs only extra 32 multiplexers and ~ 32 OR gates, or ~ 100 GEs.[1]

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

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

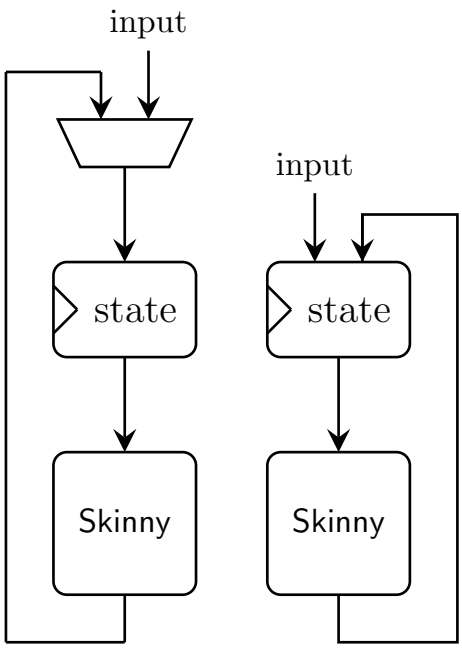
For the serial implementations we followed the currently popular bit-sliding framework [46] with minor tweaks. The state of Skinny is represented as the Feedback-Shift Register which typically operates on 8 bits at a time, while allowing the 32-bit MixColumns operation, given in Figure 6.2

It can be viewed in Figure 6.2 that several careful design choices such as a lightweight serializable ρ function without the need of any extra storage and a lightweight padding/truncation scheme allow the low area implementations to use a very small number of multiplexers on top of the Skinny circuit for the state update, three 8-bit multiplexer to be exact, two of which have a constant zero input, and ~ 22 XORs for the ρ function and block counter. For the key update functions, we did several experiments on how to serialize the operations and we found the best trade-off is to design a parallel/serial register for every tweakey, where the key schedule and mode operations are done in the same manner of the round based implementation, while the AddRoundKey operation of Skinny is done serial as shown in Figure 6.2.

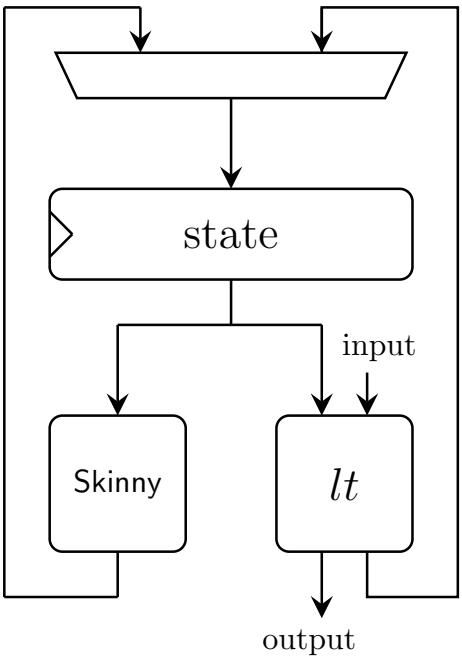
6.4 Software Implementations

We refer to Skinny document for discussions on software implementations of the various Skinny versions. The Romulus mode will have little impact on the global performance of Skinny in software as long as serial implementations are used. We expect very little increase in ROM or RAM when compared to Skinny benchmarks. The very performant 8-bit micro-controller (ATmega644 micro-controller - avr5-core) implementations reported in the Skinny document were benchmarked without assuming parallel cipher calls, and without any pre-processing. Therefore, Romulus will present a very similar performance profile as the numbers reported on micro-controllers. Generally, using little amount of RAM, Skinny is easy and efficient to implement using simple table-based approach.

[1] In earlier versions of this specification, the cost of the tweakey correction circuit was estimated to be 64 XORs instead of 384 XORs. This is because such correction circuit depends on the number of rounds for the underlying TBC. It turns out that correcting the tweakey for 40 rounds is more expensive compared to 56 rounds. However, the gains in terms of speed from the round reduction justifies this design choice and in the overall picture, the correction cost is still small.



