A Small GIFT-COFB: Lightweight Bit-Serial Architectures

Andrea Caforio[1], Daniel Collins[1], Subhadeep Banik[2], and Francesco Regazzoni[3,2]

[1] LASEC, Ecole Polytechnique Fédérale de Lausanne, Switzerland
{andrea.caforio,daniel.collins}@epfl.ch
[2] Università della Svizzera Italiana, Lugano, Switzerland
subhadeep.banik@usi.ch
[3] University of Amsterdam, Netherlands
f.regazzoni@uva.nl

Abstract. GIFT-COFB is a lightweight AEAD scheme and a submission to the ongoing NIST lightweight cryptography standardization process where it currently competes as a finalist. The construction processes 128-bit blocks with a key and nonce of the same size and has a small register footprint, only requiring a single additional 64-bit register. Besides the block cipher, the mode of operation uses a bit permutation and finite field multiplication with different constants. It is a well-known fact that implementing a hardware block cipher in a bit-serial manner, which advances only one bit in the computation pipeline in each clock cycle, results in the smallest circuits. Nevertheless, an efficient bit-serial circuit for a mode of operation that utilizes finite field arithmetic with multiple constants has yet to be demonstrated in the literature.
In this paper, we fill this gap regarding efficient field arithmetic in bit-serial circuits, and propose a lightweight circuit for GIFT-COFB that occupies less than 1500 GE, making it the to-date most area-efficient implementation of this construction. In a second step, we demonstrate how the additional operations in the mode can be executed concurrently with GIFT itself so that the total latency is significantly reduced whilst incurring only a modest area increase. Finally, we propose a first-order threshold implementation of GIFT-COFB, which we experimentally verify resists first-order side-channel analysis. (For the sake of reproducibility, the source code for all proposed designs is publicly available [14].)

Keywords: GIFT-COFB · Serial · ASIC · Threshold Implementation

1 Introduction

Resource-constrained devices have become pervasive and ubiquitous commodities in recent years to the extent that the task of securing such gadgets has spawned a dedicated branch of cryptographic research. Lightweight cryptography is a discipline that comprises the creation, analysis and implementation of resource-optimized cryptographic primitives in terms of criteria such as circuit area, power consumption and latency.

This proliferation of low-resource devices and their security requirements spurred the NIST Lightweight Cryptography competition [1]. Commencing in 2018 and recently entering its ultimate round with ten competing candidate designs, the competition can nowadays be considered the essential driving force in this research field. GIFT-COFB [9,10] is one the finalists and thus an efficient implementation of this construction on hardware and software platforms is both timely and useful. The designers of this scheme have provided results for round-based circuits, i.e., which perform one round of the underlying block cipher encryption algorithm per clock cycle. However, such circuits, although they consume less energy [6], induce a higher hardware footprint in gate count. Consequently, the minimum circuit area of GIFT-COFB remains unexplored.

A popular technique to reduce the hardware footprint of circuits is serialization. Serialized circuits operate with a datapath of width much less than the specified block size of the cipher, and therefore allow for specific resources of the circuit to be reused several times in each round. The byte-serial circuit (i.e., which advances one byte in the computation pipeline in each clock cycle) for AES-128 [16] by Moradi et al. [21], with area equivalent to around 2400 GE, remained for many years the most compact implementation of this block cipher. The implementation was subsequently extended to support both encryption and decryption capabilities as well as different key sizes [2,7,8].

A first generic technique to obtain bit-serial block cipher implementations, termed *bit-sliding*, was proposed in a work by Jean et al. [20] yielding, at the time, the smallest circuits for the ciphers AES-128, SKINNY [11] and PRESENT [13]. However, all these circuits required more clock cycles than the block size of the underlying block ciphers to execute one encryption round. The circuit for PRESENT was further compacted in [4] with a technique that made it possible to execute one round in exactly 64 clock cycles which is equal to the block size. This endeavour of computing a round function in the same number cycles as there are bits in the internal state was successfully extended to other ciphers including AES-128, SKINNY and GIFT-128 [3]. This was achieved by not treating the round as a monolithic entity by deferring some operations to the time allotted to operations of the next round. Additionally, the authors proposed bit-serial circuits for some modes of operation such as SAEAES [22], SUNDAE-GIFT [5], Romulus [18], SKINNY-AEAD [12]. It is important to note that the canon of bit-serial works has pushed implementations to a point where the corresponding circuits are predominantly comprised of storage elements with almost negligible amounts of combinatorial parts that implement the actual logic of the algorithm.

1.1 Contributions

Unlike the bit-serial AEAD implementations proposed in [3], GIFT-COFB involves finite field arithmetic for which there is no straightforward mapping into a bit-serial setting that is both circuit area and latency efficient. In this paper, we fill this gap by proposing *three* bit-serial circuits that stand as the to-date most area-efficient GIFT-COFB implementations known in the literature. More specifically, our contributions are summarized as follows:

1. GIFT-COFB-SER-S: This circuit represents an effective transformation of the *swap-and-rotate* GIFT-128 scheme into the GIFT-COFB mode of operation minimizing its area footprint.
2. GIFT-COFB-SER-F: Subsequently, we observed that the interspersing of block cipher invocations with calls to the finite field module as found in the baseline GIFT-COFB design can be reordered by leveraging its inherent mathematical structure in order to further optimize the overall latency of GIFT-COFB-SER-S while only incurring a modest area increase.
3. GIFT-COFB-SER-TI: In a natural progression, we design a bit-serial first-order threshold implementation based on GIFT-COFB-SER-F whose security is experimentally verified through statistical tests on signal traces obtained by measuring the implemented circuit on a SAKURA-G side-channel evaluation FPGA board.
4. We synthesise all of the proposed schemes on ASIC platforms using multiple standard cell libraries and compare our results to existing bit-serial implementations of NIST LWC candidate submissions, indicating our designs are among the smallest currently in the competition. A brief overview of the synthesis results is tabulated in Table 1.

Table 1: Synthesis results overview for lightweight block cipher based NIST LWC competitors using the STM 90 nm cell library at a clock frequency of 10 MHz. Latency and energy correspond to the encryption of 128 bits of AD and 1024 message bits. Highlighted schemes are NIST LWC finalists. A more detailed breakdown, including figures for the TSMC 28 nm and NanGate 45 nm processes, is given in Table 2.

	Datapath	Area	Latency	Power	Energy	Reference
	Bits	GE	Cycles	μW	nJ	
SUNDAE-GIFT	1	1201	92544	55.48	513.4	[3]
SAEAES	1	1350	24448	84.47	206.5	[3]
Romulus	1	1778	55431	82.28	456.1	[3]
SKINNY-AEAD	1	3589	72960	143.7	1048	[3]
GIFT-COFB	128	3927	400	156.3	6.254	[9]
GIFT-COFB-SER-S	1	1443	54784	50.11	275.8	Section 3
GIFT-COFB-SER-F	1	1485	51328	62.15	319.8	Section 4
GIFT-COFB-SER-TI	1	3384	51328	158.1	813.5	Section 5

1.2 Roadmap

Section 2 introduces preliminaries and the description of the GIFT-COFB AEAD scheme. Section 3 delves into the complexities of implementing finite field multiplication and presents the first bit-serial circuit for GIFT-COFB. In Section 4, we present a modified circuit in which the finite field operations are absorbed in the last encryption round of the GIFT-128 block cipher. Section 5 presents the circuit for the first order threshold implementation of GIFT-COFB and experimental results for leakage detection in which we do not observe any first-order leakage. Section 6 shows our implementation results. Section 7 concludes the paper.

2 Preliminaries

For the remainder of this paper, we denote by upper-case letters bitvectors, e.g., $X = x_{n-1}x_{n-2}\cdots x_1x_0$ represents a vector of length n composed of individual bits x_i and ϵ the empty string. We use $||$ to indicate a concatenation of two bitvectors. $\lll i$ signifies the leftward rotation of a bitvector by i positions, whereas $\ll i$ is the leftward logical shift. For any binary string X and bit b, $b * X = X$ if $b = 1$ and $0^{|X|}$ if $b = 0$.

2.1 GIFT-COFB

GIFT-COFB is a block-cipher-based authenticated encryption mode that integrates GIFT-128 as the underlying block cipher with an 128-bit key and state. The construction adheres to the *COmbined FeedBack* mode of operation [15] which provides a processing rate of 1, i.e., a single block cipher invocation per input data block. The mode only adds an additional 64-bit state L to the existing block cipher registers and thus ranks among the most lightweight AEAD algorithms in the literature.

