LOTUS-AEAD and LOCUS-AEAD

Designers/Submitters:

Avik Chakraborti - NTT Secure Platform Laboratories, Japan
Nilanjan Datta - Indian Statistical Institute, Kolkata, India
Ashwin Jha - Indian Statistical Institute, Kolkata, India
Cuauhtemoc Mancillas Lopez - Computer Science Department, CINVESTAV-IPN, Mexico
Mridul Nandi - Indian Statistical Institute, Kolkata, India
Yu Sasaki - NTT Secure Platform Laboratories, Japan

avikchkrbrti@gmail.com, nilanjan_isi_jrf@yahoo.com, ashwin.jha1991@gmail.com, cuauhtemoc.mancillas83@gmail.com, mridul.nandi@gmail.com, sasaki.yu@lab.ntt.co.jp

September 27, 2019

Chapter 1

Introduction

In this document, we propose two new modes of operation, called LOTUS-AEAD and LOCUS-AEAD, for lightweight nonce-based authenticated encryption with associated data (NAEAD) functionality. As the name suggests, LOTUS-AEAD and LOCUS-AEAD follow the general design paradigms of popular NAEAD modes OTR [9] and OCB [8, 7], respectively. However, we remark that LOTUS-AEAD and LOCUS-AEAD introduce several key changes (see section 2.1 for more details) in order to add new features. Some of the important changes include nonce-based rekeying, intermediate checksum based tag generation, short-tweak based domain separation etc.

LOTUS-AEAD and LOCUS-AEAD, achieve higher NAEAD security bounds with lighter primitives. They allow close to 2^{64} data and 2^{128} time limit, when instantiated by a block cipher with 64-bit block and 128-bit key. Notably, they satisfy the NIST lightweight standardization requirements, even with a 64-bit block cipher. To the best of our knowledge, none of the existing NAEAD modes satisfy this criteria while maintaining a state size comparable to LOTUS-AEAD and LOCUS-AEAD. Both LOTUS-AEAD and LOCUS-AEAD are fully parallelizable, and provide integrity security even when the decryption algorithm releases unverified plaintext. These modes are quite versatile, in the sense that, they are equally suitable for memory constrained environments, and high performance applications.

We instantiate LOTUS-AEAD and LOCUS-AEAD with TweGIFT-64, a tweakable variant of the GIFT-64-128 [3] block cipher. TweGIFT-64 is a dedicated design, built upon the original GIFT-64-128 block cipher, for efficient processing of small tweak values of size 4-bit. TweGIFT-64 provides sufficient security while maintaining the lightweight features of GIFT-64-128.

We propose TweGIFT-64_LOTUS-AEAD and TweGIFT-64_LOCUS-AEAD, the TweGIFT-64 based instantiations of LOTUS-AEAD and LOCUS-AEAD, respectively, as our concrete submissions. **We fix TweGIFT-64_LOTUS-AEAD as our primary submission to the standardization process.**

1.1 Notations and Conventions

For $n \in \mathbb{N}$, we write $\{0,1\}^+$ and $\{0,1\}^n$ to denote the set of all non-empty binary[1] strings, and the set of all n-bit binary strings, respectively. We write λ to denote the empty string, and $\{0,1\}^* = \{0,1\}^+ \cup \{\lambda\}$. For $A \in \{0,1\}^*$, $|A|$ denotes the length (number of the bits) of A, where $|\lambda| = 0$ by convention. For all practical purposes, we use the little-endian format for representing binary strings, i.e. the least significant bit is the right most bit. For any non-empty binary string X, $(X_{k-1}, \ldots, X_0) \overset{n}{\leftarrow} x$ denotes the n-bit block parsing of X, where $|X_i| = n$ for $0 \leq i \leq k-2$, and $1 \leq |X_{k-1}| \leq n$. For $A, B \in \{0,1\}^*$ and $|A| = |B|$, we write $A \oplus B$ to denote the bitwise XOR of A and B.

For $n, \tau, \kappa \in \mathbb{N}$, $\widetilde{\mathsf{E}}$-$n/\tau/\kappa$ denotes a tweakable block cipher family $\widetilde{\mathsf{E}}$, parametrized by the block length n, tweak length τ, and key length κ. For $K \in \{0,1\}^\kappa$, $T \in \{0,1\}^\tau$, and $M \in \{0,1\}^n$, we use $\widetilde{\mathsf{E}}_{K,T}(M) := \widetilde{\mathsf{E}}(K,T,M)$ to denote invocation of the encryption function of $\widetilde{\mathsf{E}}$ on input K, T, and M. The decryption function is analogously defined as $\widetilde{\mathsf{E}}^{-1}_{K,T}(M)$. We fix positive even integers n, τ, κ, r, and t to denote the *block size*, *tweak size*, *key size*, *nonce size*, and *tag size*, respectively, in bits. Throughout this document, we fix $n = 64$, $\tau = 4$, and $\kappa = 128$, $r = \kappa$, and $t = n$.

[1]Alphabet set is $\{0,1\}$.

We sometime use the terms (*complete*) *blocks* for n-bit strings, and *partial blocks* for m-bit strings, where $m < n$. Throughout, we use the function ozs, defined by the mapping

$$\forall X \in \bigcup_{m=1}^{n} \{0,1\}^m, \quad X \mapsto \begin{cases} 0^{n-|X|-1} \| 1 \| X & \text{if } |X| < n, \\ X & \text{otherwise,} \end{cases}$$

as the padding rule to map partial blocks to complete blocks. Note that the mapping is injective over partial blocks. For any $X \in \{0,1\}^+$ and $0 \le i \le |X| - 1$, x_i denotes the i-th bit of X. The function chop takes a string X and an integer $i \le |X|$, and returns the least significant i bits of X, i.e. $x_{i-1} \cdots x_0$.

1.1.1 Finite Field Arithmetic

The set $\{0,1\}^\kappa$ can be viewed as the finite field $\mathbb{F}_{2^\kappa}$ consisting of 2^κ elements. We interchangeably think of an element $A \in \mathbb{F}_{2^\kappa}$ in any of the following ways: (i) as a κ-bit string $a_{\kappa-1} \ldots a_1 a_0 \in \{0,1\}^\kappa$; (ii) as a polynomial $A(x) = a_{\kappa-1}x^{\kappa-1} + a_{\kappa-2}x^{\kappa-2} + \cdots + a_1 x + a_0$ over the field $\mathbb{F}_2$; (iii) a non-negative integer $a < 2^\kappa$; (iv) an abstract element in the field. Addition in $\mathbb{F}_{2^\kappa}$ is just bitwise XOR of two κ-bit strings, and hence denoted by $\oplus$. $P(x)$ denotes the primitive polynomial used to represent the field $\mathbb{F}_{2^\kappa}$, and α denotes the primitive element in this representation. The multiplication of $A, B \in \mathbb{F}_{2^\kappa}$ is defined as $A \odot B := A(x) \cdot B(x) \pmod{P(x)}$, i.e. polynomial multiplication modulo $P(x)$ in $\mathbb{F}_2$. For $\kappa = 128$, we fix the primitive polynomial

$$P(x) = x^{128} + x^7 + x^2 + x + 1. \tag{1.1}$$

Then, α, the primitive element, is $2 \in \mathbb{F}_{128}$. It is well-known [10, 7] that multiplication of any field element with α is computationally efficient. For any $A \in \mathbb{F}_{2^{128}}$, we have

$$A \odot \alpha = \begin{cases} A \ll 1 & \text{if } a_{|A|-1} = 0, \\ (A \ll 1) \oplus 0^{120}10000111 & \text{if } a_{|A|-1} = 1. \end{cases}$$

Clearly, we need one shift and one conditional XOR. We refer to this process of multiplying any element $A \in \mathbb{F}_{2^{128}}$ with α, by *α-multiplication*.

Chapter 2

Specification

In this chapter, we present the specifications of LOTUS-AEAD and LOCUS-AEAD along with their underlying block cipher TweGIFT-64. We also give detailed algorithmic descriptions for the modes. Finally, we list the recommended instantiations, TweGIFT-64_LOTUS-AEAD and TweGIFT-64_LOCUS-AEAD.