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



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

Figure 6.1: Expected architectures for Skinny and Romulus

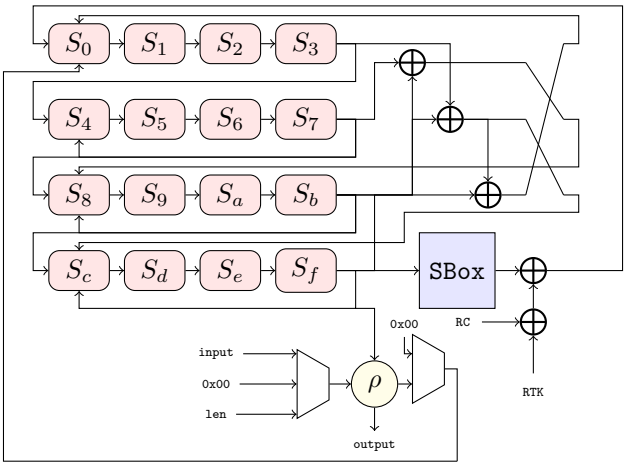


Figure 6.2: Serial State Update Function Used in Romulus

Moreover, Skinny using an 8-bit Sbox and maintaining its byte representation throughout the rounds, it will be very efficient on constrained 8-bit micro-controllers. On larger 32/64-bit platforms, Skinny will remain efficient overall, but might not fully benefit from 32/64-bit instructions.

For high-end platforms, such as latest Intel processors, very efficient highly-parallel bitsliced implementations of Skinny using SSE, AVX, AVX2 instructions on XMM/YMM registers will not be directly applicable as our Romulus mode is serial in nature. However, in the classical case of a server communicating with many lightweight devices, we note that it would be possible to consider bitslicing the key schedule [21] of Skinny (being relatively simple to compute) or using scheduling strategies [25]. Classical table-based implementation of Skinny will ensure acceptable performance on even legacy platforms, while Vector Permute (`vperm`) might lead to better results on medium range platforms by parallelizing the computation of the Sbox.

6.5 Primitives Choices

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

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

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

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

that $C^{'} = \texttt{pad}_n(C) \oplus \texttt{msb}_{n-|C|}(G(S))$. 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.

Padding for Romulus-H The padding function proposed for Romulus-H is similar to the one proposed for Romulus-N and Romulus-M, except for two differences:

1. Since the hash function requires injective padding, the padding function adds an extra all-zero block in case the input consists of full blocks only.
2. The function operates on blocks of 256 bits instead of 128 bits.

Padding for Romulus-T While Romulus-T uses Romulus-H, and inherits its padding function, it is not enough to ensure collision resistance, as the input is not a simple bit-string but a vector of 4 bit strings: A, C, N and $\overline{m}$, where A and C are of variable length, while N and $\overline{m}$ are of fixed length. While a simple injective padding (like the one used in Romulus-H) ensures that two padded strings are equal if and only if their unpadded counterparts are equal, it is not enough when the input is a vector of bit strings. For example, a naive concatenation of $A\|C$ is not immune to cases when $A_1 \neq A_2$ but $A_1\|C_1 = A_2\|C_2$. Since the main reason such cases arise is because A and C are of variable length, their length need to be encoded in the padded string.

The proposed padding is immune to such attacks and is indeed injective. We observe that after the strings $\texttt{ipad}^*_{128}(A)$ and $\texttt{ipad}^*_{128}(C)$ calls have length that is multiple of 128 bits, N is a 128-bit string, and $\overline{m}$ is a 56-bit string. Hence, the last block of the input to Romulus-H is always a partial block that consists of either $N\|\overline{m}$ or $\overline{m}$. Consider two vectors (N_1, A_1, C_2) and (N_2, A_2, C_2) where the last 256 bits after applying $\texttt{ipad}_{256}$ are identical. This implies that $m_1 = m_2$. Hence, $|\texttt{ipad}^*_{128}(C_1)| = |\texttt{ipad}^*_{128}(C_2)|$. From these observations, it is easy to see that the two strings are identical if and only if $A_1 = A_2$, $N_1 = N_2$ and $C_1 = C_2$.

On the other hand, the need for a more sophisticated padding function (compared to Romulus-H) implies an extra overhead in terms of the number of compression function calls. In the worst case, the proposed padding function incurs one extra call compared to Romulus-H. While it might be cheaper to include the byte length (or bit length) of C and do not pad A and C independently, this approach includes extra area overhead and is more complicated to implement in hardware, contradicting our design philosophy. For example, an implementation with a 32-bit input bus would not easily be able to concatenate strings whose length is not multiple of 32-bits. On top of that, counting the bytes instead of the blocks would require more complicated counters and we opt to share the same counter between all Romulus variants.