In the following, let $n = 128$ and denote by E_K a single GIFT-128 encryption using key $K \in \{0,1\}^n$. Furthermore, $N \in \{0,1\}^n$ signifies a nonce and A represents a list of n-bit associated data blocks of size $a \geq 0$. Analogously, let M be a list of n-bit plaintext blocks of size $m \geq 0$. GIFT-COFB intersperses E_K calls with that of several component functions. In particular, it uses a truncation procedure $\mathrm{Trunc}_k(x)$ that retrieves the k most significant bits of an n-bit input and a padding function $\mathrm{Pad}(x)$ that extends inputs whose lengths are not a multiple of n as follows:

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

Additionally, the internal state enters a feedback function between encryptions composed of two rotations of an input $X = (X_0, X_1)$ where $X_i \in \{0,1\}^{n/2}$ such that

$$\mathrm{Feed}(X) = (X_1, X_0 \lll 1).$$

Alongside the execution of Feed, the auxiliary state L is updated through a multiplication over the finite field $\mathrm{GF}(2^{64})$ generated by the root of the polynomial $p_{64}(x) = x^{64} + x^4 + x^3 + x + 1$. Consequently, the doubling of an element

$z = z_{63}z_{62}\cdots z_0 \in \mathrm{GF}(2^{64})$, i.e., the multiplication by the primitive element $\alpha = x = 0^{62}10$, is conveniently calculated as

$$\alpha \cdot z = \begin{cases} z \ll 1 & \text{if } z_{63} = 0 \\ (z \ll 1) \oplus 0^{59}11011 & \text{otherwise.} \end{cases} \tag{1}$$

By leveraging this multiplication, we can similarly triple an element z by calculating $(1 + \alpha) \cdot z$. The encryption of the last block of both A and M is preceded by the multiplication of L by 3^x and 3^y respectively, where

$$x = \begin{cases} 1 & \text{if } |A| \bmod n = 0 \text{ and } A \neq \epsilon, \\ 2 & \text{otherwise;} \end{cases} \quad y = \begin{cases} 1 & \text{if } |M| \bmod n = 0 \text{ and } M \neq \epsilon, \\ 2 & \text{otherwise.} \end{cases}$$

All other encryptions lead to a multiplication of L by 2, excluding the initial encryption of the nonce. Ultimately, the mode of operation produces a ciphertext C of size $|C| = |M|$ and a tag $T \in \{0,1\}^n$. A graphical diagram of GIFT-COFB is given in Figure 1.

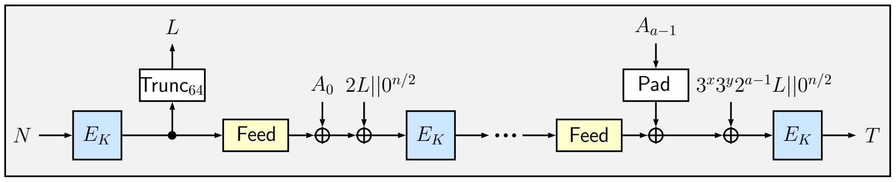
2.2 Swap-and-Rotate Methodology

Swap-and-rotate is a natural progression of the bit-sliding technique with a particular focus on reducing the latency of an encryption round such that the number of required cycles is equal to the bit-length of the internal state. This technique was first successfully demonstrated on PRESENT and GIFT-64 by Banik et al. [4] and refined for other block ciphers in a follow-up work [3]. The core idea behind the technique lies in the reliance on a small number of flip-flop pairs that swap two bits in-place at specific points in time during the round function computation while the state bits are rotating through the register pipeline one position per clock cycle.

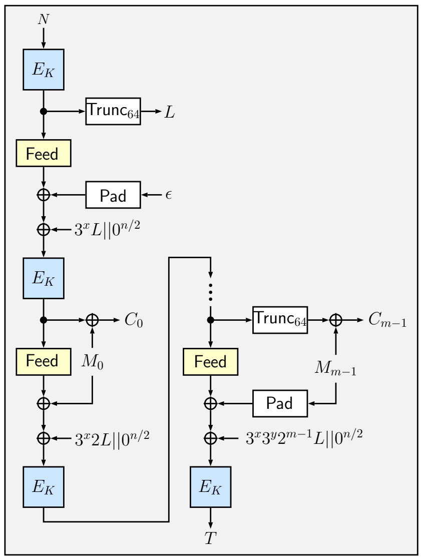
For simplicity, we represent an n-bit pipeline, i.e., shift register, as a sequence of individual flip-flops such that $\mathtt{FF}_{n-1} \leftarrow \mathtt{FF}_{n-2} \leftarrow \cdots \leftarrow \mathtt{FF}_1 \leftarrow \mathtt{FF}_0$. A swap is a hard-wired connection between two flip-flops $\mathtt{FF}_i, \mathtt{FF}_j$ that, when activated, exchanges the stored bits in registers $\mathtt{FF}_i$ and $\mathtt{FF}_j$ and takes effect in the following clock cycle (shown in Figure 2). Feature-rich cell libraries normally offer a specific register type, a so-called scan-flip-flop, that enriches a normal d-flip-flop with an additional input value in order to implement this functionality more efficiently than simply multiplexing the input bits. However, if one flip-flop is part of multiple swaps, there is usually no other solution than placing an additional multiplexer in front of a regular d-flip-flop.

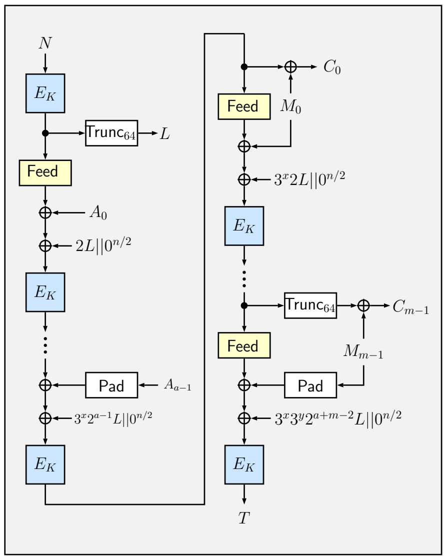
Depending on the block cipher, *swap-and-rotate* may be sufficient to fully implement the linear layer without any additional logic, especially in the case of simple bit permutations like those found in PRESENT or the bit-sliced variant of GIFT which is used in GIFT-COFB.[1] Considering the latter, a reinterpretation

[1] A detailed description of the bit-sliced GIFT representation can be found in the GIFT-COFB white paper [9].



(a) $A \neq \epsilon,\ M = \epsilon$





(b) $A = \epsilon,\ M \neq \epsilon$ (c) $A \neq \epsilon,\ M \neq \epsilon$

Fig. 1: Schematic depiction of GIFT-COFB mode of operation for all associated data and plaintext sizes. We remark that an empty associated data input will always be padded to a full block, hence the minimum number of encryption calls is two.

of GIFT-128, the state bits $x_{127} \cdots x_1x_0$ are partitioned into four lanes such that

$$S_3 = x_{127}x_{126} \cdots x_{97}x_{96}, \qquad S_2 = x_{95}x_{94} \cdots x_{65}x_{64},$$
$$S_1 = x_{63}x_{62} \cdots x_{33}x_{32}, \qquad S_0 = x_{31}x_{30} \cdots x_1x_0.$$

The bit permutation Π now reduces to four independent sub-permutations $\Pi_3, \Pi_2, \Pi_1, \Pi_0$ that act on each lane

$$\Pi(x_{127} \cdots x_0) = \Pi_3(x_{127} \cdots x_{96})\Pi_2(x_{95} \cdots x_{64})\Pi_1(x_{63} \cdots x_{32})\Pi_0(x_{31} \cdots x_0).$$

This fact was then exploited in [3] to compute each sub-permutation while the corresponding bits are advancing through $\mathtt{FF}_i$ for $96 \leq i \leq 127$. More specifically, the plaintext is loaded into $\mathtt{FF}_0$ throughout cycles 0-127. In cycles 96-127,

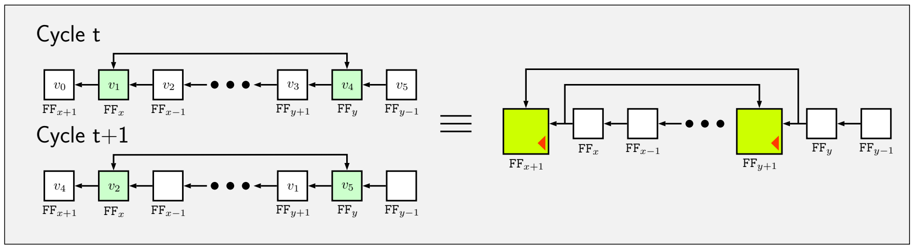


Fig. 2: The fundamental concept of *swap-and-rotate*. When a swap is active as shown by a colored box on FF_x and FF_y, then the operation performed in the pipeline swaps the contents of FF_x and FF_y and then rotates. The construct can be achieved by using scan flip-flops wired as shown above. The large green boxes on the right denote scan flip-flops.

the S-box layer of the first round and the swaps that calculate Π_3 are active. Subsequently, the swaps corresponding to Π_2, Π_1, Π_0 are active during the cycles 128-159, 160-191 and 192-224 respectively, concluding the calculation of the first round function. This pattern repeats for the remaining rounds until the 40-th and ultimate round which starts executing in cycle 5088. The first ciphertext bits are made available at FF_{127} from cycle 5120 until the last bit has exited the pipeline in cycle 5248. Hence, a full encryption takes exactly $(40+1) \cdot 128 = 5248$ cycles. A schematic timeline diagram is given Figure 3.

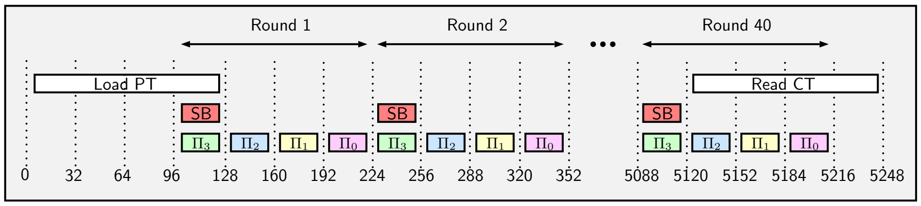


Fig. 3: Timeline diagram of the *swap-and-rotate* GIFT-128 implementation; the numbers in the x-axis denote clock cycles.

