GIFT-COFB

v1.0

Designers/Submitters:	**E-mails:**
Subhadeep Banik	subhadeep.banik@epfl.ch
Avik Chakraborti	chakraborti.avik@lab.ntt.co.jp
Tetsu Iwata	tetsu.iwata@nagoya-u.jp
Kazuhiko Minematsu	k-minematsu@ah.jp.nec.com
Mridul Nandi	mridul.nandi@gmail.com
Thomas Peyrin	thomas.peyrin@ntu.edu.sg
Yu Sasaki	sasaki.yu@lab.ntt.co.jp
Siang Meng Sim	crypto.s.m.sim@gmail.com
Yosuke Todo	yosuke.todo.xt@hco.ntt.co.jp

March 29, 2019

Chapter 1

Introduction

Authenticated encryption (AE) is a symmetric-key cryptographic primitive for providing both confidentiality and authenticity. Due to the recent rise in communication networks operated on small devices, the era of the so-called Internet of Things, AE is expected to play a key role in securing these networks.

This document describes GIFT-COFB authenticated, which instantiates the COFB (COmbined FeedBack) block cipher based AEAD mode with the GIFT block cipher. COFB primarily focuses on the hardware implementation size. Here, we consider the overhead in size, thus the state memory size beyond the underlying block cipher itself (including the key schedule) is the criteria we want to minimize, which is particularly relevant for hardware implementation.

An initial version of COFB was presented in [5] and this latest version of COFB is a minor modification over the original COFB mode.

This version supports all the desirable properties mentioned in the NIST lightweight cryptography portfolio [9], and it is efficient for lightweight implementations as well.

There are many approaches of designing a secure and lightweight block cipher based AEAD. We focus on using a lightweight, well analyzed block cipher and minimizing the total encryption/decryption state size. We deploy a hardware optimized block cipher GIFT-128 [2]. In addition to that, we use combined feedback over the block cipher output and the data blocks along with a tweak dependent secret masking (as used in XEX [11]). This combination helps us to minimize the amount of masking by a factor of 2 from [11].

The COFB mode achieves several interesting features. It achieves a high value for rate which is 1 (i.e, needs only one block cipher call for one input block). The mode is inverse-free, i.e., it does not need a block cipher inverse during decryption. In addition to these features, this mode has a quite small state size, namely $1.5n + k$ bits, in case the underlying block cipher has an n-bit block and k-bit keys.

Chapter 2

Specification

2.1 Notation

- For any $X \in \{0,1\}^*$, where $\{0,1\}^*$ is the set of all finite bit strings (including the empty string ϵ), we denote the number of bits of X by $|X|$. Note that $|\epsilon| = 0$.
- For a string X and an integer $t \leq |X|$, $\mathsf{Trunc}_t(X)$ is the first t bits of X.
- Throughout this document, n represents the block size in bits of the underlying block cipher E_K. Typically, we consider $n = 128$ and GIFT-128 is the underlying block cipher, where K is the 128-bit GIFT-128 key.
- For two bit strings X and Y, $X \| Y$ denotes the concatenation of X and Y.
- A bit string X is called a *complete* (or *incomplete*) block if $|X| = n$ (or $|X| < n$ respectively). We write the set of all complete (or incomplete) blocks as $\mathcal{B}$ (or $\mathcal{B}^<$ respectively). Note that, ϵ is considered as an incomplete block and $\epsilon \in \mathcal{B}^<$.
- Given non-empty $Z \in \{0,1\}^*$, we define the parsing of Z into n-bit blocks as

$$(Z[1], Z[2], \ldots, Z[z]) \xleftarrow{n} Z,$$

  where $z = \lceil |Z|/n \rceil$, $|Z[i]| = n$ for all $i < z$ and $1 \leq |Z[z]| \leq n$ such that $Z = (Z[1] \,\|\, Z[2] \,\|\, \cdots \,\|\, Z[z])$. If $Z = \epsilon$, we let $z = 1$ and $Z[1] = \epsilon$. We write $||Z|| = z$ (number of blocks present in Z).
- Given any sequence $Z = (Z[1], \ldots, Z[s])$ and $1 \leq a \leq b \leq s$, we represent the sub sequence $(Z[a], \ldots, Z[b])$ by $Z[a..b]$.
- For integers $a \leq b$, we write $[a..b]$ for the set $\{a, a+1, \ldots, b\}$.

2.1.1 Underlying Finite Field $\mathbb{F}_{2^n}$

Let $\mathbb{F}_{2^s}$ denote the binary Galois field of size 2^s, for a positive integer s. Field addition and multiplication between $a, b \in \mathbb{F}_{2^s}$ are represented by $a \oplus b$ (or $a+b$ whenever understood) and $a \cdot b$ respectively. Any field element $a \in \mathbb{F}_{2^s}$ can be represented by any of the following equivalent ways for $a_0, a_1, \ldots, a_{s-1} \in \{0,1\}$.

- An s-bit string $a_{s-1} \cdots a_0 \in \{0,1\}^s$.
- A polynomial $a(x) = a_0 + a_1 x + \cdots + a_{s-1} x^{s-1}$ of degree at most $(s-1)$.

2.1.2 Choice of Primitive Polynomials

In our construction, the primitive polynomial [1] used to represent the field $\mathbb{F}_{2^{64}}$ is

$$p_{64}(x) = x^{64} + x^4 + x^3 + x + 1.$$

We denote the primitive element $0^{s-2}10 \in \mathbb{F}_{2^s}$ by α_s, (here $s = 64$). We use α to mean α_s for notational simplicity.

64-bit String	Polynomial
$0^{62}10$	α
$0^{62}11$	$\alpha + 1$
$0^{61}100$	α^2

Table 2.1: Various representations of some elements in $\mathbb{F}_{2^{64}}$

Thus, the field multiplication $a(x) \cdot b(x)$ is the polynomial $r(x)$ of degree at most $(s-1)$ such that $a(x)b(x) \equiv r(x) \mod p_s(x)$.

Multiplication by Primitive Element α. We first see an example how we can multiply by α_{64}. Multiplying an element $b := b_{63}b_{62} \cdots b_0 \in \mathbb{F}_{2^{64}}$ by the primitive element α_{64} of $\mathbb{F}_{2^{64}}$ can be done very efficiently as follows:

$$b \cdot \alpha_{64} = \begin{cases} b \ll 1, & \text{if } a_{63} = 0 \\ (b \ll 1) \oplus 0^{59}11011, & \text{else,} \end{cases}$$

where $\ll r$ denotes left shift by r bits. Throughout this document, we use α to denote α_{64}. For, $b \in \mathbb{F}_{2^{64}}$, we use $2 \cdot b$ (or $2^m \cdot b$) and $3 \cdot b$ (or $3^m \cdot b$) to denote $\alpha \cdot b$ (or $\alpha^m \cdot b$) and $(1+\alpha) \cdot b$ (or $(1+\alpha)^m \cdot b$) respectively.

2.2 Recommended Parameter Choice

We propose a construction GIFT-COFB with the underlying block cipher as the only parameter. The block cipher can be chosen by the following recommendation.

1. n: Length of the block cipher state in bits. The recommended choice is $n = 128$.
2. τ: Length of the tag in bits. The recommended choice is $\tau = 128$.
3. E_K: The recommended choice of E_K is the block cipher GIFT-128.

2.3 Input and Output Data

To encrypt a message M with associated data A and nonce N, one needs to provide the information given below.