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

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

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

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

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

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

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

Decryption of Romulus-T. Romulus-T is inverse free "in principle", that is, its decryption could avoid invoking the inverse function of Skinny-128-384+. Though, when side-channel protection is desired, we recommend invoking the inverse of Skinny-128-384+ one time during integrity checking (see line 6 in Romulus-T.$\mathsf{Dec}_K(N, A, C, T)$, Figure 2.9), in order to limit the usefulness of the corresponding side-channel leakages. It can be checked that this implementation-level trick does not affect the correctness of the AE function. We refer to [20, Section 4.3] for a detailed discussion. In summary, our choice ensures both an inverse free implementation in the ordinary setting without side-channel protections and an implementation with strong side-channel security for defense in depth.

7. Implementations

In this section we provide implementations results and estimates. Source codes can be found on our GitHub page: https://github.com/romulusae

7.1 Software Performances

7.1.1 Software Implementations

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

First, the Skinny article reports very efficient 8-bit micro-controller implementations of the Skinny-128-128 version, with various tradeoffs. This is not surprising considering the byte-oriented structure of Skinny versions with 128-bit block. One can moreover mention the good performances and rankings of a simple table-based implementation of Skinny-128-128 in the FELICS benchmarks (https://www.cryptolux.org/index.php/FELICS). Since these implementations do not require any parallelism, they can directly be applied in Romulus.

We have provided new optimised table-based implementations for Romulus, which perform well on 32-bit architectures. Finally, the recent fix-slicing strategy [2, 6], originally applied to GIFT [13] and AES [62] ciphers, turns out to be also applicable to Skinny/Romulus and improve their performances, see [4].

7.1.3 Software Benchmark Efforts

Software Benchmarking by Renner *et al.* [65]. These benchmark results are mainly obtained on five different micro-controller unit platforms. The results are based on the custom made performance evaluation framework, introduced at the NIST LWC Workshop in November 2019. Precisely, the result contains speed, ROM and RAM benchmarks for software implementations of most candidates. We would like to point that, though Romulus is not designed for micro-controllers,

it still performs well among the candidates of the final round of the competition. Notably, among the finalists candidates, Romulus-N ranks at 4th position on 8-bit AVR, a key platform for comparison at is it much more constrained than larger 32-bit micro-controllers. We furthermore note that AES already ranks among the good candidates on 32-bit platforms. The detailed table can be found in [65].

Software Implementations and Benchmarking by Weatherley *et al.* [71]. Rhys Weatherley provided efficient 8-bit AVR and 32-bit ARM Cortex-M3 implementations of Romulus versions, some of them using the fix-slicing strategy [2, 6]. All these implementations are available on the corresponding GitHub repository and benchmarks on these two platforms are provided. We note however that table-based implementations are the best performing for Romulus on 32-bit platforms and these were not taken into account in these benchmarks, thus biasing negatively Romulus performances.

Again, we point that, though Romulus-N is not designed for micro-controllers, among the finalists candidates it again ranks at 5th position on 8-bit AVR, a key platform for comparison at is it much more constrained than larger 32-bit micro-controllers. Even better, Romulus-H hash function is ranking in the top three candidates on 8-bit platforms.

7.2 Hardware Performances

7.2.1 ASIC Performances

In Table 7.1, we give examples of the synthesis results of Romulus-N using the TSMC 65nm standard cell library. The results are consistent with the benchmarks of earlier version of Romulus-N and the expected gains from updating the specifications. Particularly, Romulus-v2 achieves impressive results being only 7.6 kGE with decent performance, while Romulus-v3 and Romulus-v4 achieve high speeds. Given that the earlier version (known as Romulus-N1) already had better performance than AES-GCM for a fraction of the area, we expected the current version of the specifications to pull away even further. In particular, the authors of [1] show that Romulus-v4 has equivalent throughput to the round based implementation of AES-GCM at a fraction of the area (40% smaller). With the speed gains we have shown due to reducing the number of rounds, we expect this implementation to be more than 20% faster than AES-GCM for the same area gain. In other words, even the most speed-oriented implementation of Romulus-N is faster and smaller than a comparable implementation of AES-GCM.

Table 7.1: ASIC Implementations of Romulus-N using the TSMC 65nm standard cell library.

Implementation	Architecture	Area (GE)	Throughput† (Auth. and Enc.) (Gbps)	Throughput† (Enc.) (Gbps)	Throughput† (Auth.) (Gbps)
Romulus-v1	Round-Based	6,668	1.85	1.45	2.65
Romulus-v2	2-Round Unrolled	7,615	3.35	2.65	4.55
Romulus-v3	4-Round Unrolled	9,553	5.1	4.55	7.1
Romulus-v4	8-Round Unrolled	13,518	8.25	7.1	9.85

† Estimated at clock frequency of 500 MHz for long messages.