Another peculiarity of the bit-sliced GIFT-128 variant is that the 4-bit S-box is not applied to adjacent bits of the state but to the first bits of each lane, i.e., $x_{96}, x_{64}, x_{32}, x_0$. Summa summarum, the circuit for the state pipeline is compact and simple as shown in Figure 4.[2]

[2] Note that there is an equally efficient circuit for the key schedule pipeline. An exact breakdown of all the swaps in the state pipeline is given in Appendix A.

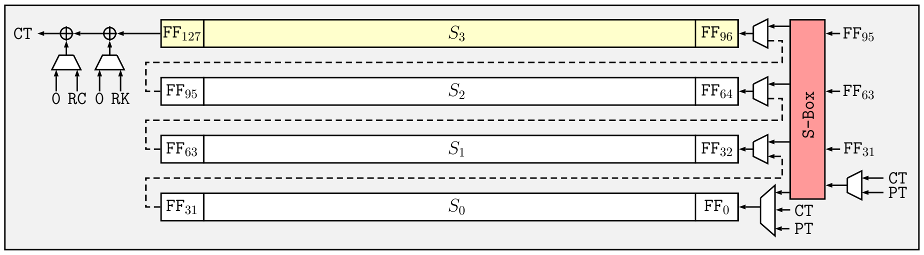


Fig. 4: The *swap-and-rotate* GIFT-128 state pipeline circuit. There are in total 9 swaps over 12 flip-flops. Their exact placement and activity cycles are given in Appendix A.

3 GIFT-COFB-SER-S

In this section, we lay the groundwork for our bit-serial GIFT-COFB circuits and describe how to efficiently implement the field multiplication as well as the feedback function. In the process, we integrate the obtained component circuits with the *swap-and-rotate* module described in Section 2.2 which yields the first lightweight bit-serial GIFT-COFB circuit. This is straightforward in the sense that there is a clear separation between the execution of the GIFT-128 encryption, the calculation of Feed and the addition of L to the internal state, alongside the loading of the plaintext as part of the next encryption. Meaning that, after the ciphertext has completely exited the pipeline, these three operations are each performed in 128 separate cycles during which the GIFT-128 pipeline executes the identity function, i.e., the state bits rotate through the shift register without the activation of any swap or the S-box. Hence, there is an overhead of 3×128 cycles between encryption invocations. We denote this circuit by GIFT-COFB-SER-S which will be the basis for the latency-optimized variant, presented in Section 4, that circumvents those periodic 384 *penalty* cycles with only a marginal increase in circuit area.[3] The exact sequence of operations between encryptions is described in Figure 5.

3.1 Implementing the Feedback Function

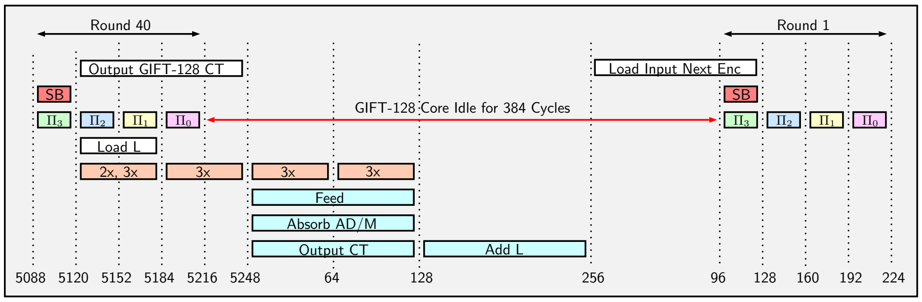
Recall the feedback function as detailed in Section 2.2, i.e.,

$$\mathrm{Feed}(X_0, X_1) = (X_1, X_0 \lll 1).$$

It is a bit-permutation belonging to the symmetric group over a set of 128 elements and executes two operations sequentially:

1. Swapping of the upper and lower halves of the word $(X_0, X_1) \rightarrow (X_1, X_0)$.
2. Leftward rotation of the lower half X_0 by one position.

[3] The letters S and F in GIFT-COFB-SER-S and GIFT-COFB-SER-F stand for slow and fast respectively.



Cycles	Operations
0-5087	The nonce is fed into the state pipeline bit by bit in cycles 0 to 127. Thereafter, the first 39 rounds of GIFT-128 are executed.
5088-5247	Round 40 executes during cycles 5088-5216. The resulting ciphertext bits exit the pipeline during cycles 5120-5247. We read the first 64-bits of the ciphertext into the L register during cycles 5120-5183 while executing the first multiplication during the same period.
For each additional data block, the following cycles are executed sequentially:	
0-127	After the ciphertext has fully exited the state pipeline, we start executing the feedback function for 128 cycles. In parallel, we can absorb the input data block and, if needed, produce the ciphertext bits. Subsequent multiplications of L are performed if required.
128-255	The state after the above is now XORed with the content of the L register and the result is written back, bit by bit, into the state register.
0-5247	A new encryption starts after L has been added to the cipher state.

Fig. 5: Timeline diagram and cycle-by-cycle description of GIFT-COFB-SER-S for two successive encryptions. Note the interval of 3×128 idle cycles between encryptions.

Proposition 1. *Using two swaps over four flip-flops, it is possible to fully implement both subroutines of the subroutine Feed in exactly 128 clock cycles.*

Proof. A schematic cycle-by-cycle diagram of the feedback function is depicted in Figure 6. The first swap $\mathtt{FF}_0 \leftrightarrow \mathtt{FF}_1$ is active from cycles 2 to 64. As a result, the state at clock cycle 64 is given as $x_{63}x_{62}\cdots x_1x_0x_{126}x_{125}\cdots x_{127}x_{64}$. Note that this is already the output of Feed if the two least significant bits were swapped, which is then done in cycle 64. The second swap $\mathtt{FF}_{127} \leftrightarrow \mathtt{FF}_{63}$ is active from cycle 64 to 127 which effectively computes the identity function over 64 cycles. One can thus see that after 128 cycles that the register contains the intended output of the Feed function. $\square$

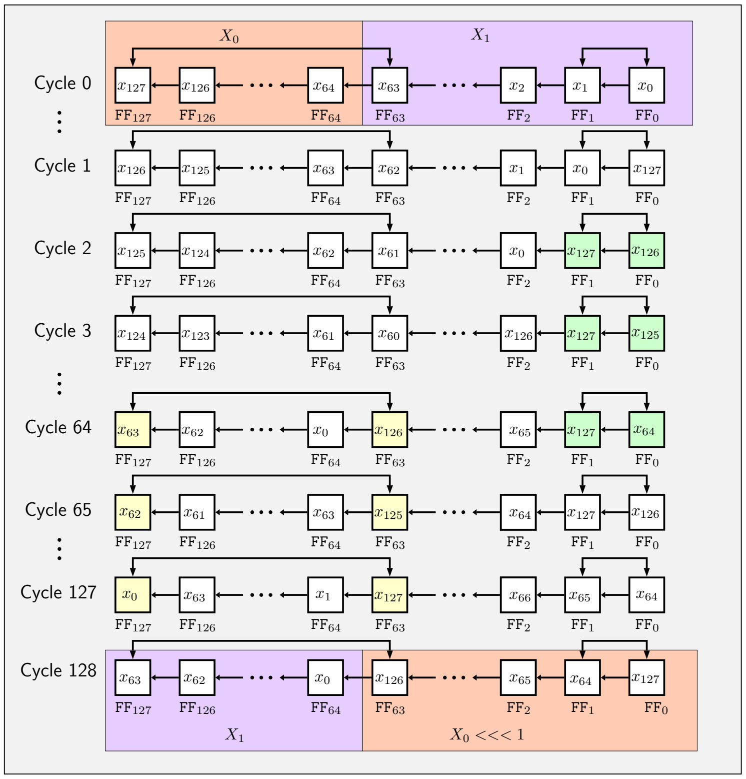


Fig. 6: Cycle-by-cycle execution diagram of the feedback function. Green marked registers denote active swaps that execute $X_0 \lll 1$ while yellow registers mark active swaps that perform $(X_0, X_1) \rightarrow (X_1, X_0)$. Note that when a swap is active as shown by a colored box on $\texttt{FF}_x$ and $\texttt{FF}_y$, then the operation performed in the pipeline is a) swap contents of $\texttt{FF}_x$ and $\texttt{FF}_y$ and then b) rotate. The construct can be achieved by using scan flip-flops wired as shown in Figure 2.

Absorbing Data Blocks and Outputting the Ciphertext. In order to avoid having to pass the message block bits twice to the circuit, once to produce the AEAD ciphertext and once for absorption into the state, i.e., Feed$(X) \oplus M$, this absorption is performed in parallel to the execution of the feedback function. Note that if $X = x_{127}x_{126} \cdots x_0$ and $M = m_{127}m_{126} \cdots m_0$, then the i-th bit u_i of Feed$(X) \oplus M$ is given as:

$$u_i = \begin{cases} m_i \oplus x_{i-64} & \text{if } 64 \leq i < 128, \\ m_i \oplus x_{i+63} & \text{if } 0 < i \leq 63, \\ m_i \oplus x_{127} & \text{if } i = 0 \end{cases}$$

By inspection of Figure 6, one can see that in order to execute the above seamlessly, the data bits must be added to $\texttt{FF}_{63}$. This is because, for any i, the state bit x_{i-64} (for $64 \leq i < 128$), x_{i+63} (for $0 < i \leq 63$) and x_{127} (for $i = 0$) is always present at $\texttt{FF}_{63}$ at clock cycle i. Thus, to implement the above, we need one additional XOR gate before the 63rd flip-flop in the state register. Additionally, $\texttt{FF}_{127}$ always contains the most significant bit of $X \lll i$ at any cycle $i \in [0, 127]$, thus the ciphertext, which is computed as $M \oplus X$, is extracted by adding the input data bit with $\texttt{FF}_{127}$. In Figure 7, we present the state pipeline circuit of GIFT-COFB-SER-S that integrates the swaps of the feedback function, and the additional XOR gates for the data absorption and ciphertext creation.