2.1 LOTUS-AEAD and LOCUS-AEAD Modes

The encryption algorithm of both LOTUS-AEAD and LOCUS-AEAD modes receives an encryption key $K \in \{0,1\}^\kappa$, a nonce $N \in \{0,1\}^\kappa$, an associated data $A \in \{0,1\}^*$, and a message $M \in \{0,1\}^*$ as inputs, and returns a ciphertext $C \in \{0,1\}^{|M|}$, and a tag $T \in \{0,1\}^n$. The decryption algorithm receives a key $K \in \{0,1\}^\kappa$, a nonce $N \in \{0,1\}^\kappa$, an associated data $A \in \{0,1\}^*$, a ciphertext $C \in \{0,1\}^*$, and a tag $T \in \{0,1\}^n$ as inputs, and returns the plaintext $M \in \{0,1\}^{|C|}$ corresponding to C, if T authenticates.

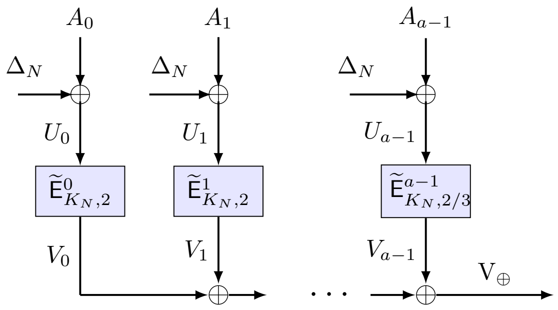
Both LOTUS-AEAD and LOCUS-AEAD operate on n-bit blocks and use a tweakable block cipher $\widetilde{\mathsf{E}}$-$n/\tau/\kappa$ as the underlying primitive. Both the algorithms share a common initialization and associated data processing phase. During the initialization phase, the κ-bit nonce N is XORed with the κ-bit secret key K to generate a κ-bit nonce-dependent encryption key K_N. Then, an n-bit nonce-dependent masking key Δ_N is generated using double encrypting a fixed value (here we have used 0^n) with key K and K_N successively with $\widetilde{\mathsf{E}}$.

2.1.1 Associated Data Processing in LOTUS-AEAD and LOCUS-AEAD

For associated data processing, we parse the data into n-bit blocks and process them in a similar way as as the hash layer of PMAC [5]. To process an associated data block, we first update the current key value via α-multiplication (see section 1.1.1). Next, we XOR the block with Δ_N and encrypt the value using $\widetilde{\mathsf{E}}$ with the fixed tweak 0010 (which we alternatively denote by 2, the integer value corresponding to the binary representation) and the updated key K_N and finally accumulate the encrypted output by XORing it to the previous checksum value. If the final block is partial, we use the tweak 0011 (or denoted by 3) to process the final block. We refer to the output of associated data processing as the AD checksum. Complete description of the associated data processing is depicted in Fig. 2.1 and formally specified in Algorithm 1.

2.1.2 Description of LOTUS-AEAD

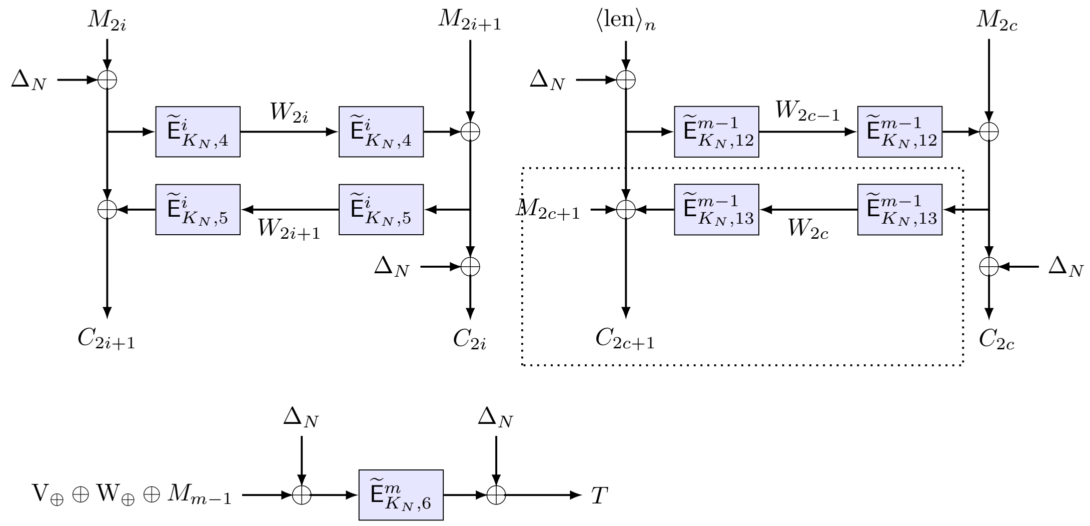
To process a message in LOTUS-AEAD, we parse the data into $2n$-bit di-blocks and process them in a similar manner as OTR [9]. For each message di-block, we apply a simple variant of two-round Feistel. However, instead of one upper layer encryption and one lower layer encryption, here we use two successive encryptions in both upper layer as well in lower layer. The intermediate states in between the encryptions in each layers are used to generate the checksum (that we call intermediate checksum), which helps in obtaining integrity security under RUP setting. To process a di-block, the key is first updated by an α-multiplication (see section 1.1.1) and the same key is used in the four tweakable block cipher $\widetilde{\mathsf{E}}$ calls. However, we use different tweaks in the upper and lower layers (tweak 0100 in the upper layer and 0101 in the lower layer) for the purpose of domain separation. Also, we use two different tweaks 1100 and 1101 during the final di-block processing. The final di-block processing is slightly different and uses the length of the final di-block. To generate the



A_0
A_1
A_{a-1}
Δ_N
U_0
U_1
U_{a-1}
$\widetilde{\mathsf{E}}^{0}_{K_N,2}$
$\widetilde{\mathsf{E}}^{1}_{K_N,2}$
$\widetilde{\mathsf{E}}^{a-1}_{K_N,2/3}$
V_0
V_1
V_{a-1}
$\mathrm{V}_{\oplus}$

Figure 2.1: Associated Data Processing for both LOCUS-AEAD and LOTUS-AEAD. Here $\widetilde{\mathsf{E}}^{i}_{K_N,2}$ denotes invocation of $\widetilde{\mathsf{E}}$ with key $\alpha^{i+1} \odot K_N$ and tweak 0010. For the final associated data block, the use of $\widetilde{\mathsf{E}}^{a-1}_{K_N,2/3}$ indicates invocation of $\widetilde{\mathsf{E}}$ with key $\alpha^{a} \odot K_N$ and tweak 0010 or 0011 depending on whether the final block is full or partial.

tag, we apply XEX [10] like transformation on the XOR of intermediate checksum, AD checksum, and the final message block. Complete specification of LOTUS-AEAD authenticated encryption is given in Algorithm 1. Corresponding verification-decryption algorithm can be found in Algorithm 2. Figure 2.2 gives a pictorial description of the encryption process.