The encryption algorithm takes as input

- An encryption key $K \in \{0,1\}^{128}$.
- A nonce $N \in \{0,1\}^{128}$. This can include the counter to make the nonce non-repeating.
- Associated data and message $A, M \in \{0,1\}^*$.

It generates the following output data:

- Ciphertext $C \in \{0,1\}^{|M|}$.
- Tag $T \in \{0,1\}^{128}$

To decrypt (with verification) a ciphertext-tag pair (C, T) with associated data A and nonce N, one needs to provide the information given below.

- An encryption key $K \in \{0,1\}^{128}$.
- A nonce $N \in \{0,1\}^{128}$.
- Associated data and ciphertext $A, C \in \{0,1\}^*$.
- Tag $T \in \{0,1\}^{128}$

It generates the following output data:

- Message $M \in \{0,1\}^{|C|} \cup \{\perp\}$, where $\perp$ is a special symbol denoting rejection.

2.4 Mathematical Components

2.4.1 Block cipher GIFT-128

GIFT-128 is an 128-bit Substitution-Permutation network (SPN) based block cipher with a key length of 128-bit. It is a 40-round iterative block cipher with

identical round function. There are two versions of GIFT, namely GIFT-64 and GIFT-128. But since we are focusing only on GIFT-128 in this document, we use GIFT and GIFT-128 interchangeably. For the rest of this document, we take the full version of GIFT paper [3] as reference.

There are different ways to perceive GIFT-128, the more pictorial description is detailed in Section 2 of [3], which looks like a larger version of PRESENT cipher with 32 4-bit S-boxes and an 128-bit bit permutation (see Figure 2.1). In this document, we will be using bitslice description which is similar to Appendix A of [3].

Round function

Each round of GIFT consists of 3 steps: SubCells, PermBits, and AddRoundKey.

Initialization. The 128-bit plaintext is loaded into the cipher state S which will be expressed as 4 32-bit segments. In the perspective of a 2-dimensional array, the bit ordering is from top-down, then right to left.

$$S = \begin{bmatrix} S_0 \\ S_1 \\ S_2 \\ S_3 \end{bmatrix} \leftarrow \begin{bmatrix} b_{124} & \cdots & b_8 & b_4 & b_0 \\ b_{125} & \cdots & b_9 & b_5 & b_1 \\ b_{126} & \cdots & b_{10} & b_6 & b_2 \\ b_{127} & \cdots & b_{11} & b_7 & b_3 \end{bmatrix}.$$

The 128-bit secret key is loaded into the key state KS partitioned into 8 16-bit words. In the perspective of a 2-dimensional array, the bit ordering is from right to left, then bottom-up.

$$KS = \begin{bmatrix} W_0 & \| & W_1 \\ W_2 & \| & W_3 \\ W_4 & \| & W_5 \\ W_6 & \| & W_7 \end{bmatrix} \leftarrow \begin{bmatrix} b_{127} & \cdots & b_{112} & \| & b_{111} & \cdots & b_{98} & b_{97} & b_{96} \\ b_{95} & \cdots & b_{80} & \| & b_{79} & \cdots & b_{66} & b_{65} & b_{64} \\ b_{63} & \cdots & b_{48} & \| & b_{47} & \cdots & b_{34} & b_{33} & b_{32} \\ b_{31} & \cdots & b_{16} & \| & b_{15} & \cdots & b_2 & b_1 & b_0 \end{bmatrix}$$

Refer to Section 2.4.2 for details of the arriving data.

SubCells. Update the cipher state with the following instructions:

$$\begin{aligned} S_1 &\leftarrow S_1 \oplus (S_0 \ \& \ S_2) \\ S_0 &\leftarrow S_0 \oplus (S_1 \ \& \ S_3) \\ S_2 &\leftarrow S_2 \oplus (S_0 \mid S_1) \\ S_3 &\leftarrow S_3 \oplus S_2 \\ S_1 &\leftarrow S_1 \oplus S_3 \\ S_3 &\leftarrow \sim S_3 \\ S_2 &\leftarrow S_2 \oplus (S_0 \ \& \ S_1) \\ \{S_0, S_1, S_2, S_3\} &\leftarrow \{S_3, S_1, S_2, S_0\}, \end{aligned}$$

where &, | and $\sim$ are `AND`, `OR` and `NOT` operation respectively.

PermBits. Different 32-bit bit permutations are applied to each S_i independently.

Table 2.2: Specifications of GIFT-128 bit permutation.

Index	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
S_0	29	25	21	17	13	9	5	1	30	26	22	18	14	10	6	2
S_1	30	26	22	18	14	10	6	2	31	27	23	19	15	11	7	3
S_2	31	27	23	19	15	11	7	3	28	24	20	16	12	8	4	0
S_3	28	24	20	16	12	8	4	0	29	25	21	17	13	9	5	1

Index	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S_0	31	27	23	19	15	11	7	3	28	24	20	16	12	8	4	0
S_1	28	24	20	16	12	8	4	0	29	25	21	17	13	9	5	1
S_2	29	25	21	17	13	9	5	1	30	26	22	18	14	10	6	2
S_3	30	26	22	18	14	10	6	2	31	27	23	19	15	11	7	3

In Table 2.2, the row "Index" shows the indexing of the 32 bits in all S_i's and the row "S_i" shows the ending position of the bits. For example, bit 1 (the 2nd rightmost bit) of S_1 is shifted 1 position to the right, to the initial position of bit 0, while bit 0 is shifted 8 positions to the left.

AddRoundKey. This step consists of adding the round key and round constant. Two 32-bit segments U, V are extracted from the key state as the round key.

$$RK = U \| V.$$

For the addition of round key, U and V are XORed to S_2 and S_1 of the cipher state respectively.

$$S_2 \leftarrow S_2 \oplus U,$$
$$S_1 \leftarrow S_1 \oplus V.$$

For the addition of round constant, S_3 is updated as follows,

$$S_3 \leftarrow S_3 \oplus \texttt{0x800000XY},$$

where the byte $\texttt{XY} = 00c_5c_4c_3c_2c_1c_0$.

Key schedule and round constants

A round key is *first* extracted from the key state before the key state update. Four 16-bit words of the key state are extracted as the round key $RK = U\|V$.

$$U \leftarrow W_2\|W_3,\ V \leftarrow W_6\|W_7.$$

The key state is then updated as follows,

$$\begin{bmatrix} W_0 & \| & W_1 \\ W_2 & \| & W_3 \\ W_4 & \| & W_5 \\ W_6 & \| & W_7 \end{bmatrix} \leftarrow \begin{bmatrix} W_6 \ggg 2 & \| & W_7 \ggg 12 \\ W_0 & \| & W_1 \\ W_2 & \| & W_3 \\ W_4 & \| & W_5 \end{bmatrix},$$

where $\ggg i$ is an i bits right rotation within a 16-bit word.

The round constants are generated using the a 6-bit affine LFSR, whose state is denoted as $c_5c_4c_3c_2c_1c_0$. Its update function is defined as:

$$c_5\|c_4\|c_3\|c_2\|c_1\|c_0 \leftarrow c_4\|c_3\|c_2\|c_1\|c_0\|c_5 \oplus c_4 \oplus 1.$$

The six bits are initialized to zero, and updated *before* being used in a given round. The values of the constants for each round are given in the table below, encoded to byte values for each round, with c_0 being the least significant bit.