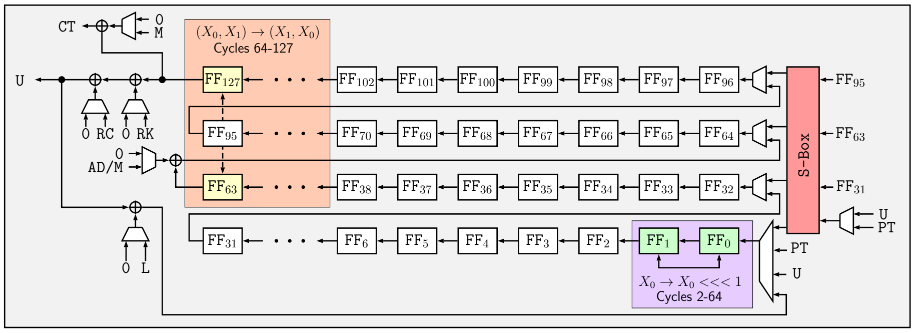


Fig. 7: GIFT-COFB-SER-S state pipeline.

3.2 Multiplication by 2 and 3

GIFT-COFB multiplies the auxiliary state L between encryptions by either the factor 2 or 3^x for $1 \leq x \leq 4$ depending on the associated data and message block sizes and padding. If it were not for the period right after the initial encryption of the nonce N in which L has to be loaded and updated in a short time interval, this would not be too much of an issue as there is ample time to calculate the multiplication while the encryption core is busy. In the following, we demonstrate how to efficiently multiply L by 2 or 3 in 64 cycles, yielding a maximum latency of 256 clock cycles for any factor 3^4.

Let $L = l_{63}l_{62}\cdots l_1 l_0$ be the individual bits of the register. On a 64-bit shift register, multiplication by 2 has the following form:

$$2 \times l_{63}l_{62}\cdots l_0 = (L \ll 1) \oplus (l_{63} * 0^{59}11011)$$

which, in plain terms, is simply a leftward shift by one position and the addition of the most significant bit l_{63} to four lower bits. On the other hand, the multiplication by three is more involved as $3 \times L = (2 \times L) \oplus L$ and is thus given as

$$3 \times l_{63}l_{62}\cdots l_0 = (L \ll 1) \oplus (l_{63} * 0^{59}11011) \oplus L.$$

A single-cycle implementation of this function necessitates 64 additional 2-input XOR gates that would incur roughly 128 GE in most standard libraries, which is a considerable overhead for a bit-serial circuit. Note that technically $3\times$ can be implemented with zero additional gates, if one is prepared to pay with latency. This is because $p_{64}(x)$ is a primitive polynomial, and since the element 2 is the root of $p_{64}(x)$ it must generate the cyclic multiplicative group of the finite field. With some arithmetic, it can be deduced that $3 = 2^d$ where $d = 9686038906114705801$ in this particular representation of the finite field. The discrete logarithm d of 3 is an integer of the order of 2^{63}, hence executing the multiplication by 2 over d would, in theory, compute the multiplication by the factor 3.

Disregarding this theoretical detour, our actual goal consists in implementing, with minimal circuitry, both the multiplication by 2 and 3 in such a way that after 64 clock cycles the first bit of the updated state exits the pipeline and after the 128 cycles the entire multiplication has finished.

Proposition 2. *By equipping the L shift register with a single auxiliary d-flip-flop, three 2-input NAND gates, one 2-input XOR gate and one 2-input XNOR gate, it is possible to multiply L by either 2 or 3, i.e., by the polynomials x or $(x+1)$.*

Proof. We begin by observing the following: if $V = v_{63}v_{62}\cdots v_0 = 2 \times l_{63}l_{62}\cdots l_0$ and $W = w_{63}w_{62}\cdots w_0 = 3 \times l_{63}l_{62}\cdots l_0$, where v_i, w_i are given as

$$v_i = \begin{cases} l_{i-1} \oplus l_{63} & \text{if } i \in \{1,3,4\}, \\ l_{63} & \text{if } i = 0, \\ l_{i-1} & \text{otherwise} \end{cases} \qquad w_i = \begin{cases} l_{i-1} \oplus l_{63} \oplus l_i & \text{if } i \in \{1,3,4\}, \\ l_{63} \oplus l_i & \text{if } i = 0, \\ l_{i-1} \oplus l_i & \text{otherwise.} \end{cases}$$

It is immediately evident that for all three cases v_i and w_i differ only by the XOR of the term l_i. In cycle 0, bit l_{63} is first stored in an auxiliary register, which we hereafter refer to as `Aux`. Using this fact, we show how to update the register. Then we calculate each update bit as follows, where α, β and γ are signals defined below:

$$u = (\alpha \cdot \texttt{Aux}) \oplus (\beta \cdot \texttt{FF}_{63}) \oplus (\gamma \cdot \texttt{FF}_{62}). \tag{2}$$

Identity function. It is simply a rotation of the L register; $\alpha = \beta = \gamma = 0$.

Multiplication by 2. Signal α is used to add l_{63}, which is stored in `Aux` in cycle 0, to the output bit. β is always 0 for multiplication by 2. γ is 1 for all but cycle 63 in order to implement a left shift (and not left rotate). Consequently, we have

$$\alpha = \begin{cases} 1 & \text{cycles } 59, 60, 62, 63, \\ 0 & \text{otherwise;} \end{cases} \qquad \beta = 0; \qquad \gamma = \begin{cases} 1 & \text{cycle} \neq 63, \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

Recall the doubling function (1). If the update function u were to simply be $\gamma \cdot \texttt{FF}_{62}$, then after 64 cycles, the register would store $l_{63}\cdots l_0 \ll 1$. Now if we added l_{63} to the LFSR update in cycles 59, 60, 62, 63, then after 64 cycles, the

LFSR state would be $(l_{63} \cdots l_0 \ll 1) \oplus (l_{63} * 0^{59}11011)$ which is the output of the doubling function.

Multiplication by 3. α, and γ as above and always $\beta = 1$. Adding $\beta \cdot \texttt{FF}_{63}$ to the update function enables the output to be $(l_{63} \cdots l_0 \ll 1) \oplus (l_{63} * 0^{59}11011) \oplus (l_{63} \cdots l_0)$, which is the output of the tripling function.

Using (2), we can implement both multiplications by factors 2 and 3 in 64 cycles using one auxiliary d-flip-flop, three 2-input NAND gates and one 2-input XOR gates and one 2-input XNOR gate. A diagram of the resulting circuit is shown in Figure 8. □

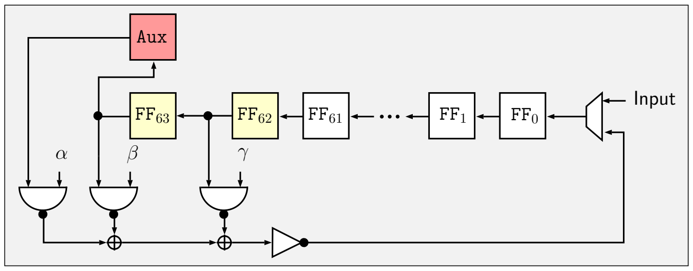


Fig. 8: Implementation of the bit-serial multiplication by 2 and 3.

3.3 GIFT-COFB-SER-S Total Latency

It can be seen that the encryption of the nonce takes 5248 cycles. Thereafter, every additional block takes $256 + 5248 = 5504$ cycles to process. Thus if the padded associated data and message consist of B blocks in total, then the time taken to produce the ciphertext and tag is $T_S = 5248 + 5504 \cdot B$ clock cycles.

4 GIFT-COFB-SER-F

The proposed bit-serial circuit from the previous section already represents the to-date most area-efficient GIFT-COFB implementation. However, as our bit-serial interpretation respects the natural order of operations as given in the specification of the mode of operation, it has a significantly elevated latency. This is mainly due to the encryption core being idle during 3×128 clock cycles between successive invocations which means that if we want to do away with those penalty cycles, the calculation of Feed, the update and addition of L, the addition of incoming associated data and message bits and the loading of the next encryption state all have to occur in parallel. This means that during 128 cycles while the GIFT ciphertext bits $c_i^{(j)}$ for datablock j leave the pipeline, the newly entering bits $v_i^{(j+1)}$ for data block $j+1$ at $\texttt{FF}_0$ are necessarily of the form

$$v_i^{(j+1)} = c_i^{(j)} \oplus \mathsf{RK}_i \oplus \mathsf{RC}_i \oplus L_i^{(j+1)} \oplus D_i^{(j+1)}, \qquad (4)$$

where $\mathsf{RK}_i \oplus \mathsf{RC}_i$ denote the i-th bit of the last round key, $L_i^{(j+1)}$ denotes the i-th bit of the L register to be added before the $(j+1)$-th data block and $D_i^{(j+1)}$ is the i-th bit of the $(j+1)$-th data block. In this section, we describe three requisite tweaks to GIFT-COFB-SER-S that let us achieve this goal.