M_{2i}
M_{2i+1}
$\langle \mathrm{len} \rangle_n$
M_{2c}
Δ_N
$\widetilde{\mathsf{E}}^{i}_{K_N,4}$
W_{2i}
$\widetilde{\mathsf{E}}^{i}_{K_N,5}$
W_{2i+1}
$\widetilde{\mathsf{E}}^{m-1}_{K_N,12}$
W_{2c-1}
$\widetilde{\mathsf{E}}^{m-1}_{K_N,13}$
W_{2c}
M_{2c+1}
C_{2i+1}
C_{2i}
C_{2c+1}
C_{2c}
$\mathrm{V}_{\oplus} \oplus \mathrm{W}_{\oplus} \oplus M_{m-1}$
$\widetilde{\mathsf{E}}^{m}_{K_N,6}$
T

Figure 2.2: Processing of an m block message M and Tag Generation for LOTUS-AEAD. The upper left part shows the message processing of an intermediate di-block and the upper right part depicts the message processing of the final di-block. The lower part shows the tag generation process. c denotes the number of di-blocks in the message i.e. $c = \lceil m/2 \rceil - 1$. The dotted part in the final di-block is executed only when the message has even number of blocks. We use the notation $\widetilde{\mathsf{E}}^{i}_{K_N,j}$ to denote invocation of $\widetilde{\mathsf{E}}$ with key $\alpha^{a+i+1} \odot K_N$ and tweak j, where a denotes the number of blocks of associated data corresponding to the message. Here $W_{\oplus}$ denotes the intermediate checksum value and $V_{\oplus}$ denotes the AD checksum value. $\langle \mathrm{len} \rangle_n$ is used to denote the n bit representation of the size of the final di-block in bits.

2.1.3 Description of LOCUS-AEAD

To process a message in LOCUS-AEAD, we parse the data into n-bit blocks and process them in a similar manner as OCB [11]. For each of the message blocks, we first mask the block, then encrypt with the tweakable

Algorithm 1 The encryption algorithm of LOTUS-AEAD.

```
1:  function LOTUS-AEAD_$\widetilde{\mathsf{E}}$.enc$(K, N, A, M)$
2:      $C \leftarrow \perp, W_\oplus \leftarrow 0, V_\oplus \leftarrow 0$
3:      $(K_N, \Delta_N) \leftarrow$ init$(K, N)$
4:      if $|A| \neq 0$ then
5:          $(K_N, V_\oplus) \leftarrow$ proc_ad$(K_N, \Delta_N, A)$
6:      if $|M| \neq 0$ then
7:          $(K_N, W_\oplus, C) \leftarrow$ proc_pt$(K_N, \Delta_N, M)$
8:      $T \leftarrow$ proc_tg$(K_N, \Delta_N, V_\oplus, W_\oplus)$
9:      return $(C, T)$

10: function init$(K, N)$
11:     $Y \leftarrow \widetilde{\mathsf{E}}_{K,0}(0^n)$
12:     $K_N \leftarrow K \oplus N$
13:     $\Delta_N \leftarrow \widetilde{\mathsf{E}}_{K_N,1}(Y)$
14:     return $(K_N, \Delta_N)$

15: function proc_ad$(K_N, \Delta_N, A)$
16:     $L \leftarrow K_N$
17:     $(A_{a-1}, \ldots, A_0) \overset{n}{\leftarrow} A$
18:     for $i = 0$ to $a - 2$ do
19:         $X \leftarrow A_i \oplus \Delta_N$
20:         $L \leftarrow L \odot \alpha$
21:         $V \leftarrow \widetilde{\mathsf{E}}_{L,2}(X)$
22:         $V_\oplus \leftarrow V_\oplus \oplus V$
23:     $X \leftarrow$ ozs$(A_{a-1}) \oplus \Delta_N$
24:     $L \leftarrow L \odot \alpha$
25:     $V \leftarrow (|A_{a-1}| = n)?\ \widetilde{\mathsf{E}}_{L,2}(X) : \widetilde{\mathsf{E}}_{L,3}(X)$
26:     $V_\oplus \leftarrow V_\oplus \oplus V$
27:     return $(L, V_\oplus)$
```

```
1:  function proc_pt$(K_N, \Delta_N, M)$
2:      $L \leftarrow K_N$
3:      $(M_{m-1}, \ldots, M_0) \overset{n}{\leftarrow} M$
4:      $d = \lceil m/2 \rceil$
5:      for $i = 0$ to $d - 2$ do
6:          $j = 2i$
7:          $X_1 \leftarrow M_j \oplus \Delta_N$
8:          $L \leftarrow L \odot \alpha$
9:          $W_1 \leftarrow \widetilde{\mathsf{E}}_{L,4}(X_1)$
10:         $Y_1 \leftarrow \widetilde{\mathsf{E}}_{L,4}(W_1)$
11:         $X_2 \leftarrow Y_1 \oplus M_{j+1}$
12:         $W_2 \leftarrow \widetilde{\mathsf{E}}_{L,5}(X_2)$
13:         $Y_2 \leftarrow \widetilde{\mathsf{E}}_{L,5}(W_2)$
14:         $W_\oplus \leftarrow W_\oplus \oplus W_1 \oplus W_2$
15:         $C_j \leftarrow X_2 \oplus \Delta_N$
16:         $C_{j+1} \leftarrow X_1 \oplus Y_2$
17:     $X_1 \leftarrow \langle |M| - 2(d-1)n \rangle_n \oplus \Delta_N$
18:     $L \leftarrow L \odot \alpha$
19:     $W_1 \leftarrow \widetilde{\mathsf{E}}_{L,12}(X_1)$
20:     $Y_1 \leftarrow \widetilde{\mathsf{E}}_{L,12}(W_1)$
21:     $X_2 \leftarrow Y_1 \oplus M_{2d-2}$
22:     $C_{2d-2} \leftarrow$ chop$(X_2 \oplus \Delta_N, |M_{2d-2}|)$
23:     $W_\oplus \leftarrow W_\oplus \oplus W_1$
24:     $C \leftarrow (C_{2d-2}, \ldots, C_0)$
25:     if $2d = m$ then
26:         $W_2 \leftarrow \widetilde{\mathsf{E}}_{L,13}(X_2)$
27:         $W_\oplus \leftarrow W_\oplus \oplus W_2$
28:         $Y_2 \leftarrow \widetilde{\mathsf{E}}_{L,13}(W_2)$
29:         $C_{2d-1} \leftarrow$ chop$(X_1 \oplus Y_2, |M_{2d-1}|) \oplus M_{2d-1}$
30:         $C \leftarrow C_{2d-1} \| C$
31:     $W_\oplus \leftarrow W_\oplus \oplus M_{m-1}$
32:     return $(L, W_\oplus, C)$

33: function proc_tg$(K_N, \Delta_N, V_\oplus, W_\oplus)$
34:     $L \leftarrow K_N \odot \alpha$
35:     $T \leftarrow \widetilde{\mathsf{E}}_{L,6}(V_\oplus \oplus W_\oplus \oplus \Delta_N) \oplus \Delta_N$
36:     return $T$
```