Rounds	Constants
1 - 16	`01,03,07,0F,1F,3E,3D,3B,37,2F,1E,3C,39,33,27,0E`
17 - 32	`1D,3A,35,2B,16,2C,18,30,21,02,05,0B,17,2E,1C,38`
33 - 48	`31,23,06,0D,1B,36,2D,1A,34,29,12,24,08,11,22,04`

Decryption of GIFT-128

We omit the description of the inverse of GIFT-128 as it is not required for GIFT-COFB.

2.4.2 Format of Incoming Data

As seen in the "Initialization" phase, the loading of the data (plaintext) bits is column-wise. Typically, that would require additional instructions to rearrange and pack the incoming data into the S_i's, and unpack them back to the initial data format after the encryption. Such practice, however, is merely a matter of perspective and does not affect the security. In fact, it costs additional clock cycles in software implementation to pack them into the desired format. To save on this unnecessary overhead, we regard the incoming data and key as having

the desired format and load them into the states in the most natural manner.

$$S = \begin{bmatrix} S_0 \\ S_1 \\ S_2 \\ S_3 \end{bmatrix} \leftarrow \begin{bmatrix} B_0 & \| & B_1 & \| & B_2 & \| & B_3 \\ B_4 & \| & B_5 & \| & B_6 & \| & B_7 \\ B_8 & \| & B_9 & \| & B_{10} & \| & B_{11} \\ B_{12} & \| & B_{13} & \| & B_{14} & \| & B_{15} \end{bmatrix},$$

$$KS = \begin{bmatrix} W_0 & \| & W_1 \\ W_2 & \| & W_3 \\ W_4 & \| & W_5 \\ W_6 & \| & W_7 \end{bmatrix} \leftarrow \begin{bmatrix} B_0\|B_1 & \| & B_2\|B_3 \\ B_4\|B_5 & \| & B_6\|B_7 \\ B_8\|B_9 & \| & B_{10}\|B_{11} \\ B_{12}\|B_{13} & \| & B_{14}\|B_{15} \end{bmatrix},$$

where B_i are the arriving bytes.

Relation to GIFT-128 LUT based implementation

An alternative implementation of GIFT is using look-up table (LUT) for the SubCells operation. Such implementation prefers having the data in the conventional format, i.e. $B_0B_1 \cdots B_{15} = b_{127}b_{126} \cdots b_1b_0$.

The conversion from an LUT implementation to our bitslice implementation is simple: Note that we perceive the incoming data as bitslice format,

$$\begin{bmatrix} B_0 & \| & B_1 & \| & B_2 & \| & B_3 \\ B_4 & \| & B_5 & \| & B_6 & \| & B_7 \\ B_8 & \| & B_9 & \| & B_{10} & \| & B_{11} \\ B_{12} & \| & B_{13} & \| & B_{14} & \| & B_{15} \end{bmatrix}$$

$$= \begin{bmatrix} b_{124}b_{120}b_{116}\cdots b_{96} & \| & b_{92}\cdots b_{64} & \| & b_{60}\cdots b_{32} & \| & b_{28}\cdots b_0 \\ b_{125}b_{121}b_{117}\cdots b_{97} & \| & b_{93}\cdots b_{65} & \| & b_{61}\cdots b_{33} & \| & b_{29}\cdots b_1 \\ b_{126}b_{122}b_{118}\cdots b_{98} & \| & b_{94}\cdots b_{66} & \| & b_{62}\cdots b_{34} & \| & b_{30}\cdots b_2 \\ b_{127}b_{123}b_{119}\cdots b_{99} & \| & b_{95}\cdots b_{67} & \| & b_{63}\cdots b_{35} & \| & b_{31}\cdots b_3 \end{bmatrix}.$$

First, unpack the data into the conventional format. Next, perform the LUT implementation of GIFT. Finally, pack the output data back to the bitslice format. No additional packing/unpacking is required for the key. This yields the exact same bitslice implementation as we described in the Section 2.4.1.

Test Vectors

Key	:	`00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F`
Plaintext	:	`00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F`
Ciphertext	:	`A9 4A F7 F9 BA 18 1D F9 B2 B0 0E B7 DB FA 93 DF`

Key	:	`E0 84 1F 8F B9 07 83 13 6A A8 B7 F1 92 F5 C4 74`
Plaintext	:	`E4 91 C6 65 52 20 31 CF 03 3B F7 1B 99 89 EC B3`
Ciphertext	:	`33 31 EF C3 A6 60 4F 95 99 ED 42 B7 DB C0 2A 38`

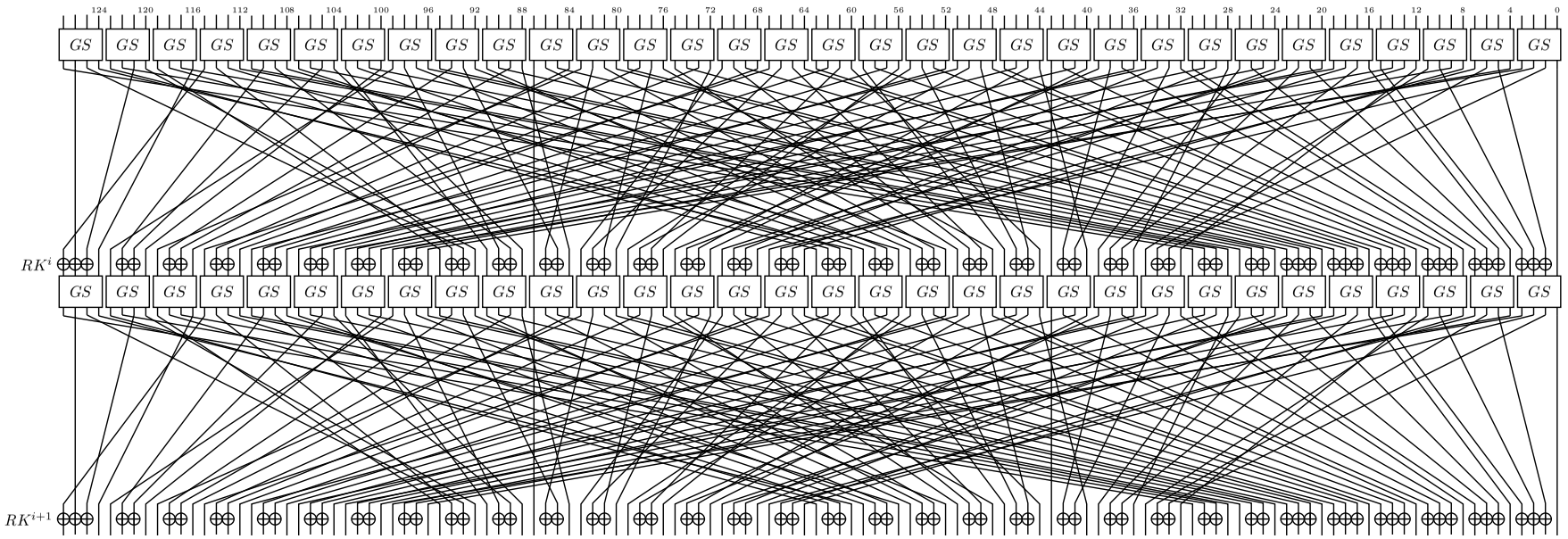

Figure 2.1: 2 rounds of GIFT-128.

2.5 COFB Authenticated Encryption Mode

In this section, we present our proposed mode, COFB in Fig. 2.3. We first specify the basic building blocks and parameters used in our construction.