7.2.2 FPGA Performances

In Table 7.2, we give examples of the implementation results of Romulus-N using the Xilinx Artix-7 FPGA. The results are consistent with the benchmarking results in [58] and the expected gains from updating the specifications. It is worth-mentioning that these implementations are mainly optimized for ASIC and potentially more optimized implementations can improve these results, but

they already show competitive performance compared, ranking in the top half of the finalists list even for the older version for the specifications.

7.2.3 Hardware Benchmark Efforts

During round 2 of the selection process, 3 main hardware benchmarking efforts were performed. While Romulus-N has been part of all three efforts, readers should be careful that the results are for the earlier versions, including 56 rounds instead of 40. Five different implementations of Romulus-N were provided with different area-speed trade-offs. Four of them are round-based implementations with different number of unrolled rounds, while Romulus-v5 is a byte-serial implementation. We observe that while reducing the number of rounds from 56 to 40 leads to an automatic 40% speed-up of Skinny, Romulus-N includes other components whose cost is fixed and the overall gain is expected to be $\leq 40\%$. Besides, the more the implementation is dominated by the speed of Skinny, the closer the speed-up is to 40%, while an implementation that uses a fast TBC architecture (*e.g.* computes 4 or 8 rounds per cycle) will experience less speed-up. Table 7.3 lists the expected gains for throughput and energy. At the same time, the area of all implementations is expected to increase by $3\% \sim 5\%$. For the rest of this section, we will cite benchmarking results based on the earlier version.

Table 7.2: FPGA Implementations of Romulus-N on the Xilinx Artix-7 FPGA.

Implementation	**Goal**	**LUTs**	**Flip-flops**	**Slices**	**Throughput (Enc.)**[‡] (Mbps)
Romulus-v1	High speed	1,030	791	467	716.3
	Low Area	963	500	294	217.5
Romulus-v2	High speed	1,436	536	634	1,369.8
	Low Area	1,156	500	374	400
Romulus-v3	High Speed	2,137	546	800	1,489.4
	Low Area	1,529	500	535	685.7

[†] Estimated at clock frequency of 75 MHz for low area implementations and at maximum achievable frequency for high speed implementations. All cases are for long messages.
[‡] Gains similar to ASIC implementations are expected when AD is included.

Table 7.3: Expected speed-up of various Romulus-N implementations compared to their round 2 counterparts.

Implementation	Skinny **Rounds per cycle**	**Speed-up**	**Energy drop**
Romulus-v1	1	$36\% \sim 35\%$	27%
Romulus-v2	2	$33\% \sim 31\%$	$25\% \sim 23\%$
Romulus-v3	4	$29\% \sim 26\%$	$23\% \sim 21\%$
Romulus-v4	8	$22\% \sim 19\%$	$19\% \sim 16\%$
Romulus-v5	1/23	40%	29%

ASIC Benchmarking by Aagard and Zidaric [1]. In this effort, 24 AEAD algorithms were benchmarked (23 NIST candidates and AES-GCM). The designs were implemented for different technologies: 65nm, 90nm and 130nm. Among the 23 candidates considered, eight are finalists. Their results show that among the considered finalists Romulus-N has the 2^{nd} smallest area, ranks 3^{rd} in terms of throughput and ranks 4^{th} in terms of energy, energy $\times$ area and throughput/Area.

ASIC Benchmarking by Khairallah *et al.* [50]. Khairallah *et al.* considered 10 AEAD algorithms, 6 of which are among the finalists. The designs where synthesized for 65nm and 28nm. Consistent with the other benchmark, Romulus-N achieves the 2^{nd} smallest area, 3^{rd} best throughput and energy $\times$ area and 4^{th} best energy. The authors also considered low-speed applications where the throughput is fixed to a given value for all implementations, rather than the best possible throughput for each implementation. In this case, Romulus-N achieves the 2^{nd} best energy and energy $\times$ area. Finally, the results show that Romulus-N is 1 of only 2 candidates considered that can be efficiently implemented with less than $7,000$ gates.

FPGA Benchmarking by The GMU Hardware Team (Mohajerani *et al.*) [58]. Mohajerani *et al.* implemented 27 round 2 candidates on different FPGAs from different vendors. Among these designs 9 are finalists. Romulus-N has the 2^{nd} smallest area among the 9 finalists on all considered FPGA. Besides, the authors consider two use cases: maximum throughput and fixed frequency at 75MHz. For maximum throughput Romulus-N ranks between 5^{th} and 6^{th} for most metrics. For fixed frequency, Romulus-N ranks 5^{th} in terms of throughput/area and between 3^{rd} and 4^{th} for energy, depending on the message size. In terms of power consumption, the Romulus-N implementation is ultra-low power, having the 2^{nd} lowest power consumption of 68 mW, while the lowest is 64 mW. However, this is an the expense of a very slow implementation. The 2^{nd} best implementation of Romulus-N still consumes very low power and ranks 4^{th} overall.