Algorithm 2 The verification-decryption algorithm of LOTUS-AEAD. The subroutine proc_ad and proc_tag are identical to the one used in the encryption algorithm of LOTUS-AEAD.

```
1:  function LOTUS-AEAD_$\widetilde{\mathsf{E}}$.dec$(K, N, A, C, T)$
2:      $M \leftarrow \perp,\ W_\oplus \leftarrow 0,\ V_\oplus \leftarrow 0$
3:      $(K_N, \Delta_N) \leftarrow$ init$(K, N)$
4:      if $|A| \neq 0$ then
5:          $(K_N, V_\oplus) \leftarrow$ proc_ad$(K_N, \Delta_N, A)$
6:      if $|M| \neq 0$ then
7:          $(K_N, W_\oplus, M) \leftarrow$ proc_ct$(K_N, \Delta_N, C)$
8:      $T' \leftarrow$ proc_tg$(K_N, \Delta_N, V_\oplus, W_\oplus)$
9:      if $T' = T$ then
10:         return $M$
11:     else
12:         return $\perp$
```

```
1:  function proc_ct$(K_N, \Delta_N, C)$
2:      $L \leftarrow K_N$
3:      $(C_{m-1}, \ldots, C_0) \overset{n}{\leftarrow} C$
4:      $d = \lceil m/2 \rceil$
5:      for $i = 0$ to $d-2$ do
6:          $j = 2i$
7:          $L \leftarrow L \odot \alpha$
8:          $X_1 \leftarrow C_j \oplus \Delta_N$
9:          $W_1 \leftarrow \widetilde{\mathsf{E}}_{L,5}(X_1)$
10:         $Y_1 \leftarrow \widetilde{\mathsf{E}}_{L,5}(W_1)$
11:         $X_2 \leftarrow C_{j+1} \oplus Y_1$
12:         $W_2 \leftarrow \widetilde{\mathsf{E}}_{L,4}(X_2)$
13:         $Y_2 \leftarrow \widetilde{\mathsf{E}}_{L,4}(W_2)$
14:         $W_\oplus \leftarrow W_\oplus \oplus W_1 \oplus W_2$
15:         $M_j \leftarrow X_2 \oplus \Delta_N$
16:         $M_{j+1} \leftarrow Y_2 \oplus X_1$
17:     $X_1 \leftarrow \langle |C| - 2(d-1)n \rangle_n \oplus \Delta_N$
18:     $L \leftarrow L \odot \alpha$
19:     $W_1 \leftarrow \widetilde{\mathsf{E}}_{L,12}(X_1)$
20:     $Y_1 \leftarrow \widetilde{\mathsf{E}}_{L,12}(W_1)$
21:     $M_{2d-2} \leftarrow$ chop$(Y_1 \oplus \Delta_N, |C_{2d-2}|) \oplus C_{2d-2}$
22:     $X_2 \leftarrow Y_1 \oplus M_{2d-2}$
23:     $W_\oplus \leftarrow W_\oplus \oplus W_1$
24:     $M \leftarrow (M_{2d-2}, \ldots, M_0)$
25:     if $2d = m$ then
26:         $W_2 \leftarrow \widetilde{\mathsf{E}}_{L,13}(X_2)$
27:         $W_\oplus \leftarrow W_\oplus \oplus W_2$
28:         $Y_2 \leftarrow \widetilde{\mathsf{E}}_{L,13}(W_2)$
29:         $M_{2d-1} \leftarrow$ chop$(X_1 \oplus Y_2, |C_{2d-1}|) \oplus C_{2d-1}$
30:         $M \leftarrow M_{2d-1} \| M$
31:     $W_\oplus \leftarrow W_\oplus \oplus M_{m-1}$
32:     return $(L, W_\oplus, M)$
```

block cipher twice and then again mask to obtain the corresponding ciphertext block. Similar to OTR, the Δ_N masking is same along a query and the intermediate states (W_i in Fig. 2.3) between the two block cipher calls are XORed together to generate the intermediate checksum. For the last message block, instead of applying XEX on the message block, we apply it on the final block message length and XOR the output with the final message block. This strategy ensures identical processing for complete or incomplete final blocks. Again, similar to LOTUS-AEAD, we update the key by α-multiplication (see section 1.1.1) before each block processing, and we use tweak 4 and 5 for non-final and final blocks respectively. The tag is generated identically to that of LOTUS-AEAD. The complete specification of LOCUS-AEAD authenticated encryption and corresponding verification-decryption algorithm is given in Algorithm 3. The message processing part of the encryption algorithm is depicted in Fig. 2.3.

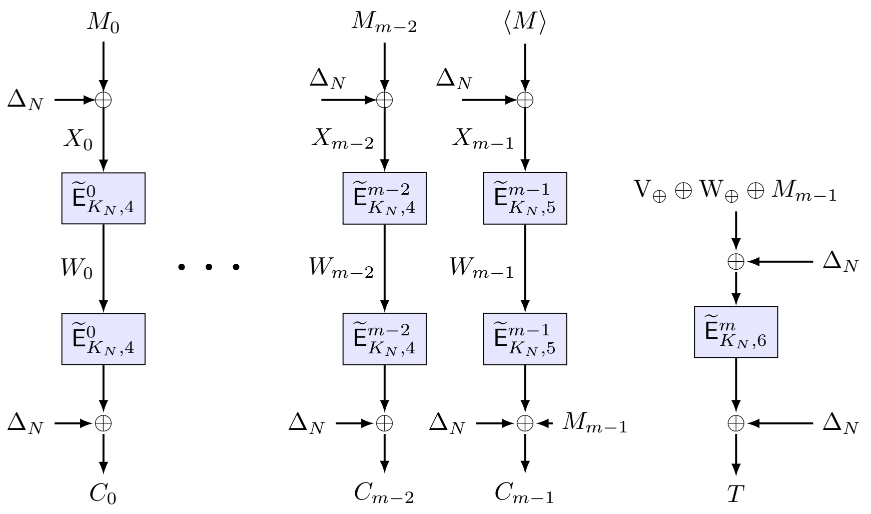

M_0
M_{m-2}
$\langle M \rangle$
Δ_N
X_0
X_{m-2}
X_{m-1}
$\widetilde{\mathsf{E}}^0_{K_N,4}$
$\widetilde{\mathsf{E}}^{m-2}_{K_N,4}$
$\widetilde{\mathsf{E}}^{m-1}_{K_N,5}$
$V_\oplus \oplus W_\oplus \oplus M_{m-1}$
W_0
W_{m-2}
W_{m-1}
$\widetilde{\mathsf{E}}^m_{K_N,6}$
M_{m-1}
C_0
C_{m-2}
C_{m-1}
T

Figure 2.3: Processing of an m block message M and tag generation for LOCUS-AEAD. $\langle \text{len} \rangle_n$ is used to denote the n bit representation of the size of the final block in bits. $W_\oplus$ denotes the intermediate checksum value and $V_\oplus$ denotes the AD checksum value. $\widetilde{\mathsf{E}}^i_{K_N,j}$ is defined in a similar manner as in Fig. 2.2.

2.2 The TweGIFT-64 Tweakable Block Cipher

TweGIFT-64, or more formally TweGIFT-64/4/128, is a 64-bit tweakable block cipher with 4-bit tweak and 128-bit key. As the name suggests, it is a tweakable variant of GIFT-64-128 [3] block cipher. TweGIFT-64 is composed of 28 rounds and each round consists following operations:

SubCells: TweGIFT-64 employs the same invertible 4-bit S-box as GIFT-64-128 and applies it to each nibble of the cipher state. Description of this S-box is given in Table 2.1.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$GS(x)$	1	A	4	C	6	F	3	9	2	D	B	7	5	0	8	E

Table 2.1: The GIFT S-Box GS. Each value is a hexadecimal number.

PermBits: TweGIFT-64 also uses the same bit permutation that was used in GIFT-64-128. This bit mapping is presented in Table 2.2. Note that, this permutation maps bits from bit position i of the cipher state to bit position $GP(i)$, where

$$GP(i) = 4\lfloor i/16 \rfloor + 16\Big(\big(3\lfloor (i \bmod 16)/4 \rfloor + (i \bmod 4)\big) \bmod 4\Big) + (i \bmod 4).$$