1. Change the swaps of Feed described in Section 3.1 as to enable its execution in parallel to the ciphertext bits leaving the state pipeline.
2. Reorder the incoming data bits as well as L such that they can be seamlessly added to the exiting ciphertext bits.
3. Enrich the L circuit from Section 3.2 with additional logic in order to compute the multiplication by the factors $2, 3, 3^2, 3^3$ and 3^4 in 128 clock cycles concurrently with the last encryption round. The updated time diagram alongside a cycle-by-cycle description is given in Figure 9.

4.1 Tweaking the Feedback Function

Note that, as explained in Section 3.1, the swap between $\mathtt{FF}_{127}$ and $\mathtt{FF}_{64}$ during the calculation of the Feed function preserves the state over 64 cycles in GIFT-COFB-SER-S. However, the same can be achieved by swapping $\mathtt{FF}_x$ and $\mathtt{FF}_{x-63}$ for any x. Since we execute Feed concurrently with the last GIFT encryption round, we want the bit exiting the pipeline at $\mathtt{FF}_{127}$ to be the output of the GIFT encryption routine in the same order as in GIFT-COFB-SER-S. Swapping out $\mathtt{FF}_{127}$ and $\mathtt{FF}_{64}$, however, disrupts that order. Thus, we replace the swap $\mathtt{FF}_{127} \leftrightarrow \mathtt{FF}_{64}$ with the swap $\mathtt{FF}_{63} \leftrightarrow v_i^{(j)}$, where $v_i^{(j)}$ is the i-th bit of the j-th incoming block as defined above.

A side effect of this choice affects the S-box inputs of just the first round of every new encryption with an incoming data block. In GIFT-COFB-SER-F, in the first S-box invocation of a new encryption, inputs are now of the form $\mathtt{FF}_{95}$, $v_i^{(j)}, \mathtt{FF}_{31}, \mathtt{FF}_{63}$ instead of $\mathtt{FF}_{95}$, $\mathtt{FF}_{63}$, $\mathtt{FF}_{31}, v_i^{(j)}$ due to the $\mathtt{FF}_{63} \leftrightarrow v_i^{(j)}$ swap. As a result, we need two more multiplexers that swap $\mathtt{FF}_{63}$ and $v_i^{(j)}$ before entering into the S-box during cycles 96 to 127. The resulting circuit for GIFT-COFB-SER-F is depicted in Figure 10.

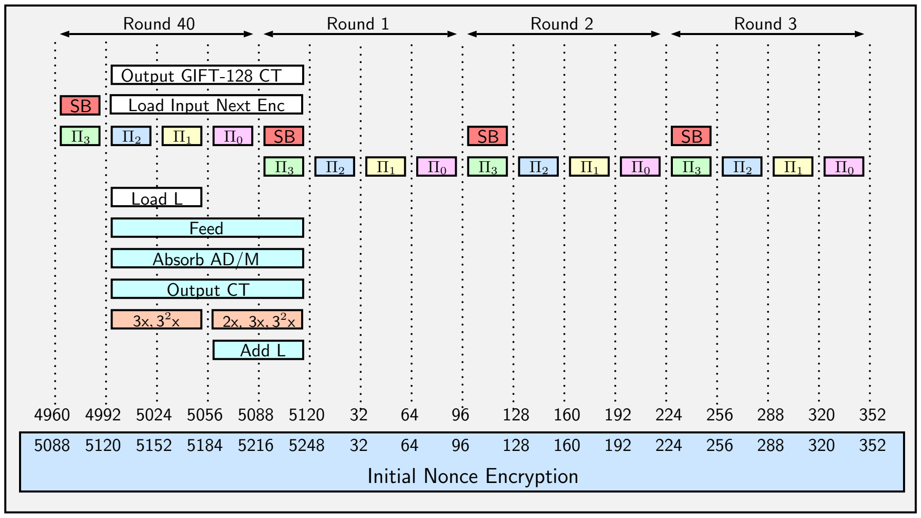
4.2 Reordering Data Bits

The absorption of associated data/message bits and L normally occurs after the computation of the feedback function. However, we have to do it with the last encryption round, which involves some re-ordering of data bits and L. Consider the inverse transformation of Feed:

$$\mathrm{Feed}^{-1}(X_0, X_1) = ((X_1 \ggg 1), X_0).$$

Note that Feed is a linear function, we have since $\mathrm{Feed}^{-1}(L, 0^{64}) = 0^{64} || L$:

$$\mathrm{Feed}(X \oplus \mathrm{Feed}^{-1}(D) \oplus \mathrm{Feed}^{-1}(L, 0^{64})) = \mathrm{Feed}(X) \oplus D \oplus L || 0^{64}.$$



Cycles	Operations
0-5087	The nonce is fed into the state pipeline bit by bit in cycles 0 to 127. Thereafter, the first 39 rounds of GIFT-128 are executed.
5088-5247	Round 40 executes during cycles 5088-5216. The resulting ciphertext bits exit the pipeline during cycles 5120-5247. We read the first 64-bits of the ciphertext into the L register during cycles 5120-5183 while executing the first multiplication during the same period such that in the second 64 cycles it is added back to the cipher state. In the same 128 cycles, we also execute Feed and add the data bits.
For each additional data block, the following cycles are executed sequentially:	
0-4959	We perform the first 39 rounds of the new encryption call.
4960-4991	The first 32 cycles of the last round of the encryption call.
4992-5119	We execute the following in parallel: we finish executing the encryption call, we load the new cipher state, perform multiplication of L, execute Feed, absorb data bits and the (updated) L, output the ciphertext and start executing round 1 of the next encryption call.

Fig. 9: Timeline diagram of GIFT-COFB-SER-F. Note that the initial 128 cycles to load the nonce cannot be parallelized with other functions, hence the initial encryption of the nonce takes 5248 cycles.

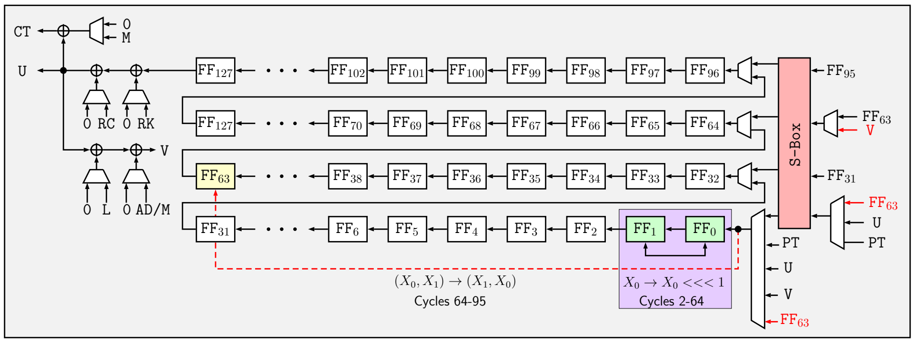


Fig. 10: GIFT-COFB-SER-F state pipeline. U denotes the input bit during intermediate cipher rounds and V the input during the first round of a new encryption. Wires marked in red enable the concurrent execution of Feed and the S-box.

We need to re-order the incoming data bits and the output of the L function by the permutation Feed^{-1} before adding it to the state, thereafter performing the Feed function over the modified state $X \oplus \mathrm{Feed}^{-1}(D) \oplus 0^{64}||L$ which thus correctly computes the input to the next encryption call. This comes with a convenient side effect:

Proposition 3. *Placing the addition of L before Feed yields 64 spare cycles that can be used to perform the finite field multiplications.*

Proof. When we add the string $\mathrm{Feed}^{-1}(L, 0^{64}) = 0^{64}||L$, the first 64 cycles are spent adding the zero string. These 64 cycles can be used to load L into its register and simultaneously multiply it by either $2, 3, 3^2, 3^3$ or 3^4 such that in cycle 64 the first correctly updated bits exit L and the entire register is updated in a total of 128 cycles. □

4.3 Enhancing the Multiplier

We proceed to demonstrate that the assertion from the previous proposition, namely that after 64 cycles the first correctly multiplied bit exits the L pipeline, can be integrated into the existing multiplier from Section 3.2 with modest overhead.

Proposition 4. *By equipping the L shift register with four auxiliary d-flip-flops, nine 2-input NAND gates, eight 2-input XOR gates and one 2-input XNOR gate, it is possible to multiply L by either 2, 3, 3^2, 3^3 or 3^4 in 128 cycles.*

Proof. Again let $L = l_{63}l_{62}\cdots l_0$ be the individual state, then the multiplication by 3^2 is written as

$$3^2 \times l_{63}l_{62}\cdots l_0 = (L \ll 2) \oplus (l_{63} * 0^{58}110110) \oplus L \oplus (l_{62} * 0^{59}11011)$$

Recall the multiplication circuit for the factors 2 and 3 from Section 3.2. We re-introduce signals α, β, γ as α_0, β_0 and γ_0, and to capture multiplication by 3^2 we further add δ_0 and α_1. Let `Aux0` be the register that stores l_{63} in cycle 0, and analogously denote by `Aux1` the auxiliary register that stores l_{62} in the same cycle. Then, the circuit for the multiplication by the factors 2, 3, 3^2 can be written as

$$u = (\alpha_0 \cdot \texttt{Aux0}) \oplus (\alpha_1 \cdot \texttt{Aux1}) \oplus (\beta_0 \cdot \texttt{FF}_{63}) \oplus (\gamma_0 \cdot \texttt{FF}_{62}) \oplus (\delta_0 \cdot \texttt{FF}_{61}).$$

In order to compute the multiplication by the higher factors 3^3 and 3^4, we equip the L pipeline with a second 3^2 circuit at the beginning that continuously overwrites register $\texttt{FF}_2$, which is therefore a scan flip-flop, as the bits enter the pipeline. In cycle 2, the values $\texttt{FF}_2$ and $\texttt{FF}_1$ are l_{63} and l_{62} respectively, which are stored in this cycle in auxiliary flip-flops `Aux2` and `Aux3`. The updated bit for these cases can be written as:

$$u' = (\alpha_2 \cdot \texttt{Aux2}) \oplus (\alpha_3 \cdot \texttt{Aux3}) \oplus (\beta_1 \cdot \texttt{FF}_2) \oplus (\delta_1 \cdot \texttt{FF}_0).$$

The resulting circuit full multiplier is shown in Figure 11.