Discussion. We believe the results of Romulus-N show excellent potential for the algorithm, especially in extremely constrained applications, such as low area, low power and low energy. However, we should point out that our goal is not to get the fastest/smallest design, but the fastest and smallest design within certain security guarantees. In other words, the goal is to find a trade-off between (long-term) security, speed, energy, power and area. Besides, we design Romulus-N in order to be a versatile primitive, you can easily have many implementations of the same algorithm; tiny and slow, big and fast, small and energy-efficient, etc. Different benchmarking results show that Romulus-N is a well-rounded algorithm when it comes to trade-offs, with very high security and large security margin. Besides the ease of combination of Romulus-N and Romulus-M adds another dimension of versatility. The same implementations of Romulus-N can be used for Romulus-M with minor modifications.

Acknowledgments

We would like to thank Yusuke Naito for his feedback on MDPH scheme and Alexandre Adomnicai for his implementations of Romulus on micro-controllers. The third and fifth authors are supported by the Temasek Labs grant (DSOCL16194).

Bibliography

[1] Aagaard, M.D., Zidaric, N.: ASIC Benchmarking of Round 2 Candidates in the NIST Lightweight Cryptography Standardization Process. Cryptology ePrint Archive, Report 2021/049 (2021) https://eprint.iacr.org/2021/049.

[2] Adomnicai, A., Najm, Z., Peyrin, T.: Fixslicing: A New GIFT Representation. IACR Cryptol. ePrint Arch. **2020** (2020) 412

[3] Adomnicai, A., Najm, Z., Peyrin, T.: Fixslicing: A New GIFT Representation Fast Constant-Time Implementations of GIFT and GIFT-COFB on ARM Cortex-M. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2020**(3) (2020) 402–427

[4] Adomnicai, A., Peyrin, T.: Fixslicing - Application to Some NIST LWC Round 2 Candidates. NIST Lightweight Cryptography Workshop 2020 (2020) Available at https://csrc.nist.gov/CSRC/media/Events/lightweight-cryptography-workshop-2020/documents/papers/fixslicing-lwc2020.pdf.

[5] Adomnicai, A., Peyrin, T.: Fixslicing AES-like Ciphers New bitsliced AES speed records on ARM-Cortex M and RISC-V. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(1) (2021) 402–425

[6] Adomnicai, A., Peyrin, T.: Fixslicing AES-like Ciphers New bitsliced AES speed records on ARM-Cortex M and RISC-V. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(1) (2021) 402–425

[7] Andreeva, E., Bogdanov, A., Luykx, A., Mennink, B., Mouha, N., Yasuda, K.: How to Securely Release Unverified Plaintext in Authenticated Encryption. In: ASIACRYPT (1). Volume 8873 of Lecture Notes in Computer Science., Springer (2014) 105–125

[8] Andreeva, E., Bogdanov, A., Luykx, A., Mennink, B., Mouha, N., Yasuda, K.: How to Securely Release Unverified Plaintext in Authenticated Encryption. IACR Cryptol. ePrint Arch. **2014** (2014) 144

[9] Andreeva, E., Mennink, B., Preneel, B.: Security Reductions of the Second Round SHA-3 Candidates. In: ISC. Volume 6531 of Lecture Notes in Computer Science., Springer (2010) 39–53

[10] Armknecht, F., Fleischmann, E., Krause, M., Lee, J., Stam, M., Steinberger, J.P.: The Preimage Security of Double-Block-Length Compression Functions. In Lee, D.H., Wang, X., eds.: Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings. Volume 7073 of Lecture Notes in Computer Science., Springer (2011) 233–251

[11] Ashur, T., Dunkelman, O., Luykx, A.: Boosting Authenticated Encryption Robustness with Minimal Modifications. In Katz, J., Shacham, H., eds.: Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August

20-24, 2017, Proceedings, Part III. Volume 10403 of Lecture Notes in Computer Science., Springer (2017) 3–33

[12] 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

[13] Banik, S., Pandey, S.K., Peyrin, T., Sasaki, Y., Sim, S.M., Todo, Y.: GIFT: A Small Present - Towards Reaching the Limit of Lightweight Encryption. In: Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings. (2017) 321–345