Algorithm 3 The encryption and verification-decryption algorithms of LOCUS-AEAD. The subroutine proc_ad and proc_tag are identical to the one used in LOTUS-AEAD.

```
function LOCUS-AEAD_$\widetilde{\mathsf{E}}$.enc$(K, N, A, M)$
    $C \leftarrow \perp,\ W_\oplus \leftarrow 0,\ V_\oplus \leftarrow 0$
    $(K_N, \Delta_N) \leftarrow \mathsf{init}(K, N)$
    if $|A| \neq 0$ then
        $(K_N, V_\oplus) \leftarrow \mathsf{proc\_ad}(K_N, \Delta_N, A)$
    if $|M| \neq 0$ then
        $(K_N, W_\oplus, C) \leftarrow \mathsf{proc\_pt}(K_N, \Delta_N, M)$
    $T \leftarrow \mathsf{proc\_tg}(K_N, \Delta_N, V_\oplus, W_\oplus)$
    return $(C, T)$

function proc_pt$(K_N, \Delta_N, M)$
    $L \leftarrow K_N$
    $(M_{m-1}, \ldots, M_0) \xleftarrow{n} M$
    for $j = 0$ to $m - 2$ do
        $X \leftarrow M_j \oplus \Delta_N$
        $L \leftarrow L \odot \alpha$
        $W \leftarrow \widetilde{\mathsf{E}}_{L,4}(X)$
        $W_\oplus \leftarrow W_\oplus \oplus W$
        $Y \leftarrow \widetilde{\mathsf{E}}_{L,4}(W)$
        $C_j \leftarrow Y \oplus \Delta_N$
    $L \leftarrow L \odot \alpha$
    $X \leftarrow \langle |M_{m-1}| \rangle_n \oplus \Delta_N$
    $W \leftarrow \widetilde{\mathsf{E}}_{L,5}(X)$
    $Y \leftarrow \widetilde{\mathsf{E}}_{L,5}(W)$
    $C_{m-1} \leftarrow \mathsf{chop}(Y \oplus \Delta_N, |M_{m-1}|) \oplus M_{m-1}$
    $W_\oplus \leftarrow W_\oplus \oplus W \oplus M_{m-1}$
    $C \leftarrow (C_{m-1}, \ldots, C_0)$
    return $(L, W_\oplus, C)$
```

```
function LOCUS-AEAD_$\widetilde{\mathsf{E}}$.dec$(K, N, A, C, T)$
    $M \leftarrow \perp,\ W_\oplus \leftarrow 0,\ V_\oplus \leftarrow 0$
    $(K_N, \Delta_N) \leftarrow \mathsf{init}(K, N)$
    if $|A| \neq 0$ then
        $(K_N, V_\oplus) \leftarrow \mathsf{proc\_ad}(K_N, \Delta_N, A)$
    if $|M| \neq 0$ then
        $(K_N, W_\oplus, M) \leftarrow \mathsf{proc\_ct}(K_N, \Delta_N, C)$
    $T' \leftarrow \mathsf{proc\_tg}(K_N, \Delta_N, V_\oplus, W_\oplus)$
    if $T' = T$ then
        return $M$
    else
        return $\perp$

function proc_ct$(K_N, \Delta_N, A, C, T)$
    $L \leftarrow K_N$
    $(C_{m-1}, \ldots, C_0) \xleftarrow{n} C$
    for $j = 0$ to $m - 2$ do
        $Y \leftarrow C_j \oplus \Delta_N$
        $L \leftarrow L \odot \alpha$
        $W \leftarrow \widetilde{\mathsf{E}}^{-1}_{L,4}(Y)$
        $W_\oplus \leftarrow W_\oplus \oplus W$
        $X \leftarrow \widetilde{\mathsf{E}}^{-1}_{L,4}(W)$
        $M_j \leftarrow X \oplus \Delta_N$
    $L \leftarrow L \odot \alpha$
    $X \leftarrow \langle |C_{m-1}| \rangle_n \oplus \Delta_N$
    $W \leftarrow \widetilde{\mathsf{E}}_{L,5}(X)$
    $Y \leftarrow \widetilde{\mathsf{E}}_{L,5}(W)$
    $M_{m-1} \leftarrow \mathsf{chop}(Y \oplus \Delta_N, |C_{m-1}|) \oplus C_{m-1}$
    $W_\oplus \leftarrow W_\oplus \oplus W \oplus M_{m-1}$
    $M \leftarrow (M_{m-1}, \ldots, M_0)$
    return $(L, W_\oplus, M)$
```

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$GP(i)$	0	17	34	51	48	1	18	35	32	49	2	19	16	33	50	3
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$GP(i)$	4	21	38	55	52	5	22	39	36	53	6	23	20	37	54	7
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$GP(i)$	8	25	42	59	56	9	26	43	40	57	10	27	24	41	58	11
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$GP(i)$	12	29	46	63	60	13	30	47	44	61	14	31	28	45	62	15

Table 2.2: The GIST Bit-Permutation GP.

AddRoundKey: In this step, a 32-bit round key is extracted from the master key state and added to the cipher state. This operation is also identical to that of GIFT.

AddRoundConstant: A single bit "1" and a 6-bit round constant are XORed into the cipher state at bit position 63, 23, 19, 15, 11, 7 and 3, respectively. The round constants are generated using the same 6-bit affine LFSR as SKINNY [4] and GIFT-64-128 [3].

AddTweak: For tweak processing, we first expand the 4-bit tweak into a 16-bit codeword using an efficient linear code and then XOR this expanded codeword to the state at an interval of 4 rounds.

The complete specification of TweGIFT-64 is presented in Algorithm 4.

2.3 Recommended Instantiations

We recommend the following concrete instantiations:

- TweGIFT-64_LOTUS-AEAD: This is the NAEAD scheme obtained by instantiating the LOTUS-AEAD mode of operation with TweGIFT-64 block cipher. Here, the key size is 128 bits; nonce size is 128 bits; and tag size is 64 bits. **We also recommend TweGIFT-64_LOTUS-AEAD, due to its inverse-free nature, as the primary version among our submissions.**

- TweGIFT-64_LOCUS-AEAD: This is the NAEAD scheme obtained by instantiating the LOCUS-AEAD mode of operation with TweGIFT-64 block cipher. Here, the key size is 128 bits; nonce size is 128 bits; and tag size is 64 bits.

Algorithm 4 The TweGIFT-64 tweakable block cipher.

```
 1: function TweGIFT($K, t, X$)
 2:     $C \leftarrow 000000$
 3:     for $i = 0$ to $27$ do
 4:         $X \leftarrow$ SubCells($X$)
 5:         $X \leftarrow$ PermBits($X$)
 6:         $(K, X) \leftarrow$ AddRoundKey($K, X$)
 7:         $(C, X) \leftarrow$ AddRoundConstant($C, X$)
 8:         if $i \in \{3, 7, 11, 15, 19, 23\}$ then
 9:             $X \leftarrow$ AddTweak($X, t$)
10:     return $X$

11: function SubCells($X$)
12:     $(x_{15}, \ldots, x_0) \overset{4}{\leftarrow} X$
13:     for $i = 0$ to $15$ do
14:         $x_i \leftarrow GS(x_i)$
15:     return $X$

16: function AddRoundKey($K, X$)
17:     $(k_7, \ldots, k_0) \overset{16}{\leftarrow} K$
18:     $(x_{63}, \ldots, x_0) \overset{1}{\leftarrow} X$
19:     $u_{15..0} \leftarrow k_1$
20:     $v_{15..0} \leftarrow k_0$
21:     for $i = 0$ to $31$ do
22:         $x_{4i+1} \leftarrow x_{4i+1} \oplus u_i$
23:         $x_{4i} \leftarrow x_{4i} \oplus v_i$
24:     $k_7 \| \cdots \| k_0 \leftarrow k_1 \ggg 2 \| k_0 \ggg 12 \| k_7 \| \cdots \| k_2$
25:     return $(K, X)$
```

```
 1: function AddTweak($X, t$)
 2:     $(x_{63}, \ldots, x_0) \overset{1}{\leftarrow} X$
 3:     $(t_3, \ldots, t_0) \overset{1}{\leftarrow} t$
 4:     $t_\oplus \leftarrow t_0 \oplus t_1 \oplus t_2 \oplus t_3$
 5:     for $i = 0$ to $3$ do
 6:         $t_{i+4} \leftarrow t_i \oplus t_\oplus$
 7:     $t_{15..8} \leftarrow t_{7..0}$
 8:     for $i = 0$ to $15$ do
 9:         $x_{4i+2} \leftarrow x_{4i+2} \oplus t_i$
10:     return $X$

11: function PermBits($X$)
12:     $(x_{63}, \ldots, x_0) \overset{1}{\leftarrow} X$
13:     for $i = 0$ to $63$ do
14:         $x_{GP(i)} \leftarrow x_i$
15:     return $X$

16: function AddRoundConstant($C, X$)
17:     $(c_5, \ldots, c_0) \overset{1}{\leftarrow} C$
18:     $(x_{63}, \ldots, x_0) \overset{1}{\leftarrow} X$
19:     $x_{63} \leftarrow x_{63} \oplus 1$
20:     for $i = 0$ to $5$ do
21:         $x_{4i+3} \leftarrow x_{4i+3} \oplus c_i$
22:     $(c_5, \ldots, c_0) \leftarrow (c_4, \ldots, c_0, c_5 \oplus c_4 \oplus 1)$
23:     return $(C, X)$
```

Chapter 3

Security

In this chapter, we summarize the security details of TweGIFT-64_LOTUS-AEAD and TweGIFT-64_LOCUS-AEAD. Section 3.1 gives the concrete data and time limits achieved by the two instantiations. It also lists all the relevant conditions to be adhered in order to maintain adequate security level. Section 3.2 presents a brief analysis against generic attacks (assuming the underlying block cipher is ideal), i.e. the security of modes. Section 3.3 presents a brief analysis on the security of TweGIFT-64, showing that it displays close to ideal behavior under the given data and time limit.