Key and Block cipher. The underlying cryptographic primitive is an n-bit block cipher, E_K. We assume that n is a multiple of 4. The key of the scheme is the key of the block cipher, i.e. K.

Padding Function. For $x \in \{0,1\}^*$, we define padding function Pad as

$$\mathsf{Pad}(x) = \begin{cases} x & \text{if } x \neq \epsilon \text{ and } |x| \bmod n = 0 \\ x \| 10^{(n-(|x| \bmod n)-1)} & \text{otherwise.} \end{cases} \tag{2.1}$$

Feedback Function. Let $Y \in \{0,1\}^n$ and $(Y[1], Y[2]) \xleftarrow{n/2} Y$, where $Y[i] \in \{0,1\}^{n/2}$. We define $G : \mathcal{B} \to \mathcal{B}$ as

$$G(Y) = (Y[2], Y[1] \lll 1),$$

where for a string X, $X \lll r$ is the left rotation of X by r bits. We also view G as the $n \times n$ non-singular matrix, so we write $G(Y)$ and $G \cdot Y$ interchangeably. For $M \in \mathcal{B}$ and $Y \in \mathcal{B}$, we define $\rho_1(Y, M) = G \cdot Y \oplus M$. The feedback function ρ and its corresponding ρ' are defined as

$$\begin{aligned} \rho(Y, M) &= (\rho_1(Y, M), Y \oplus M), \\ \rho'(Y, C) &= (\rho_1(Y, Y \oplus C), Y \oplus C). \end{aligned}$$

Note that when $(X, M) = \rho'(Y, C)$ then $X = (G \oplus I) \cdot Y \oplus C$, where I is the $n \times n$ identity matrix. Our choice of G ensures that $G \oplus I$ has rank $n-1$. When Y is chosen randomly for both computations of X (through ρ and ρ'), X also behaves randomly. We need this property when we bound probability of bad events later.

We present the specifications of COFB in Fig. 2.3, where α and $(1+\alpha)$ are written as 2 and 3. See also Fig. 2.2. The encryption and decryption algorithms are denoted by COFB-$\mathcal{E}_K$ and COFB-$\mathcal{D}_K$. We remark that the nonce length is n bits, which is enough for the security up to the birthday bound. The nonce is processed as $E_K(N)$ to yield the first internal chaining value. The encryption algorithm takes A and M, and outputs C and T such that $|C| = |M|$ and $|T| = n$. The decryption algorithm takes (N, A, C, T) and outputs M or $\perp$.

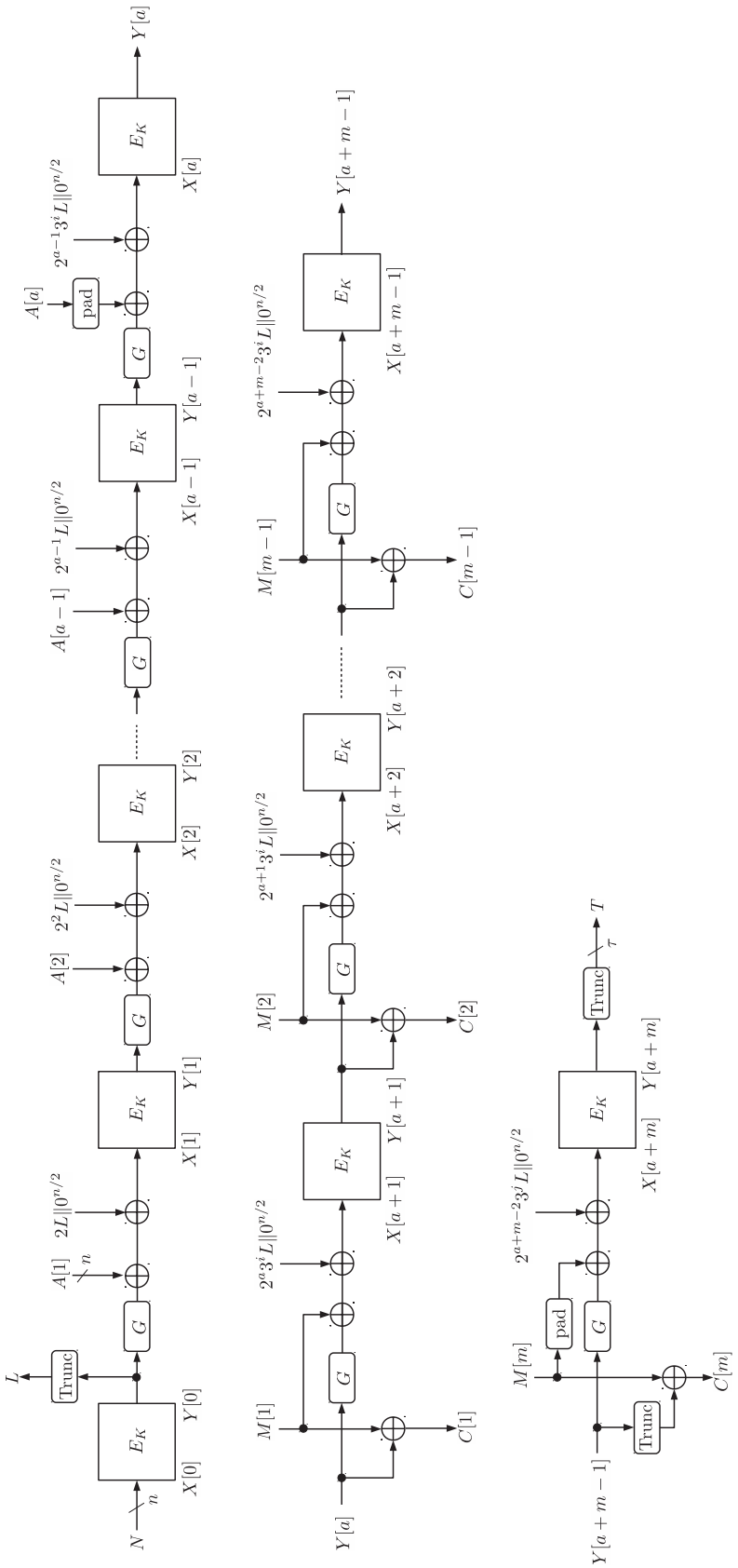

Figure 2.2: Encryption of COFB. In the rightmost figure, the case of encryption for empty M (hence a MAC for (N, A)) can be highlighted as $T = \mathsf{Trunc}_\tau(Y[a])$