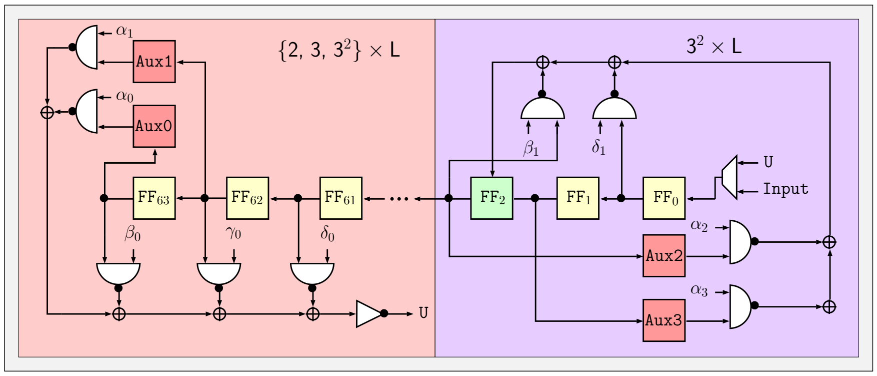


Fig. 11: L state pipeline that performs the multiplication by the factors 2, 3, 3^2, 3^3 and 3^4.

As before in Section 3.2, we give an exact list of activation cycles for each control signal below.

Identity function: All signals are set to 0.

Multiplication by 2. We have $\alpha_0 = \alpha$, $\beta_0 = \beta$ and $\gamma_0 = \gamma$ where α, β, γ are as in (3), additionally $\delta_0 = \alpha_1 = 0$. Since only the left half of the diagram is relevant, all other signals are 0.

Multiplication by 3. As in multiplication by 2 except for $\beta_0 = 1$.

Multiplication by 3^2. As above, only the left portion of the diagram is used. δ_0 steers the addition of l_{61}, and is active except for the last two cycles. α_0 enables the addition of l_{63}, similarly α_1 enables the addition of l_{62}. Furthermore, γ_0 is always 0 as it is only used in the multiplication by 3 and β_0 is always 1. In summary, we have

$$\alpha_0 = \begin{cases} 1 & \text{cycles } 58, 59, 61, 62, \\ 0 & \text{otherwise;} \end{cases} \quad \alpha_1 = \begin{cases} 1 & \text{cycles } 59, 60, 62, 63, \\ 0 & \text{otherwise;} \end{cases}$$

$$\delta_0 = \begin{cases} 1 & \text{cycle} < 62, \\ 0 & \text{otherwise;} \end{cases} \quad \beta_0 = 1; \quad \gamma_0 = 0.$$

Multiplication by $\mathbf{3^3}$. We first use the 3^2 multiplier on the right in the diagram that executes on newly entered bits then finish with a multiplication by 3 by the left multiplier. As we always update $\texttt{FF}_2$ for the factor 3^2, the activation cycles of the signals α_2, α_3, β_1 and δ_1 are analogous to the signals α_0, α_1, β_0 and δ_0 in the left 3^2 multiplication module except they occur 62 cycles before.

Multiplication by $\mathbf{3^4}$. The first phase is exactly as in the case of multiplication by 3^3, and the second phase is exactly as in multiplication by 3^2. □

4.4 GIFT-COFB-SER-F Total Latency.

It can be seen from Figure 9 that the encryption of the nonce takes 5248 cycles. Thereafter every additional block takes 5120 cycles to process. Thus if the padded AD and message consist of B blocks in total, then the time taken to produce the ciphertext and Tag is $T_F = 5248 + 5120 \cdot B$ clock cycles. We can see that for each block of data processed we save $\frac{T_S - T_F}{B} = 384$ clock cycles.

5 First-Order Threshold Implementation

In Boolean masking, sensitive values x are decomposed into s shares of the form x_i such that $\sum_0^{s-1} x_i = x$ where any set of up to $s-1$ shares are jointly independent of x. This technique can be used to provide security guarantees when an adversary can query up to $d = s-1$ wires in a circuit at any one point in time, i.e., to provide d-th order security. In the following, we restrict our attention to the case where $d = 1$.

Threshold implementations [23] are a family of masking schemes which provide provable first-order security guarantees even in the presence of hardware glitches [17]. In a threshold implementation (TI) of a given design, n-ary Boolean functions (i.e., sub-functions of the design) $f(x_{n-1}, ..., x_0) = z$ are divided into s components f_i such that $\sum_0^{s-1} f_i = f$. We consider sharings of values f that are *non-complete*, i.e., each f_i is independent of at least one value x_j, and *uniform*, i.e., for all x_i the number of sets $\{x_{n-1}, ..., x_0\}$ which satisfies for a given $(y_{k-1}, ..., y_0)$ both $\sum_i x_i = x$ and $f(x) = (y_{k-1}, ..., y_0)$ is constant. We also assume that maskings are uniform, i.e., for each x, each valid masking $\{x_{n-1}, ..., x_0\}$ such that $\sum_i x_i = x$ occurs with the same probability.

5.1 GIFT-COFB-SER-TI First-Order Threshold Implementation

We first note that the GIFT S-box $\mathtt{S}$ is cubic and can be decomposed into two quadratic S-boxes $\mathtt{S}_F$ and $\mathtt{S}_G$ from $\{0,1\}^4 \to \{0,1\}^4$ such that $\mathtt{S} = \mathtt{S}_F \circ \mathtt{S}_G$. Since $\mathtt{S}_F$ and $\mathtt{S}_G$ are quadratic, they can be masked using a direct sharing approach using three shares for a first-order threshold implementation such that

$$\mathtt{S}_G = \mathtt{S}_{G_1} \oplus \mathtt{S}_{G_2} \oplus \mathtt{S}_{G_3};\ \mathtt{S}_F = \mathtt{S}_{F_1} \oplus \mathtt{S}_{F_2} \oplus \mathtt{S}_{F_3},$$

where $\mathtt{S}_{G_1}, \mathtt{S}_{G_2}, \mathtt{S}_{G_3}$ and $\mathtt{S}_{F_1}, \mathtt{S}_{F_2}, \mathtt{S}_{F_3}$ are the component function of $\mathtt{S}_F$ and $\mathtt{S}_G$ respectively. This approach was used in [19] from which we take the proposed non-complete and uniform first-order TI. We provide the algebraic expression of the component functions in Appendix B.

Consequently, our implementation uses three shares for the state and L registers while the key and round constant pipeline remain unshared. The only challenge in constructing the circuit is to place the $\mathtt{S}_{F_i}$ and $\mathtt{S}_{G_i}$ substitution boxes such that they correctly compute the masked GIFT S-box in consonance with the other operations done in parallel. This can readily be achieved by noting that we can replace the unmasked $\mathtt{S}$ and replace it with $\mathtt{S}_{G_i}$, and place $\mathtt{S}_{F_i}$ after the first flip-flop of each lane, i.e., $\mathtt{FF}_0, \mathtt{FF}_{32}, \mathtt{FF}_{64}, \mathtt{FF}_{96}$, which executes for 32 cycles starting in cycle 97 of each round. A schematic of one of the three shares of the state pipeline is shown in Figure 13.

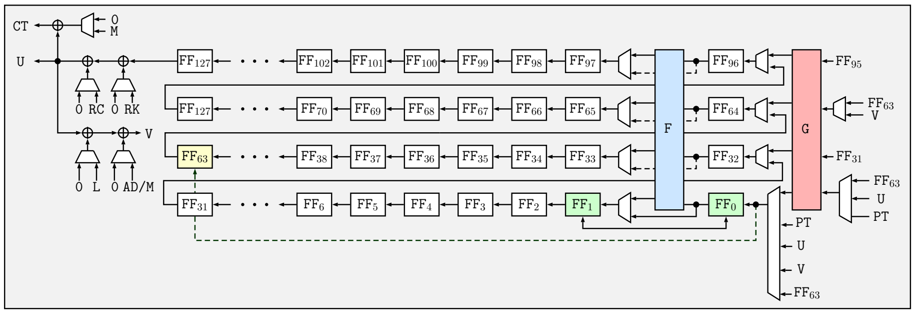


Fig. 12: One of the three state pipeline shares of the GIFT-COFB-SER-TI circuit.

5.2 Evaluation

We applied the TVLA methodology [24] and performed non-specific t-tests (using Welsh's t-test) to validate the first-order security of our threshold implementation GIFT-COFB-SER-TI. We took a threshold of $|t| > 4.5$ for any value of t computed to reject the null hypothesis that GIFT-COFB-SER-TI encryption operations admit indistinguishable mean power consumption in the case that the input is either uniform or fixed.