3.1 Security Claims

NAEAD modes	Security Model	Data complexity (in $\log_2$ of bytes)	Time complexity (in $\log_2$)
TweGIFT-64_LOTUS-AEAD	IND-CPA	64	128
TweGIFT-64_LOTUS-AEAD	INT-CTXT (RUP)	64	128
TweGIFT-64_LOCUS-AEAD	IND-CPA	64	128
TweGIFT-64_LOCUS-AEAD	INT-CTXT (RUP)	64	128

Table 3.1: Security levels for TweGIFT-64_LOTUS-AEAD and TweGIFT-64_LOCUS-AEAD. The data and time limits indicate the amount of data and time required to make the attack advantage close to 1.

In Table 3.1, we list the security levels of TweGIFT-64_LOTUS-AEAD and TweGIFT-64_LOCUS-AEAD. We assume a nonce-respecting adversary, i.e. for a fixed key, no pair of distinct encryption queries share the same public nonce value, although we remark that the security may even hold when the public nonce value is sampled uniformly at random from the nonce space for each encryption query. TweGIFT-64_LOTUS-AEAD and TweGIFT-64_LOCUS-AEAD provide integrity security under a stronger model (see section 3.2.3), where the decryption algorithm releases unverified plaintext (RUP model). Here we strongly remark that the same is not true in case of privacy, i.e. we do not claim privacy security in RUP model. All our security claims are based on full round TweGIFT-64, and we do not claim any security for TweGIFT-64_LOTUS-AEAD and TweGIFT-64_LOCUS-AEAD with round-reduced variants of TweGIFT-64.

3.1.1 Statement

We declare that there are no hidden weaknesses in LOTUS-AEAD and LOCUS-AEAD modes of operation. Further, to the best of our knowledge, public third-party analysis do not raise any security threat to the submissions, TweGIFT-64_LOTUS-AEAD and TweGIFT-64_LOCUS-AEAD, within the data and time limit prescribed in Table 3.1.

3.2 Security of **LOTUS-AEAD** and **LOCUS-AEAD**

The two modes are quite identical barring the fact that LOTUS-AEAD processes diblocks of data at a time. So from the point of view of generic attacks similar strategies apply to both LOTUS-AEAD and LOCUS-AEAD.

We describe some possible strategies to attack the LOTUS-AEAD and LOCUS-AEAD modes, and give a rough estimate on the amount of data and time required to mount those attacks. In the following discussion:

- D denotes the data complexity of the attack. This parameter quantifies the online resource requirements, and includes the total number of blocks (among all messages and associated data) processed through the underlying block cipher for a fixed master key. For the sake of simplicity, we also use D to denote the data complexity of forging attempts.

- T denotes the time complexity of the attack. This parameter quantifies the offline resource requirements, and includes the total time required to process the offline evaluations of the underlying block cipher. Since one call of the block cipher can be assumed to take a constant amount of time, we generally take T as the total number of offline calls to the block cipher.

3.2.1 Guessing Key and (Mask)

Guessing the Master Key: The adversary can try to guess the master key using offline block cipher queries. Once the master key is known the adversary can certainly distinguish, forge valid ciphertexts, or worse, recover the plaintext. But, since the master key is chosen uniformly, this attack strategy would require $T \approx 2^{\kappa}$ many offline queries and a constant number of construction queries, i.e. $D = O(1)$.

Guessing the Nonce-based Key and Mask: The adversary can try to guess the nonce-based key and mask for some nonce value using a combination of offline and online queries. Once the key and the mask is known, the adversary can forge valid ciphertexts for this particular nonce value. Note that guessing just one of the key or the mask is not sufficient as the other value is random. Further, guessing both the key and the mask requires the product, $DT \approx 2^{n+\kappa}$. This can be argued using list matching arguments, i.e. the adversary creates a list $\mathcal{L}_T$ of T offline query-response tuples and a list $\mathcal{L}_D$ of D online query-response tuples (with empty plaintext and associated data). It then tries to get a matching between $\mathcal{L}_T$ and $\mathcal{L}_D$, and for each matching it tries an appropriate forging attempt. A matching between any element of $\mathcal{L}_T$ and any element of $\mathcal{L}_D$ would happen with approx. 2^{-192} probability (as the key and mask are random and almost independent of each other). So, we need $DT \approx 2^{n+\kappa}$.

3.2.2 Privacy Security of LOTUS-AEAD and LOCUS-AEAD

We consider the well-established notion of IND-CPA for privacy security. In IND-CPA security model, the adversary is concerned with distinguishing attacks using the output of the encryption algorithm on adversary's choices of input. Note that, the adversary is not allowed to repeat the nonce value, i.e. the adversary is *nonce-respecting*. In addition to the encryption queries, the adversary is also allowed offline queries to the block cipher. A trivial attack strategy is guessing the master key (as discussed in section 3.2.1). Non-trivially, the adversary can distinguish the modes from the ideal if there is no randomness in some ciphertext (or tag) blocks. This is possible in the following two ways:

- Online-Online Block Matching: For a pair of distinct online (in this case encyrption) query block, the triple of key, tweak and input of the underlying block cipher, matches. Then, the block that appears later will have non-random behavior. Note that this matching is only accidental and will happen with probability approx. 2^{-192}. So we need $D^2 \approx 2^{n+\kappa}$.

- Online-Offline Block Matching: For an online query block, the triple key, tweak and input of the underlying block cipher matches with the key, tweak and input of the block cipher in some offline query. Again this matching will happen accidentally with probability approx. 2^{-192}, which gives $DT \approx 2^{n+\kappa}$.

In the absence of above given cases, the two modes behave identically to an ideal nonce-based authenticated encryption scheme.

3.2.3 Integrity Security of LOTUS-AEAD and LOCUS-AEAD in RUP

We consider a stronger variant of the well-known INT-CTXT notion for integrity security. In this case, the adversary is concerned with forging a new and valid (passes verification) ciphertext and tag pair. In classical INT-CTXT security model the adversary is allowed to make encryption queries to the encryption algorithm and forging queries to the decryption algorithm. The decryption algorithm does not release plaintext unless

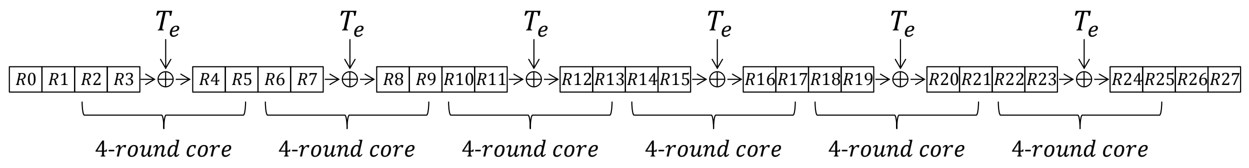


Figure 3.1: Six 4-round cores of TweGIFT-64. T_e is the 16-bit expanded tweak generated from the 4-bit tweak.

the ciphertext and tag pair passes verification. But we allow additional power to the adversary in that the decryption algorithm releases the plaintext, irrespective of the authentication result, i.e. we claim INT-CTXT in RUP model [1]. Note that the release of unverified plaintext does not help the adversary in anyway, as only an encrypted form of the plaintext (which is never released) is used to compute the final tag.