Algorithm COFB-$\mathcal{E}_K(N, A, M)$

1. $Y[0] \leftarrow E_K(N),\ L \leftarrow \mathsf{Trunc}_{n/2}(Y[0])$
2. $(A[1], \ldots, A[a]) \xleftarrow{n} \mathsf{Pad}(A)$
3. **if** $M \neq \epsilon$ **then**
4. $\quad (M[1], \ldots, M[m]) \xleftarrow{n} \mathsf{Pad}(M)$
5. **for** $i = 1$ **to** $a - 1$
6. $\quad L \leftarrow 2 \cdot L$
7. $\quad X[i] \leftarrow A[i] \oplus G \cdot Y[i-1] \oplus L \| 0^{n/2}$
8. $\quad Y[i] \leftarrow E_K(X[i])$
9. **if** $|A| \bmod n = 0$ **and** $A \neq \epsilon$ **then** $L \leftarrow 3 \cdot L$
10. **else** $L \leftarrow 3^2 \cdot L$
11. **if** $M = \epsilon$ **then** $L \leftarrow 3^2 \cdot L$
12. $X[a] \leftarrow A[a] \oplus G \cdot Y[a-1] \oplus L \| 0^{n/2}$
13. $Y[a] \leftarrow E_K(X[a])$
14. **for** $i = 1$ **to** $m - 1$
15. $\quad L \leftarrow 2 \cdot L$
16. $\quad C[i] \leftarrow M[i] \oplus Y[i + a - 1]$
17. $\quad X[i + a] \leftarrow M[i] \oplus G \cdot Y[i + a - 1] \oplus L \| 0^{n/2}$
18. $\quad Y[i + a] \leftarrow E_K(X[i + a])$
19. **if** $M \neq \epsilon$ **then**
20. $\quad$ **if** $|M| \bmod n = 0$ **then** $L \leftarrow 3 \cdot L$
21. $\quad$ **else** $L \leftarrow 3^2 \cdot L$
22. $\quad C[m] \leftarrow M[m] \oplus Y[a + m - 1]$
23. $\quad X[a+m] \leftarrow M[m] \oplus G \cdot Y[a+m-1] \oplus L \| 0^{n/2}$
24. $\quad Y[a + m] \leftarrow E_K(X[a + m])$
25. $\quad C \leftarrow \mathsf{Trunc}_{|M|}(C[1] || \ldots || C[m])$
26. $\quad T \leftarrow \mathsf{Trunc}_{\tau}(Y[a + m])$
27. **else** $C \leftarrow \epsilon$, $T \leftarrow \mathsf{Trunc}_{\tau}(Y[a])$
28. **return** (C, T)

Algorithm COFB-$\mathcal{D}_K(N, A, C, T)$

1. $Y[0] \leftarrow E_K(N),\ L \leftarrow \mathsf{Trunc}_{n/2}(Y[0])$
2. $(A[1], \ldots, A[a]) \xleftarrow{n} \mathsf{Pad}(A)$
3. **if** $C \neq \epsilon$ **then**
4. $\quad (C[1], \ldots, C[c]) \xleftarrow{n} \mathsf{Pad}(C)$
5. **for** $i = 1$ **to** $a - 1$
6. $\quad L \leftarrow 2 \cdot L$
7. $\quad X[i] \leftarrow A[i] \oplus G \cdot Y[i-1] \oplus L \| 0^{n/2}$
8. $\quad Y[i] \leftarrow E_K(X[i])$
9. **if** $|A| \bmod n = 0$ **and** $A \neq \epsilon$ **then** $L \leftarrow 3 \cdot L$
10. **else** $L \leftarrow 3^2 \cdot L$
11. **if** $C = \epsilon$ **then** $L \leftarrow 3^2 \cdot L$
12. $X[a] \leftarrow A[a] \oplus G \cdot Y[a-1] \oplus L \| 0^{n/2}$
13. $Y[a] \leftarrow E_K(X[a])$
14. **for** $i = 1$ **to** $c - 1$
15. $\quad L \leftarrow 2 \cdot L$
16. $\quad M[i] \leftarrow Y[i + a - 1] \oplus C[i]$
17. $\quad X[i + a] \leftarrow M[i] \oplus G \cdot Y[i + a - 1] \oplus L \| 0^{n/2}$
18. $\quad Y[i + a] \leftarrow E_K(X[i + a])$
19. **if** $C \neq \epsilon$ **then**
20. $\quad$ **if** $|C| \bmod n = 0$ **then**
21. $\quad\quad L \leftarrow 3 \cdot L$
22. $\quad\quad M[c] \leftarrow Y[a + c - 1] \oplus C[c]$
23. $\quad$ **else**
24. $\quad\quad L \leftarrow 3^2 \cdot L,\ c' \leftarrow |C| \bmod n$
25. $\quad\quad M[c] \leftarrow \mathsf{Trunc}_{c'}(Y[a+c-1] \oplus C[c]) \| 10^{n-c'-1}$
26. $\quad X[a + c] \leftarrow M[c] \oplus G \cdot Y[a + c - 1] \oplus L \| 0^{n/2}$
27. $\quad Y[a + c] \leftarrow E_K(X[a + c])$
28. $\quad M \leftarrow \mathsf{Trunc}_{|C|}(M[1] || \ldots || M[c])$
29. $\quad T' \leftarrow \mathsf{Trunc}_{\tau}(Y[a + c])$
30. **else** $M \leftarrow \epsilon$, $T' \leftarrow \mathsf{Trunc}_{\tau}(Y[a])$
31. **if** $T' = T$ **then return** M, **else return** $\perp$

Figure 2.3: The encryption and decryption algorithms of COFB

Chapter 3

Performance

3.1 Hardware Performance

The COFB mode was designed with rate 1, that is every message block is processed only once. Such designs are not only beneficial for throughput, but also energy consumption. However the design does need to maintain an additional 64 bit state, which requires a 64-bit register to additionally included in any hardware circuit that implements it. Although this might not be energy efficient for short messages, in the long run COFB performs excellently with respect to energy consumption. The GIFT block cipher was designed with a motivation for good performance on lightweight platforms. The roundkey additon for the cipher is over only half the state and the keyschedule being only a bit permutation does not require logic gates. These characteristics make the GIFT well suited for lightweight applications. In fact as reported in [2], among the block ciphers defined for 128-bit block size GIFT-128 has the lowest hardware footprint and very low energy consumption. Thus GIFT-COFB combines the best of both the advantages of the design ideologies.

Figure 3.1 details the hardware circuit for round based GIFT-COFB. The mode is designed to require one additional 64-bit state apart from the ones used in the block cipher circuit. Thus the design requires an additional 64-bit register. The initial nonce (denoted by *Nonce* in the above figure) to the encryption routine, and other control signals are generated centrally depending on the length of the plaintext and associated data. Depending on the phase of operation the state register may need to feed either the nonce, the output of the GIFT-128 round function, which is the sum of the encryption output, associated data/plaintext and the additional state *Delta*.

The state *Delta* is updated by multiplying with suitable filed elements of the form $\gamma = \alpha^x(1+\alpha)^y$ with $x+y \leq 4$. Thus we allocate 4 clock cycles to compute the potential Delta update signal. Depending on the value of γ, we update the *Delta* register by either doubling, tripling or the identity operation. For example if $\gamma = \alpha^2$, we execute doubling for 2 cycles and the identity operation