We used the SAKURA-G side-channel evaluation board[4] with two Spartan 6 FPGA cores, one which performed GIFT-COFB-SER-TI operations clocked at a slow 1.5MHz and the other that interfaces between the cryptographic core and a computer (which generates pre-masked shares for the DUT). To prevent unintended optimisations that could lead to leakage during synthesis, we added `DONT_TOUCH`, `KEEP` and `KEEP_HIERARCHY` constraints to our code. Measurements were taken with a Tektronix MSO44 at 625MS/s taking 3000 data points per trace, which corresponds to 7 cycles of S-box evaluation (the only non-linear component of GIFT-COFB) in the second round of the first GIFT encryption call, i.e., while GIFT is encrypting the nonce. During testing, we reset the cryptographic core between each GIFT-COFB encryption and interleaved encryptions with random and fixed inputs. A sample trace is shown in Figure 13a.

As a measure to ensure our setup was calibrated properly, we first performed a t-test in the leaky *masks off* setting, which is as follows. Recall that GIFT-COFB-SER-TI is a three-share TI. Then, in the masks off setting, one input value is set to the original input (fixed or random) and the other two to constant values (the zero vector here). We present the results in Figure 13b revealing that significant, potentially exploitable leakage was detected with just 20 thousand traces. Then, with masks on (i.e., with uniform masking used), we found no evidence of leakage with 10,000,000 traces, evident in Figure 13c.

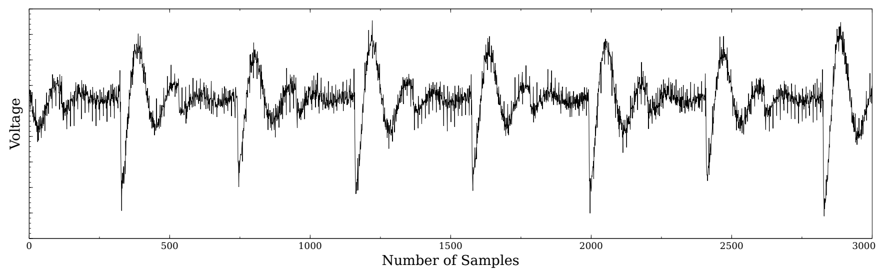
6 Implementation

All the investigated schemes were synthesized on ASIC platforms using the Synopsys Design compiler v2019.03. In particular, the `compile_ultra` directive was used to generate the netlists for all constructions except GIFT-COFB-SER-TI whose hierarchy is conserved via the `no_autoungroup` flag which ensures that entity boundaries are preserved preventing any security-degrading optimization that may violate the threshold implementation properties. Power figures were calculated with the Synopsys Power Compiler that bases its analysis on back-annotated netlists created by running the target circuits through a comprehensive testbench.

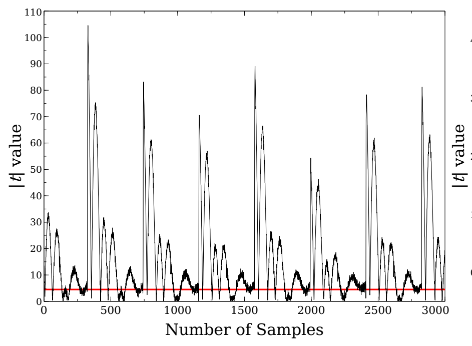
In Table 2, we tabulate the synthesis results for the proposed schemes and other bit-serial AEAD schemes for three different cell libraries. Namely, the relatively recent TSMC 28 nm process, the high-leakage NanGate 45 nm library and the comparably large STM 90 nm library.

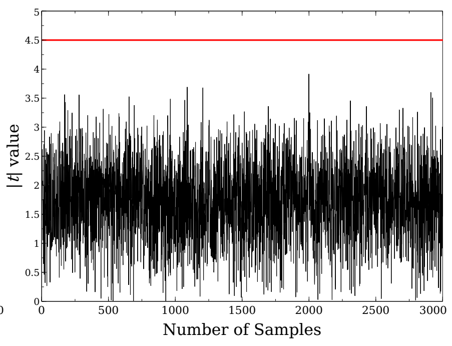
Naturally, due to the increased complexity of both Feed and the multiplier, GIFT-COFB-SER-F incurs a slightly larger circuit area than GIFT-COFB-SER-S which is offset by the latency savings as part of the parallelization of all component functions. We note that both GIFT-COFB-SER-S and GIFT-COFB-SER-F significantly undercut Romulus, the only other lightweight block cipher scheme among the NIST LWC finalists, in both area and power/energy.

[4] https://satoh.cs.uec.ac.jp/SAKURA/hardware/SAKURA-G.html



(a) A sample trace taken over 7 cycles.





(b) 20 thousand traces and masks off. (c) 10 million traces and masks off.

Fig. 13: Sample trace (top) and t-test results for the GIFT-COFB-SER-TI circuit (bottom). The red lines correspond to a threshold of $|t| = 4.5$.

7 Conclusion

In this paper, we investigated bit-serial architectures for the AEAD mode GIFT-COFB, a finalist in the NIST lightweight cryptography competition. In the process, we propose two architectures: the first follows a natural order of operations in which the finite field operations and other state updates are performed in the time period between 2 successive calls to the encryption module. The second absorbs all these operations in the last 128 cycles of the encryption operation, and saves 384 clock cycles in the processing of every block of associated data or message. We then extended the second architecture to construct a first order threshold implementation of GIFT-COFB. We verify the first-order security claims by performing statistical tests on power traces resulting from an implementation of the circuit on the SAKURA-G FPGA platform.

Acknowledgements. This project is partially supported by the European Union Horizon 2020 research and innovation program under the CPSoSAware project (grant 871738).

Table 2: Comprehensive synthesis figures for bit-serial AEAD schemes. Latency and energy correspond to the encryption of 128 bits of associated data and 1024 message bits. Highlighted schemes are NIST LWC finalists.

Scheme	Library	Area	Latency	Critical Path	Power (μW)		Energy (nJ)	
		(GE)	(Cycles)	(ns)	10 MHz	100 MHz	10 MHz	100 MHz
SUNDAE-GIFT	TSMC 28 nm	1732	92544	0.54	15.06	139.8	138.8	128.6
	NanGate 45 nm	1913	92544	1.28	52.42	271.71	485.1	251.4
	STM 90 nm	1201	92544	1.91	55.48	504.0	513.4	466.4
SAEAES	TSMC 28 nm	1927	24448	1.63	18.66	187.9	45.62	45.7
	NanGate 45 nm	2073	24448	3.05	61.16	329.7	149.5	80.6
	STM 90 nm	1350	24448	5.20	84.47	779.26	206.5	190.5
Romulus	TSMC 28 nm	2601	55431	0.54	24.16	225.2	133.9	124.8
	NanGate 45 nm	2878	55431	1.25	42.99	387.8	238.3	214.5
	STM 90 nm	1778	55431	2.29	82.28	796.8	456.1	441.2
SKINNY-AEAD	TSMC 28 nm	5335	72960	0.85	42.12	421.3	307.3	307.4
	NanGate 45 nm	5976	72960	1.31	167.4	861.2	1218	628.3
	STM 90 nm	3589	72960	2.02	143.7	1437	1048	1048
GIFT-COFB-SER-S	TSMC 28 nm	2095	54784	0.97	15.61	144.1	85.52	79.31
	NanGate 45 nm	2308	54784	1.40	55.25	245.5	302.7	135.1
	STM 90 nm	1443	54784	2.97	50.11	495.3	274.5	272.4
GIFT-COFB-SER-F	TSMC 28 nm	2148	51328	1.12	18.83	174.9	96.89	89.99
	NanGate 45 nm	2365	51328	2.22	66.62	343.55	342.8	176.8
	STM 90 nm	1485	51328	3.66	62.15	627.0	319.8	322.6
GIFT-COFB-SER-TI	TSMC 28 nm	4821	51328	1.15	42.30	393.2	217.7	202.3
	NanGate 45 nm	5317	51328	2.31	149.01	777.4	766.7	400.0
	STM 90 nm	3384	51328	3.74	158.1	1437	813.5	739.4

References

1. Nist lightweight cryptography project. https://csrc.nist.gov/projects/lightweight-cryptography
2. Balli, F., Banik, S.: Six shades of AES. In: Buchmann, J., Nitaj, A., Rachidi, T. (eds.) Progress in Cryptology - AFRICACRYPT 2019 - 11th International Conference on Cryptology in Africa, Rabat, Morocco, July 9-11, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11627, pp. 311–329. Springer (2019). https://doi.org/10.1007/978-3-030-23696-0_16
3. Balli, F., Caforio, A., Banik, S.: The area-latency symbiosis: Towards improved serial encryption circuits. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(1), 239–278 (2021). https://doi.org/10.46586/tches.v2021.i1.239-278
4. Banik, S., Balli, F., Regazzoni, F., Vaudenay, S.: Swap and rotate: Lightweight linear layers for spn-based blockciphers. IACR Trans. Symmetric Cryptol. **2020**(1), 185–232 (2020). https://doi.org/10.13154/tosc.v2020.i1.185-232
5. Banik, S., Bogdanov, A., Peyrin, T., Sasaki, Y., Sim, S.M., Tischhauser, E., Todo, Y.: Sundae-gift v1.0. NIST Lightweight Cryptography Project (2019), https://csrc.nist.gov/Projects/lightweight-cryptography/round-2-candidates
6. Banik, S., Bogdanov, A., Regazzoni, F.: Exploring energy efficiency of lightweight block ciphers. In: Selected Areas in Cryptography - SAC 2015 - 22nd International Conference, Sackville, NB, Canada, August 12-14, 2015, Revised Selected Papers. pp. 178–194 (2015). https://doi.org/10.1007/978-3-319-31301-6_10