Here, the adversary can apply previous strategies of key and mask guessing as in section 3.2.1, or guessing the internal variables V (encryption of masked AD blocks) or W (decryption of masked ciphertext blocks) using the approach in section 3.2.2; or, try a new strategy of guessing the tag output for each forging attempt. Previous strategies would lead to bounds of the form $T \approx 2^{\kappa}$ and $DT \approx 2^{n+\kappa}$. On the other hand, the tag guessing strategy is similar to guessing a 64-bit random value which would yield $D \approx 2^{n}$.

3.2.4 Validation of Security Claims

The security levels in Table 3.1 can be validated by simply substituting $n = 64$ and $\kappa = 128$.

3.3 Security Analysis of TweGIFT-64

In this section, we provide the security analysis on TweGIFT-64 in the related-key setting. Without exploiting the tweak, TweGIFT-64 offers exactly the same security as the original GIFT-64-128. Hence we focus our attention on the attacks that exploit the tweak injection.

The exact security bound, e.g. the lower bound of the number of active S-boxes and the upper bound of the differential characteristic probability, can be obtained by using various tools based on MILP and SAT, however to derive such bounds for the entire construction with 128-bit key difference is often infeasible.

Here we focus on the feature that the tweak expansion function ensures a large number of active bits at the expanded tweak. This implies that differential trails with non-zero tweak difference will have a relatively large number of active S-boxes around the tweak injection. This motivates us to evaluate the tight bound of the differential characteristic probability for the 2-round transformation followed by the tweak injection and another 2-round transformation, which we call "4-round core". Let p_{core} be the maximum differential characteristic probability of the 4-round core. Then, the probability for the entire construction is upper bounded by $(p_{core})^6$ because 28 rounds of TweGIFT-64 contain six sequence of the 4-round core (Fig. 3.1).

We used the MILP based tool to derive p_{core}. It turned out that the p_{core} is 2^{-16}, hence the maximum differential characteristic probability of 28 rounds can be upper bounded by $2^{-16\times 6} = 2^{-96}$. This can also be viewed that the maximum differential characteristic probability reaches $2^{-16\times 4} = 2^{-64}$ and we have 8 rounds for the margin.

One of the best differential trails for the 4-round core with probability 2^{-16} is fully specified in Table 3.2. The tweak expansion ensures that the number of active bits in T_e is at least 8 when the tweak difference is non-zero. We note that, in the very middle round, both AddRoundKey and AddTweak are performed. However the key bits and tweak bits are XORed to different bit positions of the state. Thus, they cannot cancel each other to avoid affecting the state.

Remarks on Three 8-Round Cores

28 rounds of TweGIFT-64 can also be viewed as containing three of the 8-round core: 4-round transformation followed by the tweak injection and another 4-round transformation. We also evaluated the maximum differential characteristic probability of the 8-round core by using the MILP-based tool, which turned out to be $2^{-26.7}$. Hence, from this evaluation, the probability for the entire construction can be upper bounded only by $2^{3\times -26.7} = 2^{-80.1}$.

Table 3.2: The Best Differential Trail for 4-Round Core of TweGIFT-64.

Round	Mask	Differential Mask	Probability
1	Before SC	0000 0000 0000 0000	1
	After SC	0000 0000 0000 0000	
	Round key	0000 0000 0000 0000	
2	Before SC	0000 0000 0000 0000	1
	After SC	0000 0000 0000 0000	
	Round key	0000 0000 0000 0000	
tweak difference: 8880 0008 8880 0008			
3	Before SC	8880 0008 8880 0008	2^{-16}
	After SC	3330 0003 3330 0003	
	Round key	3030 3030 1212 2121	
4	Before SC	0000 0000 0000 0000	1
	After SC	0000 0000 0000 0000	
	Round key	0000 0000 0000 0000	

This observation demonstrates the difficulties of exploiting our tweak injection in another way. The difficulty of controlling differential trails lies in the heavy weight of the expanded tweak and thus to count as many tweak injection as possible would be the best to derive good bounds.

Chapter 4

Features

Here we succinctly cover the salient features of our proposals:

1. **High Security**: The use of nonce-based encryption key and masking key ensures that both LOTUS-AEAD and LOCUS-AEAD achieve beyond the birthday bound security. In fact, both of them achieve the optimal security level, $DT = O(2^{n+\kappa})$, where D and T denote the data and time complexity, respectively. Here $D < 2^n$, and $T < 2^\kappa$ are obvious conditions.

2. **Lightweight**: To the best of our knowledge, LOTUS-AEAD and LOCUS-AEAD, are the only modes which can achieve the NIST lightweight standardization requirements with 64-bit block ciphers. This reduces the overall state size of the AEAD candidates, when instantiated with ultra-lightweight block ciphers. Our dedicated tweakable block cipher, TweGIFT-64, is a perfectly suitable candidate for this.

3. **High Performance**: LOTUS-AEAD and LOCUS-AEAD preserve the inherent high performance features of OCB and OTR. Both of them are single pass and fully parallelizable. Moreover, LOTUS-AEAD is inverse-free which makes it very efficient in applications, where both encryption and decryption modules are required to be implemented on the same device.

4. **INT-RUP Secure**: Most of the existing block cipher based modes, notably OCB, OTR, COFB [6], SUNDAE [2], do not provide any security in RUP model. This might be an issue in memory-constrained lightweight environments or low-latency real-time streaming protocols. Our proposals solves the problem partially as they provide integrity security under RUP model.

5. **Versatility**: Probably, the single most important feature of our proposals is their scope of applicability. At one end of the spectrum, the parallelizability of LOTUS-AEAD and LOCUS-AEAD make them a perfect candidate for applications in high performance infrastructures. On the other end, their overall state size is competitively small with respect to many existing lightweight candidates, which makes them suitable for low-area hardware implementations.

Chapter 5

Design Rationale

In this section, we briefly describe the various design choices and rationale for our proposals.

5.1 Choice of the Modes: LOTUS-AEAD and LOCUS-AEAD

Our primary goal is to design a lightweight AEAD that should be efficient, provides high performance capability and performs reasonably well in low-end devices as well. For efficiency, the AEAD should be one pass. To obtain high performance capability, we aim for parallelizability. In addition, we demand integrity in RUP model. This is specially useful for memory-constrained lightweight applications.

We start with two well-known modes, namely OCB and OTR. Both OCB and OTR satisfy the first two properties. OCB is online, one-pass and parallelizable. OTR has all these features plus it offers inverse-free feature, albeit in exchange for a larger state (as it works on di-blocks). However, both of them are insecure under the RUP model. This motivates us to design an AE mode which is structurally as simple as OCB and OTR but achieves RUP security while keeping the primary features, such as efficiency and parallelism, intact.

The new proposals LOTUS-AEAD and LOCUS-AEAD replace one block cipher call by two calls. The rationale behind this modification is the observation that the intermediate state between the two block cipher invocations can be used to generate a checksum, which is completely hidden and hence cannot be controlled by the adversary (even if the adversary is allowed to make RUP queries). This hidden checksum ensures integrity security in RUP model. The additional block cipher call per message block increases the number of block cipher calls from ℓ to $2\ell + 1$ to process an ℓ-block message. However, this seems optimal as for any INT-RUP secure parallel AEAD mode, at least $2\ell + 1$ non-linear invocations are necessary to process an ℓ-block message.

5.1.1 Associated Data Processing

The associated data processing phase is based on a simple variant of the hash layer of PMAC, and the computation is completely parallel. The associated data processing can be done in parallel with the plaintext and/or ciphertext processing in order to maximize the performance in parallel computing environments.

5.1.2 Tag Generation

Both OCB and OTR generate the tag using the checksum (simple XORs) of all the plain text blocks and the output of the processed associated data. However, two separate states are required to hold the message checksum and the AD checksum. We obtain INT-RUP security, by using an intermediate checksum (hidden to the adversary) instead of the plaintext checksum. Moreover, we do not store the intermediate checksum and AD checksum separately. Rather, we XOR the two checksums, which means that in a sequential implementations, the intermediate checksum can be computed on top of the AD checksum. This reduces the overall state size by size of one block.