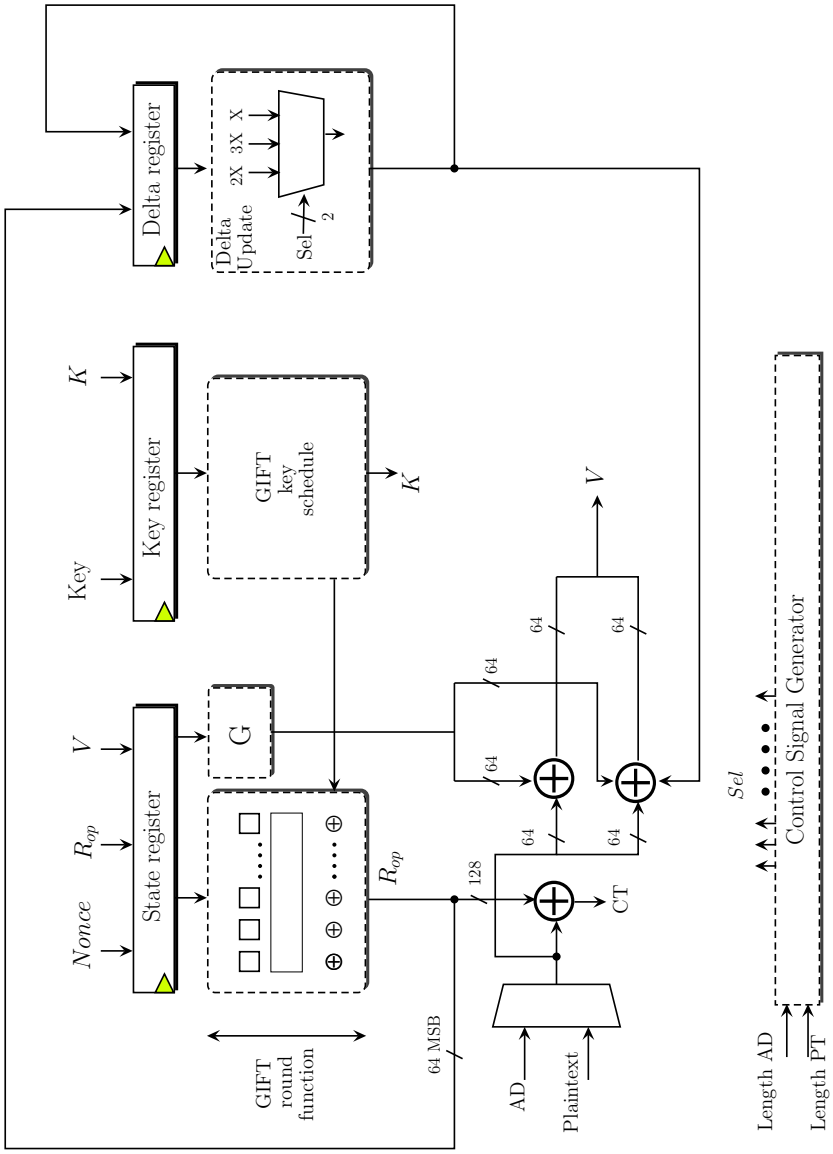

Figure 3.1: Hardware circuit for round based GIFT-COFB

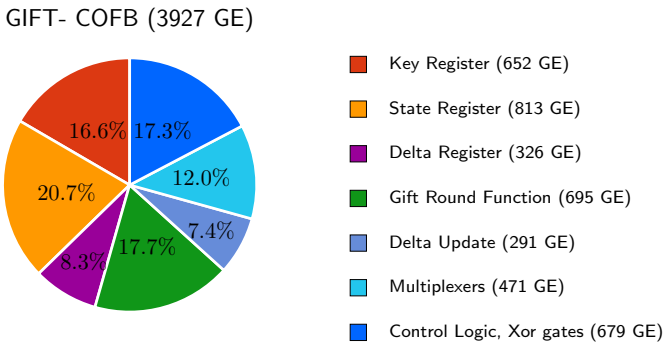

Figure 3.2: Component-wise breakup of the GIFT-COFB circuit

for 2 more cycles. Thus in addition to the field operation, the circuit requires a 3:1 multiplexer controlled by a Sel signal generated centrally.

3.1.1 Timing

The GIFT-128 block cipher takes $E = 40$ cycles to complete one encryption function. This is the number of clock cycles required in the encryption of the nonce. Each block of associated data would take E cycles to process. Before each block of associated data or plaintext is processed we spend $D_u = 4$ cycles to update the $Delta$. Thus if n_a, n_m are the total number of associated data/ message blocks an encryption pass requires $T = E + (n_a + n_m)(E + D_u)$ cycles to compute.

3.1.2 Performance

We present the synthesis results for the design. The following design flow was used: first the design was implemented in VHDL. Then, a functional verification was first done using Mentor Graphics Modelsim software. The designs were synthesized using the standard cell library of the 90nm logic process of STM (CORE90GPHVT v2.1.a) with the Synopsys Design Compiler, with the compiler being specifically instructed to optimize the circuit for area. A timing simulation was done on the synthesized netlist. The switching activity of each gate of the circuit was collected while running post-synthesis simulation. The average power was obtained using Synopsys Power Compiler, using the back annotated switching activity.

Our implementation of GIFT-COFB occupied 3927 GE. A component-wise breakup of the circuit is given in Figure 3.2. The power consumed at an operating frequency is 156.3 μW. The energy consumption figures for various lengths of data inputs are given in Table 3.1.

Block Cipher	Area (GE)	Power(μW)	Energy(nJ)					
			AD	PT	AD	PT	AD	PT
			0B	16B	16B	16B	16B	32B
GIFT-128	3927	156.3	1.31		2.00		2.69	

Table 3.1: Implementation results for GIFT-COFB. (Power reported at 10 MHz)

3.1.3 Performance in FPGA

A comprehensive study of implementation of the GIFT cipher on various FPGA platforms (of the Spartan 6 and Artix 7 families) was done in [8]. The authors reported three possible architectures of GIFT: round-based, and two types of serial architectures that operates using different widths of datapaths. The first serial architecture (referred to as Serial-1) uses 8-bit input/output ports for data loading/unloading and has a serialized application of the substitution layer based on two 4-bit S-boxes. Thus the substitution layer of GIFT-128 would take 128/8=16 cycles in Serial-1. The permutation layer and key addition is performed in the 17th clock cycle much like the PRESENT architecture in [12], resulting in an encryption latency of $16 + 40 * 17 + 16 = 712$ cycles.

The second serial architecture (referred to as Serial-2) uses 32-bit input/output ports for data loading/unloading and has a serialized application of the substitution layer based on eight 4-bit. S-boxes. Thus 4 clock cycles are required for the substitution layer. This implementation takes advantage of the fact that the GIFT-128 permutation function can be written as the composition of a columnwise permutation and a transposition function. The strategy therefore is to compute at the same time the columnwise S-box, key addition and permutation functions in the 4 cycles allotted for the substitution function. The 5-th cycle is used for the matrix transposition operation, resulting in a 5-cycle round and a total latency of $4 + 5 * 40 + 4 = 208$ cycles.

In this submission we have tweaked slightly the format of the incoming data, and thus the implementations reported in [8] needs to be tweaked slightly, only for the Serial-1 and Serial-2 architectures. Namely we will need to spend one extra cycle per round to permute the arrangement of bits of the incoming bitslice format to the conventional format before applying round function operations. This incurs only a 40 cycle penalty in the encryption latency.

3.1.4 Threshold Implementation

The algebraic degree of the GIFT S-box is 3 (same as PRESENT) and as such constructing threshold circuits is slightly more difficult than for quadratic S-boxes. However threshold implementations of the round-based GIFT-128 circuit has been extensively studied in [6]. Since the S-box is cubic, the number of

direct shares it must be decomposed to needs to be at least 4. However, the authors in [6] report three philosophies.

The first decomposes the S-box as the composition $F \circ G$ of two quadratic S-boxes F, G, and implements each decomposed S-box using 3 shares with a register separating the two shared implementations, as in [10]. The shares of both G, F being algebraically similar to each other, and differing only in the order of input bits, the authors further apply an optimization due to [7], that reduces the area of the circuit by implementing the shares over 3 cycles, using a multiplexer to permute the order of bits each time.

