Low-latency Meets Low-area: An Improved Bit-Sliding Technique for AES, SKINNY and GIFT

Fatih Balli, Andrea Caforio and Subhadeep Banik

LASEC, École Polytechnique Fédérale de Lausanne, Switzerland
fatih.balli@epfl, andrea.caforio@epfl.ch, subhadeep.banik@epfl.ch

Abstract. The *bit-sliding* work of Jean et al. (CHES 2017) showed that the smallest-size circuit for SPN based blockciphers such as AES, SKINNY and, PRESENT can be achieved via bit-serial implementations. Their technique decreases the bitsize of the datapath, and it naturally leads to significant loss in latency (as well as the maximum throughput). Their designs complete a single round of the encryption in 168, 168 (for 128-bit blocks), 68 clock cycles (for 64-bit block) respectively. A follow-up work by Banik et al. (FSE 2020) introduced the *swap-and-rotate* technique that both eliminates this loss in latency and achieves even smaller footprint.
In the paper, we extend these results on bit-serial implementations all the way to three authenticated encryption schemes from NIST LWC. Our first focus is to decrease latency and improve throughput with the use of *swap-and-rotate* technique. Our blockcipher implementations have the most efficient round operations in the sense that a round function of a n-bit blockcipher is computed in exactly n clock cycles. This leads to implementations that are similar in size to the state-of-the-art, but have much lower latency (savings up to 20 percent).
Though these results are promising, blockciphers themselves are not end-user primitives, as they need to used together with a mode of operation. Hence, in the second part of the paper, we use our blockciphers in bit-serial implementations for three active NIST authenticated encryption candidates: SUNDAE-GIFT, Romulus and SAEAES. We provide the smallest blockcipher-based authenticated encryption circuits known in the literature so far.

Keywords: lightweight, latency, swap, rotate, blockcipher, authenticated encryption, NIST LWC, AES, SKINNY, GIFT

1 Introduction

The number of applications that rely on standard cryptographic primitives to establish a secure communication and transmit their data grow larger each day. Given the large variety of platforms these applications are run by, the available device resources spread to a large spectrum. In this spectrum, those applications related to IoT, sensor networks, RFID, vehicle-to-vehicle communication, suffer from having tight budget for security. These applications hence rely on the future promise by the emerging field *lightweight cryptography*, where new standards aiming primitives that are cheaper in terms of hardware footprint, energy, power and latency are anticipated. NIST's ongoing standardization process for lightweight cryptography (NIST LWC) clearly reflects the vivid effort in this domain. Candidates, 32 of which are elected for the second round, provide new designs for an authenticated encryption primitive, with the final goal of attaining security and lightweighness at the same time.

Lightweight metric. In the crypto community, the definition of security is defined with rigor, generally by resorting to algorithmic games. However, the term *lightweight* does not

candidate	primary cipher	primary size (block+key)	alternative cipher	alternative size (block+key)	cipher calls
COMET	AES	128+128	CHAM, Speck	64/128+128	enc
ESTATE	TweAES	128+128	TweGIFT	128+128	enc+dec
ForkAE	ForkSKINNY	128+288	ForkSKINNY	64/128+192/256/288	enc+dec
GIFT-COFB	GIFT*	128+128	-	-	enc
Hyena	GIFT	128+128	-	-	enc
LOTUS-AEAD	TweGIFT	64+128	-	-	enc
mixFeed	AES	128+128	-	-	enc
Pyjamask	Pyjamask	128+128	Pyjamask	96+128	enc
Romulus	SKINNY	128+384	SKINNY	128+256/384	enc
SAEAES	AES	128+128	-	-	enc
Saturnin	Saturnin	256+256	-	-	enc
SKINNY-AEAD	SKINNY	128+384	SKINNY	128+256/384	enc
SUNDAE-GIFT	GIFT*	128+128	-	-	enc

Table 1: The list of second-round candidates of NIST LWC that are based on blockciphers. GIFT refers to the original blockcipher from [BPP+17], and GIFT* refers to the version that assumes different ordering of the input bits and the key [BCI+19, BBP+19].

hold up to the same precision.

First, there is a question of which metric should be focused. In practice, each application is concerned most about the particular implementation aspect of the cryptographic primitive that conflicts with its own tight-budget. For battery-powered sensors, the energy spent per encrypted bit is a matter of priority, and directly determines how long the battery lasts. Sometimes, multiple metrics must be taken into consideration. As an example, an RFID device might prioritize power, but at the same time the silicon area footprint is also crucial.

Secondly, there is a question of how to quantify this metric. This is highly dependent on how the scheme is implemented, e.g. as an application-specific integrated circuit (ASIC), with field-programmable gate array (FPGA) or programmed to a microprocessor. This further depends on the specific details, e.g. the transistor-size of the cell library in ASIC plays a key role in area, power consumption as well as response time of the gates.

In this paper, the term *lightweight* refers to the combination of both the area (in terms of gate-equivalence GE) and the latency (in terms of clock cycles) of an architecture realized as ASIC. We believe that these two metrics are most easily reproducible, in the sense that given two implementations A and B, there is a methodological and fair approach to decide which one performs better[1]. In our implementations, we prioritize area, and take the latency as the second goal. Therefore, our goal is to remain in the same area budget of the ultra-small (i.e. 1-bit datapath) implementations and reduce the latency (i.e. the number of clock cycles) as much as possible.

Reducing latency is an important goal, as it directly translates into a gain in the throughput, as well as a reduction in the energy consumption[2]. Given the variety of application profiles, it is worth noting that the ideal goal from an implementation is to lower the cost in one or few metrics so that the design is sufficient to meet the expectation of most (if not all) applications.