[14] 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 2016 (2). Volume 9815 of Lecture Notes in Computer Science., Springer (2016) 123–153

[15] 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

[16] Beierle, C., Jean, J., Kölbl, S., Leander, G., Moradi, A., Peyrin, T., Sasaki, Y., Sasdrich, P., Sim, S.M.: SKINNY-AEAD and SKINNY-HASH. Submission to NIST Lightweight Cryptography Project (2019)

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

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

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

[20] Bellizia, D., Bronchain, O., Cassiers, G., Grosso, V., Guo, C., Momin, C., Pereira, O., Peters, T., Standaert, F.: Mode-Level vs. Implementation-Level Physical Security in Symmetric Cryptography - A Practical Guide Through the Leakage-Resistance Jungle. In Micciancio, D., Ristenpart, T., eds.: Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part I. Volume 12170 of Lecture Notes in Computer Science., Springer (2020) 369–400

[21] Benadjila, R., Guo, J., Lomné, V., Peyrin, T.: Implementing Lightweight Block Ciphers on x86 Architectures. In: SAC 2013. Volume 8282 of Lecture Notes in Computer Science., Springer (2013) 324–351

[22] Berti, F., Guo, C., Pereira, O., Peters, T., Standaert, F.: TEDT, a Leakage-Resist AEAD Mode for High Physical Security Applications. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2020**(1) (2020) 256–320

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

[24] Black, J., Rogaway, P., Shrimpton, T.: Black-Box Analysis of the Block-Cipher-Based Hash-Function Constructions from PGV. In: CRYPTO. Volume 2442 of Lecture Notes in Computer Science., Springer (2002) 320–335

[25] Bogdanov, A., Lauridsen, M.M., Tischhauser, E.: Comb to Pipeline: Fast Software Encryption Revisited. In Leander, G., ed.: FSE 2015. Volume 9054 of Lecture Notes in Computer Science., Springer (2015) 150–171

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

[27] Chakraborti, A., Datta, N., Nandi, M., Yasuda, K.: Beetle Family of Lightweight and Secure Authenticated Encryption Ciphers. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2018**(2) (2018) 218–241

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

[29] Chakraborti, A., Iwata, T., Minematsu, K., Nandi, M.: Blockcipher-based Authenticated Encryption: How Small Can We Go? (Full version of [28]). IACR Cryptology ePrint Archive **2017** (2017) 649

[30] Chun Guo and Mustafa Khairallah and Thomas Peyrin: AET-LR: Rate-1 Leakage-Resilient AEAD based on the Romulus Family. NIST Lightweight Cryptography Workshop, https://csrc.nist.gov/CSRC/media/Events/lightweight-cryptography-workshop-2020/documents/papers/AET-LR-lwc2020.pdf (2021)

[31] Cogliati, B., Lee, J., Seurin, Y.: New Constructions of MACs from (Tweakable) Block Ciphers. IACR Trans. Symmetric Cryptol. **2017**(2) (2017) 27–58

[32] Coron, J., Dodis, Y., Malinaud, C., Puniya, P.: Merkle-Damgård Revisited: How to Construct a Hash Function. In: CRYPTO. Volume 3621 of Lecture Notes in Computer Science., Springer (2005) 430–448

[33] Demay, G., Gazi, P., Hirt, M., Maurer, U.: Resource-Restricted Indifferentiability. In Johansson, T., Nguyen, P.Q., eds.: Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings. Volume 7881 of Lecture Notes in Computer Science., Springer (2013) 664–683

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

[35] Dong, X., Hua, J., Sun, S., Li, Z., Wang, X., Hu, L.: Meet-in-the-Middle Attacks Revisited: Focusing on Key-recovery and Collision Attacks. Cryptology ePrint Archive, Report 2021/427 (2021) https://eprint.iacr.org/2021/427.