The second is a direct sharing approach using 4 shares, and a third approach proposed by them uses only 3 shares for strictly resource-constrained platforms. In total the authors propose 9 different threshold circuits for GIFT-128, depending on whether the key is shared or not, and the type of circuit optimization used. The circuits were synthesized using the TSMC low power 65 nm standard cell library. The smallest implementation uses 3 shares and 256 random bits and occupies around 13349 GE and is around 5.38 times the size of the unprotected circuit. The largest implementation uses 4 direct shares for both the key and datapath and occupies around 94 kGE. For more results, we refer the reader to [6].

Chapter 4

Security of GIFT-COFB

Our security claims are summarized in Table 4.1.

Construction	State Size(bits)	IND-CPA(bits)	INT-CTXT(bits)
GIFT-COFB	192 (excluding the key state)	64	58

Table 4.1: IND-CPA and INT-CTXT security of GIFT-COFB under the nonce respecting scenario

4.1 IND-CPA Security of GIFT-COFB

To attack against the privacy of GIFT-COFB, we assume that an adversary runs in time t and makes at most q_e encryption queries $(N_i, A_i, M_i)_{i=1...q_e}$ to GIFT-COFB with an aggregate of total σ_e many blocks. In return the adversary receives $(C_i, T_i)_{i=1...q_e}$. In this interaction, the adversary tries to distinguish the construction from a random function with the same domain and range.

If we use a hybrid argument, then we first make a transition by using an n-bit (uniform) random permutation P instead of the underlying block cipher GIFT_K, and then to use an n-bit (uniform) random function R instead of P. This two-step transition requires the first two terms of our bound, from the standard PRP-PRF switching lemma and from the computation to the information security reduction (e.g., see [18]). Then what we need is a bound for COFB using R, denoted by COFB[R].

The adversary can distinguish COFB[R] construction from a random function with the same domain and range if it finds a state collision among the internal states (block cipher inputs) of two encryption queries. It is easy to see that the probability of a collision is bounded by $\frac{\binom{\sigma_e}{2}}{2^{128}}$. This holds as for any two of the σ_e block cipher inputs (corresponding to σ_e input data blocks) are equal with the probability 2^{-128} (from the randomness of the previous block cipher outputs).

Hence, the privacy or IND-CPA advantage of GIFT-COFB can be bounded by $\mathbf{Adv}^{\text{prp}}_{\text{GIFT}}(q_e, t) + \frac{\binom{q_e}{2}}{2^{128}} + \frac{\binom{\sigma_e}{2}}{2^{128}}$.

4.2 IND-CTXT Security of GIFT-COFB

On the other hand, to attack against the integrity of GIFT-COFB, assume that an adversary makes at most q_e encryption queries $(N_i, A_i, M_i)_{i=1\ldots q_e}$ to COFB$-\mathcal{E}_K$ with an aggregate of total σ_e many blocks. In return the adversary receives $(C_i, T_i)_{i=1\ldots q_e}$. The adversary also tries to forge with q_f decryption queries $(N^*_j, A^*_j, C^*_j, T^*_j)_{j=1\ldots q_f}$ with a total number of σ_f blocks to COFB $- \mathcal{D}_K$ and receives $M^*_j or\ \perp$. Let $q = q_e + q_f + \sigma_e + \sigma_f$. The trivial solution for forging is to guess the tag which can be bounded by $\frac{q_f}{2^{128}}$ (One of the q_f forged tags is valid).

A bad case **B1** occurs if an adversary can obtain an intermediate block cipher input state collision between an encryption query and a decryption query or between two decryption queries. The probability of this event is bounded by $\frac{(q_e+\sigma_e+2\sigma_f)\sigma_f}{2^{128}} + \frac{64q_f}{2^{64}}$ (Actually the last term is $\frac{0.5nq_f}{2^{64}}$ and here $n = 128$).

To bound the probability of **B1**, we assume the following bad events do not hold. The bad events are as follows.

- **B2**: Multicollision of size more than $n/2$ (with $n = 128$) on the right half of the intermediate block cipher inputs for the encryption queries do not occur. This event is bounded by a negligible probability $\frac{2\sigma_e}{2^{64}}$.
- **B3**: Let $X_i[j]$ be the j^{th} block cipher input in the i^{th} encryption query and $X^*_i[j]$ be the j^{th} block cipher input in the i^{th} decryption query. For each of the decryption queries, after the prefix p_i (defined in footnote[1]), we define the following event **B3** as

  $$X^*_i[p_i + 1] = N_j \text{ and } X^*_i[p_i + 2] = X_{i'}[j'], \text{ for some } i, j, i', j'.$$

  This event is bounded by a negligible probability $\frac{q_f}{2^{64}} + \frac{q_f(q_e-1)}{2^{128}}$.

The part $\frac{(q_e+\sigma_e+2\sigma_f)\sigma_f}{2^{128}}$ in **B1** occurs for block cipher input state collision between an encryption and an decryption query or between two decryption queries. Here, the number of such bad pairs is bounded by $(q_e + \sigma_e + 2\sigma_f)\sigma_f$.

The part $\frac{64q_f}{2^{64}}$ in **B1** occurs for block cipher input state collision between an encryption query and a decryption query. The probability bound comes from the fact that for the i^{th} decryption query, the $(p_i + 1)^{st}$ block cipher input is fresh with high probability due to fact that **B2, B3** do not hold. The $(p_i + 1)^{st}$

[1]A prefix for a decryption query is defined as the common prefix blocks between the decryption query input string and an encryption query output string. The length of the common prefix for the i^{th} decryption query is denoted as p_i. Note that, if the decryption query uses a fresh nonce, then the decryption query input string does not share any common prefix with any of the encryption query output strings and we set $p_i = -1$.

block cipher input is not fresh with probability bounded by $\frac{64}{2^{64}}$. It comes from the fact that no multi-collision of size more than 64 occurs.

Forging event should imply one of the bad events. Hence the INT-CTXT advantage of GIFT-COFB is bounded by

$$\frac{q_f}{2^{128}} + \frac{2\sigma_e}{2^{64}} + \frac{q_f}{2^{64}} + \frac{q_f(q_e-1)}{2^{128}} + \frac{(q_e+\sigma_e+2\sigma_f)\sigma_f}{2^{128}} + \frac{64q_f}{2^{64}}$$

$$= \frac{3\sigma_e+q_f}{2^{64}} + \frac{(q_e+\sigma_e+2\sigma_f)\sigma_f + q_f + (q_e-1)q_f}{2^{128}} + \frac{64q_f}{2^{64}}.$$

4.3 Security Analysis of GIFT (Extract)

The security analysis of GIFT-128 is provided in Section 4 of [3]. Here we highlight several important features.

Differential cryptanalysis. Zhu et al. applied the mixed-integer-linear-programming based differential characteristic search method for GIFT-128 and found an 18-round differential characteristic with probability 2^{-109} [14], which was further extended to a 23-round key recovery attack with complexity $(Data, Time, Memory) = (2^{120}, 2^{120}, 2^{80})$. We expect that full (40) rounds are secure against differential cryptanalysis.

Linear cryptanalysis. GIFT-128 has a 9-round linear hull effect of $2^{-45.99}$, which means that we would need around 27 rounds to achieve correlation potentially lower than 2^{-128}. Therefore, we expect that 40-round GIFT-128 is enough to resist against linear cryptanalysis.

Integral attacks. The lightweight 4-bit S-box in GIFT may allow efficient integral attacks. The bit-based division property is evaluated against GIFT-128 by the designers, which detected a 11-round integral distinguisher.