5.1.3 Nonce and Position Dependent Keys

A notable change in LOTUS-AEAD and LOCUS-AEAD is the use of nonce and position dependent keys. OCB and OTR have only birthday bound security on the block size. This is because the security is generally lost once the input/output of any two distinct block cipher calls matches, as the two calls share the same encryption key. In LOTUS-AEAD and LOCUS-AEAD, we overcome the birthday bound barrier by changing the key and tweak pair for each block cipher call. So even if there is a collision among inputs/outputs, the security remains intact, as the block cipher keys or tweaks are distinct. In fact, our modes are secure up to data complexity of 2^n, and time complexity of 2^κ, and combined data-time complexity up to $2^{n+\kappa}$ (see section 3 for more details). This, in turn, helps us to construct AEAD algorithms with the desired security level using an ultra-lightweight block cipher TweGIFT-64.

5.2 Choice of the Tweakable Block Cipher: TweGIFT-64

The main motivation behind TweGIFT-64 is the lack of a good short-tweak tweakable block cipher, which is essential for instantiating our modes LOTUS-AEAD and LOCUS-AEAD. There are some extremely good tweakable block cipher candidates, most notably SKINNY [4], but they are designed to handle general purpose tweaks, and hence not optimized for very short tweaks of size 4 bits. In contrast, TweGIFT-64 is especially designed on top of the GIFT-64-128 block cipher to handle such small tweak values.

5.2.1 Tweak Expansion

For the tweak expansion, we use a simple linear code that converts a 4-bit tweak value into an 8-bit codeword. The linear code is described below:

$$(x_3, x_2, x_1, x_0) \to (S \oplus x_3, S \oplus x_2, S \oplus x_1, S \oplus x_0, x_3, x_2, x_1, x_0), \text{ where } S = x_0 \oplus x_1 \oplus x_2 \oplus x_3.$$

This linear code has two advantages: (i) it is a distance 4 code, and (ii) it is very simple and requires only 7 XOR operations. We use two copies of the codeword to get a 16-bit codeword, which has distance 8. This high distance linear code ensures good differential characteristics for TweGIFT-64 (see section 3.3).

5.2.2 Tweak Injection

We choose to inject the expanded tweak into the block cipher state by masking it to the third bit of each nibble. This bit position has been chosen as the other three positions are already masked by the round key and round constant bits. As shown in section 3.3, tweak injection at intervals of 4 rounds ensures very low differential probability for TweGIFT-64.

Chapter 6

Hardware Implementation

In this chapter, we provide a brief idea on the FPGA implementations of our designs. We first briefly describe our hardware implementation details of the TweGIFT-64 module. We have implemented TweGIFT-64 on Virtex 6 (target device xc6vlx760) using the RTL approach and a basic iterative type architecture. We would like to emphasize that our implementation is round based and it uses 64-bit data path, a smaller implementation can be obtained using smaller datapaths 4-bit, 8-bit, 16-bit or even serialized implementations.

6.1 Implementation of TweGIFT-64

Table 6.1 provides the implementation details of TweGIFT-64 on Virtex 6. It is evident from the results that the difference in the number of LUTs is 119 (caused by the inclusion of the decryption rounds and the multiplexers to select the input to the state register). The difference in terms of the number of slices is about 36 such that one slice in Virtex 6 has 4 LUTs and 2 Flip-flops (depends how a design is optimized and placed by the Xilinx tools).

Table 6.1: TweGIFT-64 Implemented FPGA Results on Virtex 6

Mode	**# Slice Registers**	**# LUTs**	**# Slices**	**Frequency (MHZ)**	**Gbps**	**Mbps/ LUT**	**Mbps/ Slice**
Enc-Dec	273	734	270	425.99	0.94	1.28	3.48
Enc	275	333	134	540.56	1.19	3.57	8.88

Table 6.2: TweGIFT-64 Implemented FPGA Results on Virtex 7

Platform	**# Slice Registers**	**# LUTs**	**# Slices**	**Frequency (MHZ)**	**Gbps**	**Mbps/ LUT**	**Mbps/ Slice**
Enc-Dec	273	730	265	441.71	0.97	1.32	3.66
Enc	275	329	134	554.32	1.22	3.71	9.10

6.2 Implementation of LOCUS-AEAD and LOTUS-AEAD

The hardware implementations of LOCUS-AEAD and LOTUS-AEAD are written in VHDL and are implemented on both Virtex 6 xc6vlx760 and Virtex 7 xc7vx415t. We use the RTL approach and use a basic round based architecture. The areas are provided in terms of the number of slice registers, slice LUTs and the number of occupied slices. The detailed implementation results are depicted in Table 6.3.

Table 6.3: LOCUS-AEAD and LOTUS-AEAD (combined Enc-Dec circuit) Implemented FPGA Results.

Platform	**Scheme**	**# Slice Registers**	**# LUTs**	**# Slices**	**Frequency (MHZ)**	**Throughput (Gbps)**	**Mbps/ LUT**	**Mbps/ Slice**
Virtex 6	LOCUS-AEAD	437	1146	418	348.67	0.39	0.34	0.94
Virtex 7	LOCUS-AEAD	430	1154	439	392.20	0.44	0.38	1.00
Virtex 6	LOCUS-AEAD-Enc	427	698	250	368.34	0.41	0.59	1.65
Virtex 7	LOCUS-AEAD-Enc	424	704	272	406.84	0.46	0.65	1.68
Virtex 6	LOTUS-AEAD	571	868	317	351.25	0.39	0.45	1.24
Virtex 7	LOTUS-AEAD	565	865	317	424.45	0.48	0.55	1.50
Virtex 6	LOTUS-AEAD-Enc	564	801	251	380.84	0.43	0.53	1.70
Virtex 7	LOTUS-AEAD-Enc	564	800	249	414.42	0.47	0.58	1.87
Virtex 6	LOTUS-AEAD-Dec	566	804	245	379.83	0.43	0.53	1.74
Virtex 7	LOTUS-AEAD-Dec	563	791	254	418.91	0.47	0.59	1.85

Bibliography

[1] Elena Andreeva, Andrey Bogdanov, Atul Luykx, Bart Mennink, Nicky Mouha, and Kan Yasuda. How to securely release unverified plaintext in authenticated encryption. *IACR Cryptology ePrint Archive*, 2014:144, 2014.

[2] Subhadeep Banik, Andrey Bogdanov, Atul Luykx, and Elmar Tischhauser. Sundae: Small universal deterministic authenticated encryption for the internet of things. *IACR Transactions on Symmetric Cryptology*, 2018(3):1–35, Sep. 2018.

[3] Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. 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*, pages 321–345, 2017.

[4] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The SKINNY family of block ciphers and its low-latency variant MANTIS. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II*, pages 123–153, 2016.

[5] John Black and Phillip Rogaway. A block-cipher mode of operation for parallelizable message authentication. In *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings*, pages 384–397, 2002.

[6] Avik Chakraborti, Tetsu Iwata, Kazuhiko Minematsu, and Mridul Nandi. Blockcipher-based authenticated encryption: How small can we go? In *CHES 2017*, pages 277–298, 2017.

[7] Ted Krovetz and Phillip Rogaway. The Software Performance of Authenticated-Encryption Modes. In *FSE*, pages 306–327, 2011.

[8] Ted Krovetz and Phillip Rogaway. OCB(v1.1). Submission to CAESAR, 2016. `https://competitions.cr.yp.to/round3/ocbv11.pdf`.

[9] Kazuhiko Minematsu. AES-OTR v3.1. Submission to CAESAR, 2016. `https://competitions.cr.yp.to/round3/aesotrv31.pdf`.

[10] Phillip Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. In *Advances in Cryptology - ASIACRYPT 2004, 10th International Conference on the Theory and Application of Cryptology and Information Security, Jeju Island, Korea, December 5-9, 2004, Proceedings*, pages 16–31, 2004.

[11] Phillip Rogaway, Mihir Bellare, and John Black. OCB: A block-cipher mode of operation for efficient authenticated encryption. *ACM Trans. Inf. Syst. Secur.*, 6(3):365–403, 2003.