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

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

[38] Handschuh, H., Preneel, B.: Key-Recovery Attacks on Universal Hash Function Based MAC Algorithms. In: CRYPTO 2008. Volume 5157 of Lecture Notes in Computer Science., Springer (2008) 144–161

[39] Hirose, S.: Some Plausible Constructions of Double-Block-Length Hash Functions. In: FSE 2006. Volume 4047 of Lecture Notes in Computer Science., Springer (2006) 210–225

[40] Hirose, S., Park, J.H., Yun, A.: A Simple Variant of the Merkle-Damgård Scheme with a Permutation. In: ASIACRYPT. Volume 4833 of Lecture Notes in Computer Science., Springer (2007) 113–129

[41] Hoang, V.T., Krovetz, T., Rogaway, P.: Robust Authenticated-Encryption AEZ and the Problem That It Solves. In: EUROCRYPT (1). Volume 9056 of Lecture Notes in Computer Science., Springer (2015) 15–44

[42] Iwata, T., Khairallah, M., Minematsu, K., Peyrin, T.: New Results on Romulus. NIST Lightweight Cryptography Workshop 2020 (2006) Available at https://csrc.nist.gov/CSRC/media/Events/lightweight-cryptography-workshop-2020/documents/papers/new-results-romulus-lwc2020.pdf.

[43] Iwata, T., Khairallah, M., Minematsu, K., Peyrin, T.: Duel of the Titans: The Romulus and Remus Families of Lightweight AEAD Algorithms. IACR Cryptology ePrint Archive **2019** (2019) 992

[44] Iwata, T., Khairallah, M., Minematsu, K., Peyrin, T.: Duel of the Titans: The Romulus and Remus Families of Lightweight AEAD Algorithms. IACR Trans. Symmetric Cryptol. **2020**(1) (2020) 43–120

[45] Iwata, T., Minematsu, K., Peyrin, T., Seurin, Y.: ZMAC: A Fast Tweakable Block Cipher Mode for Highly Secure Message Authentication. In: CRYPTO 2017 (3). Volume 10403 of Lecture Notes in Computer Science., Springer (2017) 34–65

[46] Jean, J., Moradi, A., Peyrin, T., Sasdrich, P.: Bit-Sliding: A Generic Technique for Bit-Serial Implementations of SPN-based Primitives - Applications to AES, PRESENT and SKINNY. In: CHES 2017. Volume 10529 of Lecture Notes in Computer Science., Springer (2017) 687–707

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

[48] 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.

[49] Khairallah, M., Chattopadhyay, A., Peyrin, T.: Looting the LUTs: FPGA Optimization of AES and AES-like Ciphers for Authenticated Encryption. In: INDOCRYPT 2017. Volume 10698 of Lecture Notes in Computer Science., Springer (2017) 282–301

[50] Khairallah, M., Peyrin, T., Chattopadhyay, A.: Preliminary Hardware Benchmarking of a Group of Round 2 NIST Lightweight AEAD Candidates. Cryptology ePrint Archive, Report 2020/1459 (2020) https://eprint.iacr.org/2020/1459.

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

[52] 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

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

[54] 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

[55] Maurer, U.M., Renner, R., Holenstein, C.: Indifferentiability, Impossibility Results on Reductions, and Applications to the Random Oracle Methodology. In: TCC. Volume 2951 of Lecture Notes in Computer Science., Springer (2004) 21–39

[56] Medwed, M., Standaert, F., Großschädl, J., Regazzoni, F.: Fresh Re-keying: Security against Side-Channel and Fault Attacks for Low-Cost Devices. In: AFRICACRYPT 2010. Volume 6055 of Lecture Notes in Computer Science., Springer (2010) 279–296

[57] Messerges, T.S.: Securing the AES Finalists Against Power Analysis Attacks. In: FSE 2000. Volume 1978 of Lecture Notes in Computer Science., Springer (2000) 150–164

[58] Mohajerani, K., Haeussler, R., Nagpal, R., Farahmand, F., Abdulgadir, A., Kaps, J.P., Gaj, K.: FPGA Benchmarking of Round 2 Candidates in the NIST Lightweight Cryptography Standardization Process: Methodology, Metrics, Tools, and Results. Cryptology ePrint Archive, Report 2020/1207 (2020) https://eprint.iacr.org/2020/1207.

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

[60] Naito, Y.: Optimally Indifferentiable Double-Block-Length Hashing Without Post-processing and with Support for Longer Key Than Single Block. In: LATINCRYPT. Volume 11774 of Lecture Notes in Computer Science., Springer (2019) 65–85

[61] Naito, Y., Sugawara, T.: Lightweight Authenticated Encryption Mode of Operation for Tweakable Block Ciphers. IACR Cryptology ePrint Archive **2019** (2019) 339

[62] National Institute of Standards and Technology: FIPS 197: Advanced Encryption Standard (November 2001)

[63] Pereira, O., Standaert, F., Vivek, S.: Leakage-Resilient Authentication and Encryption from Symmetric Cryptographic Primitives. In Ray, I., Li, N., Kruegel, C., eds.: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015, ACM (2015) 96–108