7. Banik, S., Bogdanov, A., Regazzoni, F.: Atomic-aes: A compact implementation of the AES encryption/decryption core. In: Progress in Cryptology - INDOCRYPT 2016 - 17th International Conference on Cryptology in India, Kolkata, India, December 11-14, 2016, Proceedings. pp. 173–190 (2016). https://doi.org/10.1007/978-3-319-49890-4_10
8. Banik, S., Bogdanov, A., Regazzoni, F.: Compact Circuits for Combined AES Encryption/Decryption. Journal of Cryptographic Engineering pp. 1–15 (2017). https://doi.org/10.1007/s13389-017-0176-3
9. Banik, S., Chakraborti, A., Iwata, T., Minematsu, K., Nandi, M., Peyrin, T., Sasaki, Y., Sim, S.M., Todo, Y.: Gift-cofb v1.0. NIST Lightweight Cryptography Project (2019), https://csrc.nist.gov/Projects/lightweight-cryptography/round-2-candidates
10. Banik, S., Pandey, S.K., Peyrin, T., Sasaki, Y., Sim, S.M., Todo, Y.: GIFT: A small present - towards reaching the limit of lightweight encryption. In: Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings. pp. 321–345 (2017). https://doi.org/10.1007/978-3-319-66787-4_16
11. Beierle, C., Jean, J., Kölbl, S., Leander, G., Moradi, A., Peyrin, T., Sasaki, Y., Sasdrich, P., Sim, S.M.: The SKINNY family of block ciphers and its low-latency variant MANTIS. In: Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II. pp. 123–153 (2016). https://doi.org/10.1007/978-3-662-53008-5_5
12. Beierle, C., Jean, J., Kölbl, S., Leander, G., Moradi, A., Peyrin, T., Sasaki, Y., Sasdrich, P., Sim, S.M.: Skinny-aead and skinny-hash. NIST Lightweight Cryptography Project (2019), https://csrc.nist.gov/Projects/lightweight-cryptography/round-2-candidates
13. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.: PRESENT: an ultra-lightweight block cipher. In: Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings. pp. 450–466 (2007). https://doi.org/10.1007/978-3-540-74735-2_31
14. Caforio, A., Collins, D., Banik, S., Regazzoni, F.: A Small GIFT-COFB: Lightweight Bit-Serial Architectures (Repository) (5), https://github.com/qantik/cofbserial
15. Chakraborti, A., Iwata, T., Minematsu, K., Nandi, M.: Blockcipher-based authenticated encryption: How small can we go? J. Cryptol. **33**(3), 703–741 (2020). https://doi.org/10.1007/s00145-019-09325-z
16. Daemen, J., Rijmen, V.: The Design of Rijndael: AES - The Advanced Encryption Standard. Information Security and Cryptography, Springer (2002). https://doi.org/10.1007/978-3-662-04722-4
17. Dhooghe, S., Nikova, S., Rijmen, V.: Threshold implementations in the robust probing model. In: Proceedings of ACM Workshop on Theory of Implementation Security Workshop. pp. 30–37 (2019)
18. Iwata, T., Khairallah, M., Minematsu, K., Peyrin, T.: Romulus v1.2. NIST Lightweight Cryptography Project (2019), https://csrc.nist.gov/Projects/lightweight-cryptography/round-2-candidates
19. Jati, A., Gupta, N., Chattopadhyay, A., Sanadhya, S.K., Chang, D.: Threshold implementations of GIFT: A trade-off analysis. IEEE Trans. Information Forensics and Security **15**, 2110–2120 (2020). https://doi.org/10.1109/TIFS.2019.2957974

20. Jean, J., Moradi, A., Peyrin, T., Sasdrich, P.: Bit-sliding: A generic technique for bit-serial implementations of spn-based primitives - applications to aes, PRESENT and SKINNY. In: Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings. pp. 687–707 (2017). https://doi.org/10.1007/978-3-319-66787-4_33
21. Moradi, A., Poschmann, A., Ling, S., Paar, C., Wang, H.: Pushing the limits: A very compact and a threshold implementation of AES. In: Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings. pp. 69–88 (2011). https://doi.org/10.1007/978-3-642-20465-4_6
22. Naito, Y., Mitsuru Matsui and, Y.S., Suzuki, D., Sakiyama, K., Sugawara, T.: SAEAES. NIST Lightweight Cryptography Project (2019), https://csrc.nist.gov/Projects/lightweight-cryptography/round-2-candidates
23. Nikova, S., Rijmen, V., Schläffer, M.: Secure hardware implementation of non-linear functions in the presence of glitches. J. Cryptology **24**(2), 292–321 (2011). https://doi.org/10.1007/s00145-010-9085-7
24. Schneider, T., Moradi, A.: Leakage assessment methodology. In: International Workshop on Cryptographic Hardware and Embedded Systems. pp. 495–513. Springer (2015)

A Swap-and-Rotate GIFT-128 State Pipeline

In Table 3, we give the exact placement and activation periods of the nine swaps that implement the *swap-and-rotate* GIFT-128 permutation Π as specified in the work by Banik et al. [3].

Table 3: *Swap-and-rotate* listing of all swaps and their activation cycles.

Swap	Cycles
$\mathtt{FF}_{96} \leftrightarrow \mathtt{FF}_{97}$	Cycle mod 8 = 5
$\mathtt{FF}_{96} \leftrightarrow \mathtt{FF}_{98}$	Cycle mod 8 = 5
$\mathtt{FF}_{96} \leftrightarrow \mathtt{FF}_{99}$	Cycle mod 8 = 5 or Cycle mod 8 = 7
$\mathtt{FF}_{99} \leftrightarrow \mathtt{FF}_{101}$	6, 7, 10, 11, 14, 15, 18, 19, 22, 23, 26, 27, 30, 31, 34 35, 72, 73, 80, 81, 88, 89, 96, 97
$\mathtt{FF}_{99} \leftrightarrow \mathtt{FF}_{103}$	0, 1, 42, 43, 50, 51, 58, 59, 66, 67, 104, 105, 112, 113, 120, 121
$\mathtt{FF}_{99} \leftrightarrow \mathtt{FF}_{105}$	74, 75, 82, 83, 90, 91, 98, 99
$\mathtt{FF}_{105} \leftrightarrow \mathtt{FF}_{111}$	6, 7, 18, 19, 28, 29, 38, 39, 50, 51, 60, 61, 70, 71, 82, 83, 92, 93, 102, 103, 114, 115, 124, 125
$\mathtt{FF}_{105} \leftrightarrow \mathtt{FF}_{117}$	4, 5, 26, 27, 36, 37, 58, 59, 68, 69, 90, 91, 100, 101, 122, 123
$\mathtt{FF}_{105} \leftrightarrow \mathtt{FF}_{123}$	2, 3, 34, 35, 66, 67, 98, 99

B ANF Equations of the 3-Share GIFT-128 S-Box

Below we list the exact ANF equations for all component functions of the 3-share first-order threshold implementation of the GIFT S-box as proposed in [19].

$$\begin{aligned}
\mathtt{S}_{G_1}(a_2,b_2,c_2,d_2,a_3,b_3,c_3,d_3) &= a_3+b_3+b_2c_2+b_2c_3+b_3c_2,\\
&c_3+1,\\
&b_3+a_2c_2+a_2c_3+a_3c_2,\\
&a_3+b_3+c_3+d_3+a_2b_2+a_2b_3+a_3b_2;\\
\mathtt{S}_{G_2}(a_1,b_1,c_1,d_1,a_3,b_3,c_3,d_3) &= a_1+b_1+b_1c_3+b_3c_1+b_3c_3,\\
&c_1,\\
&b_1+a_1c_3+a_3c_1+a_3c_3,\\
&a_1+b_1+c_1+d_1+a_1b_3+a_3b_1+a_3b_3;\\
\mathtt{S}_{G_3}(a_1,b_1,c_1,d_1,a_2,b_2,c_2,d_2) &= a_2+b_2+b_1c_1+b_1c_2+b_2c_1\\
&c_2,\\
&b_2+a_1c_1+a_1c_2+a_2c_1,\\
&a_2+b_2+c_2+d_2+a_1b_1+a_1b_2+a_2b_1;\\
\mathtt{S}_{F_1}(a_2,b_2,c_2,d_2,a_3,b_3,c_3,d_3) &= d_3+a_2b_2+a_2b_3+a_3b_2,\\
&b_3+c_3+d_3+a_2d_2+a_2d_3+a_3d_2+1,\\
&a_3+b_3,\\
&a_3+1;\\
\mathtt{S}_{F_2}(a_1,b_1,c_1,d_1,a_3,b_3,c_3,d_3) &= d_1+a_1b_3+a_3b_1+a_3b_3,\\
&b_1+c_1+d_1+a_1d_3+a_3d_1+a_3d_3,\\
&a_1+b_1,\\
&a_1;\\
\mathtt{S}_{F_3}(a_1,b_1,c_1,d_1,a_2,b_2,c_2,d_2) &= d_2+a_1b_1+a_1b_2+a_2b_1,\\
&b_2+c_2+d_2+a_1d_1+a_1d_2+a_2d_1,\\
&a_2+b_2,\\
&a_2.
\end{aligned}$$