Meet-in-the-middle attacks. Meet-in-the-middle attack exploits the property that a part of key does not appear during a certain number of rounds. The designers and the follow-up work by Sasaki [13] showed the attack against 15-rounds of GIFT-64 and mentioned the difficulty of applying it to GIFT-128 because of the larger ratio of the number of subkey bits to the entire key bits per round; each round uses 32 bits and 64 bits of keys per round in GIFT-64 and GIFT-128, respectively, while the entire key size is 128 bits for both.

Chapter 5

Design Rationale

5.1 AEAD Scheme: GIFT-COFB

COFB is a block cipher based authenticated encryption mode that uses GIFT-128 as the underlying block cipher and GIFT-COFB can be viewed as an efficient integration of the COFB and GIFT-128. GIFT-128 maintains an 128-bit state and 128-bit key. To be precise, GIFT is a family of block ciphers parametrized by the state size and the key size and all the members of this family are lightweight and can be efficiently deployed on lightweight applications. COFB mode on the other hand, computes of "COmbined FeedBack" (of block cipher output and data block) to uplift the security level. This actually helps us to design a mode with low state size and eventually to have a low state implementation. This technique actually resist the attacker to control the input block and next block cipher input simultaneously. Overall, a combination of GIFT and COFB can be considered to be one of the most efficient lightweight, low state block cipher based AEAD construction.

5.2 Block Cipher Primitive: GIFT-128

GIFT is considered to be one of the lightest design existing in the literature. It is denoted as "Small PRESENT" as the design rationale of GIFT follows that of PRESENT [4]. However, GIFT has got rid of several well known weaknesses existing in PRESENT with regards to linear cryptanalysis. Overall GIFT promises much increased efficiency (both lighter and faster) over PRESENT. GIFT is a very simple design that outperforms even SIMON and SKINNY for round based implementations. It consists of very simple operations such that the total hardware footprint is almost consumed by the underlying and the cipher storage. The design is somewhat "optimal" as a weaker S-box (than GIFT S-box) would lead to a weaker design. The linear layer is completely free for a round-based implementation in hardware (consisting of simply bit-wiring) and the constants are generated thanks to a very lightweight LFSR. The key schedule is also very

light, simply consisting of shifts. The presented security analysis details and hardware implementation results also support the claims made by the designers.

Although there is almost no impact on hardware implementation, there are several motivations for using bitslice implementation (non-LUT based) instead of LUT based implementation of GIFT-128 when we consider software implementation. Here, we will state the 3 most obvious benefits relating to its 3 steps in a round function.

Firstly for the non-linear layer, for LUT based implementation, we can consider updating 2 GIFT S-boxes (1 byte) in a single memory call with a reasonable 256 entries LUT. This would require 16 lookups and it takes approximately 16 to 64 cycles to do all S-boxes in a round, assuming a few cycles to access the RAM. Using bitslice implementation, it requires just 11 basic operations (or 10 with XNOR operation) to compute all the S-boxes in parallel. And more importantly, using bitslice implementation has the nice feature that it doesn't need any RAM and that it is constant time, mitigating potential timing attacks.

Secondly for the linear layer, while it is basically free on hardware, for software implementation it is extremely slow and complex to implement. This effect can be reduced by doing several blocks in parallel using none other than bitslice implementation. Even for a single block encryption, bitslice implementation is still more efficient that LUT based implementation because of the way the bits are packed.

Third and lastly the key addition, for LUT based implementation, the subkeys need to be XORed to bit positions that are 3 bits apart, making the key addition tedious and non-trivial. An option is to precompute the subkeys, but even so the key addition would require several XOR operations to update the 128-bit state. Using bitslice, the bits that were once 3 bits apart are now packed together in 32-bit words, making the key addition as simple as just 2 XOR operations.

5.3 Authenticated Encryption Mode: COFB

COFB is a lightweight AEAD mode. The mode presented in this write up differs slightly with the original proposal. They are as follows.

- We change the nonce to be 128 bit.
- We change the feedback (more precisely the G matrix) to make it more hardware efficient.
- We now deal with empty data. We change the mask update function for the purpose.
- We change the padding for the associated data. To be precise, if the associated data is empty, then padding the associated data will yield the constant block 10^{n-1} (n: block cipher state size).

We observed that, the updates make the design more lightweight and more efficient to deal with short data inputs. However, this updates does not have impact on the security of the mode (only a nominal 1-bit security degradation).

Acknowledgments

Subhadeep Banik is supported by the Ambizione grant PZ00P2_179921, awarded by the Swiss National Science Foundation. Thomas Peyrin is supported by the Temasek Labs grant (DSOCL16194).

Bibliography

[1] Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication. NIST Special Publication 800-38B, 2005. National Institute of Standards and Technology.

[2] 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.

[3] Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Siang Meng Sim, Yosuke Todo, and Yu Sasaki. Gift: A small present. Cryptology ePrint Archive, Report 2017/622, 2017. `https://eprint.iacr.org/2017/622`.

[4] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In *CHES 2007*, pages 450–466, 2007.

[5] Avik Chakraborti, Tetsu Iwata, Kazuhiko Minematsu, and Mridul Nandi. Blockcipher-based authenticated encryption: How small can we go? In *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, pages 277–298, 2017.

[6] Naina Gupta, Arpan Jati, Anupam Chattopadhyay, Somitra Kumar Sanadhya, and Donghoon Chang. Threshold implementations of GIFT: A trade-off analysis. *IACR Cryptology ePrint Archive*, 2017:1040, 2017.

[7] Sebastian Kutzner, Phuong Ha Nguyen, Axel Poschmann, and Huaxiong Wang. On 3-share threshold implementations for 4-bit s-boxes. In *Constructive Side-Channel Analysis and Secure Design - 4th International Workshop, COSADE 2013, Paris, France, March 6-8, 2013, Revised Selected Papers*, pages 99–113, 2013.

[8] Carlos Andres Lara-Nino, Arturo Diaz-Perez, and Miguel Morales-Sandoval. Fpga-based assessment of midori and gift lightweight block ciphers. In *Information and Communications Security - 20th International*

Conference, ICICS 2018, Lille, France, October 29-31, 2018, Proceedings, pages 745–755, 2018.

[9] NIST. Lightweight cryptography project, 2019.

[10] Axel Poschmann, Amir Moradi, Khoongming Khoo, Chu-Wee Lim, Huaxiong Wang, and San Ling. Side-channel resistant crypto for less than 2, 300 GE. *J. Cryptology*, 24(2):322–345, 2011.

[11] 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.

[12] Carsten Rolfes, Axel Poschmann, Gregor Leander, and Christof Paar. Ultra-Lightweight Implementations for Smart Devices - Security for 1000 Gate Equivalents. In Gilles Grimaud and François-Xavier Standaert, editors, *CARDIS*, volume 5189 of *Lecture Notes in Computer Science*, pages 89–103. Springer, 2008.

[13] Yu Sasaki. Integer linear programming for three-subset meet-in-the-middle attacks: Application to gift. In Atsuo Inomata and Kan Yasuda, editors, *Advances in Information and Computer Security*, pages 227–243, Cham, 2018. Springer International Publishing.

[14] Baoyu Zhu, Xiaoyang Dong, and Hongbo Yu. Milp-based differential attack on round-reduced gift. Cryptology ePrint Archive, Report 2018/390, 2018. `https://eprint.iacr.org/2018/390`.

Changelog

- 29-03-2019: version v1.0