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

[65] Renner, S., Pozzobon, E., Mottok, J.: NIST LWC Software Performance Benchmarks on Microcontrollers (2020)

[66] Ristenpart, T., Shacham, H., Shrimpton, T.: Careful with Composition: Limitations of the Indifferentiability Framework. In Paterson, K.G., ed.: Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings. Volume 6632 of Lecture Notes in Computer Science., Springer (2011) 487–506

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

[68] Rogaway, P., Shrimpton, T.: Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance. In: FSE. Volume 3017 of Lecture Notes in Computer Science., Springer (2004) 371–388

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

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

[71] Weatherley, R.: Lightweight Cryptography Primitives (2020)

A. Appendix

Table A.1: Domain separation byte B of Romulus.

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	int(B)	case
Romulus-N	0	0	0	0	1	0	0	0	8	A main
	0	0	0	1	1	0	0	0	24	A last unpadded
	0	0	0	1	1	0	1	0	26	A last padded
	0	0	0	0	0	1	0	0	4	M main
	0	0	0	1	0	1	0	0	20	M last unpadded
	0	0	0	1	0	1	0	1	21	M last padded
Romulus-M	0	0	1	0	1	0	0	0	40	A main
	0	0	1	0	1	1	0	0	44	M auth main
	0	0	1	1	1	1	1	1	63	w: (even,even,padded,padded)
	0	0	1	1	1	1	1	0	62	w: (even,even,padded,unpadded)
	0	0	1	1	1	1	0	1	61	w: (even,even,unpadded,padded)
	0	0	1	1	1	1	0	0	60	w: (even,even,unpadded,unpadded)
	0	0	1	1	1	0	1	1	59	w: (even,odd,padded,padded)
	0	0	1	1	1	0	1	0	58	w: (even,odd,padded,unpadded)
	0	0	1	1	1	0	0	1	57	w: (even,odd,unpadded,padded)
	0	0	1	1	1	0	0	0	56	w: (even,odd,unpadded,unpadded)
	0	0	1	1	0	1	1	1	55	w: (odd,even,padded,padded)
	0	0	1	1	0	1	1	0	54	w: (odd,even,padded,unpadded)
	0	0	1	1	0	1	0	1	53	w: (odd,even,unpadded,padded)
	0	0	1	1	0	1	0	0	52	w: (odd,even,unpadded,unpadded)
	0	0	1	1	0	0	1	1	51	w: (odd,odd,padded,padded)
	0	0	1	1	0	0	1	0	50	w: (odd,odd,padded,unpadded)
	0	0	1	1	0	0	0	1	49	w: (odd,odd,unpadded,padded)
	0	0	1	1	0	0	0	0	48	w: (odd,odd,unpadded,unpadded)
	0	0	1	0	0	1	0	0	36	M enc main
Romulus-T	0	1	0	0	0	0	1	0	66	Key-Derivation Function (KDF)
	0	1	0	0	0	1	0	0	68	Tag Generation Function (TGF)
	0	1	0	0	0	0	0	0	64	TBC calls generating random values to encrypt message blocks
	0	1	0	0	0	0	0	1	65	TBC calls updating internal state value S for each message block

B. Changelog

- 29-03-2019: version v1.0
- 06-06-2019: version v1.01
  - added link to webpage and GitHub
- 22-07-2019: version v1.1
  - added an improved authenticity bound of Romulus-N in Section 4.2, and a corrected and improved nonce-misusing authenticity bound of Romulus-M in Section 4.3. Both improvements are based on the analysis in [61], and the analysis on Romulus-M is also based on [31].
  - added Section 2.4.6 to describe some possible options for a cryptographic hash function based on Skinny.
- 20-09-2019: version v1.2
  - added two paragraphs in Section 6 on how the design choices relate to the serial low-area hardware implementations. Added Figure 6.2.
  - added the synthesis results for the low area implementation in Table 7.1.
- 17-05-2021: version v1.3
  - reorganized the family members: we removed non-primary members from Romulus-N and Romulus-M. The pseudocode of Romulus-N and Romulus-M was simplified to reflect this. No algorithmic change has been made on these modes.
  - Chun Guo (Shandong University, a co-author of TEDT [22]) joined the team
  - added Romulus-H hash function
  - added Romulus-T leakage-resilient AEAD mode
  - Skinny-128-384+ is used as internal primitive (basically Skinny-128-384 reduced to 40 rounds instead of 56, due a very large security margin)
  - Updated the implementations to reflect the new version.
  - Added discussions on third-party benchmarks during round 2.