Mode of operations. Among the active second round candidates, 13 out of 32 are based on blockciphers. These candidates simply design a mode of operation around a given blockcipher to function as an authenticated encryption scheme. From Table 1, note that:

[1] GE measurements are not perfectly reliable across different technology libraries. For fairness, we compare GE-sizes of two implementations A and B synthesized in the same technology library.

[2] The energy consumption is lower, if we assume that the power consumption remains same. This holds true for our most of our implementations.

- 4 candidates use either the standard or a tweaked version of AES as their primary choice. These are COMET, ESTATE, mixFeed, SAEAES [GJN19, CDJ+19a, CN19, NMMaS+19].

- 3 candidates use either the standard or a forked version of SKINNY as their primary choice. These are ForkAE, Romulus, SKINNY-AEAD [ALP+19, IKMP19, BJK+19].

- 2 candidates use either the standard or a tweaked version of 128-bit variant of GIFT. Namely, ESTATE uses the tweaked version of GIFT as the alternative choice [CDJ+19a], whereas Hyena uses the original version [CDJN19, BPP+17]. Besides these two, LOTUS-AEAD also employs the 64-bit variant of GIFT [CDJ+19b, BPP+17].

- 2 candidates use GIFT*. These two candidates are GIFT-COFB, SUNDAE-GIFT [BCI+19, BBP+19]. The difference between GIFT* and GIFT is that the latter assumes a different indexing of the input and output bits. We denote their modified version with GIFT*, as it leads to a significant difference from design and implementation perspective.

- Pyjamask and Saturnin are the exceptions to the popular approach, as they bring their own dedicated blockcipher design into the standardization [GJK+19, CDL+19].

Given the modular approach taken by these candidates, one can pose the two following questions, from the lightweight perspective. (1) How lightweight is the blockcipher employed at the core? (2) What is the cost of the surrounding mode of operation? Our work and results are tightly related to these two important questions.

Bottleneck of storage. Most low-area implementations of SPN-based blockciphers eventually face a dilemma of storage. Namely, all implementations (except full unrolling) need to store the key and the cipher state during the encryption operation. For a blockcipher with ℓ_b-bit block and ℓ_k-bit key, this requires the use of $\ell_b + \ell_k$ flip-flops. More concretely, we measured the area-cost of storing 256-bit as 1088 GE, 832 GE and 1451 GE for the cell libraries UMC90, STM90 and Nangate45 respectively. It naturally follows that for the smallest implementations of AES (resp. SKINNY-128-128, GIFT), the 73% (resp. 83%, 69%) of the circuit is due to merely D flip-flops [JMPS17, BPP+17]. Therefore, the state-of-the-art ultra-small ASIC implementations of blockciphers contain mostly storage elements, and space for further area optimizations is limited. We take this as an indicator that we should divert our focus to the other aspects of the circuit while remaining in the same area budget.

Contributions. In the first part of the paper, we provide 1-bit serial architectures for the popular 128-bit blocksize variants of the blockciphers AES, SKINNY, GIFT, and GIFT*, which are popular among NIST candidates. Our implementations can be employed by 10 candidates out of 13 listed in Table 1. Our approach have the following benefits, and the detailed comparison with the state-of-the-art is given in Table 2:

- In terms of circuit area, each of our blockcipher implementations is an evident contender to be the smallest implementation.

- Each implementation fully utilizes both the state and the key pipelines. Each round consisting of 128-bit is executed exactly in 128 clock cycles. This ensures that we get the maximum throughput from 1-bit serial implementation. This leads to approximately a 20% reduction in latecy (in clock cycle units) over the circuits reported in [JMPS17, BPP+17] (note that the AES, SKINNY, GIFT circuits in these papers report a latency of 168, 168, 160 cycles per round respectively). Our circuit design is novel in the sense that both pipelines are continuously active.

blockcipher	Area (GE) Nangate45	UMC90	Power (μW) @ 100 KHz Nangate45	UMC90	Latency round	total	Energy (nJ/128-bit) Nangate45	UMC90	Ref.
AES (standard)	1974	1600	102.4	0.55	**128**	**1408**	**1441.8**	**7.7**	Sec. 3
AES (non-standard)	1982	1596	100.2	0.67	168	1776	1779.6	11.9	[JMPS17]
AES (standardized)[a]	2029	1641	-	-	168	1904			[JMPS17]
SKINNY-128-128	1748	1355	87.69	0.43	**128**	**5248**	**4602.0**	**22.6**	Sec. 4
SKINNY-128-128	1740	1363	86.66	0.56	168	6976	6045.4	39.1	[JMPS17]
SKINNY-128-256	2502	1927	124.96	0.62	**128**	**6272**	**7837.5**	**38.9**	Sec. 4
SKINNY-128-256	2501	1937	123.49	1.15	168	8448	10432.4	97.2	[JMPS17]
SKINNY-128-384	3263	2518	162.79	0.82	**128**	**7296**	**11877.2**	**59.8**	Sec. 4
SKINNY-128-384	3260	2508	160.03	1.55	168	9920	15875.0	153.8	[JMPS17]
	STM90	UMC90	Power (μW) @ 10 MHz STM90	UMC90			STM90	UMC90	
GIFT	1215	1531	51.8	51.3	**128**	**5248**	27.2	26.9	Sec. 5.3
GIFT	1213	-	40.3	-	160	6528	26.3		[BPP+17]
GIFT* [b]	1108	1332	49.8	48.7	**128**	**5248**	**26.1**	**25.5**	Sec. 5

Table 2: The comparison of our work with the state-of-the-art in terms of latency, area, power and energy. The measurements respect to the use of the same library and clock frequency, Nangate45 and UMC90 for AES, SKINNY and STM90 for GIFT*. For UMC90, the power measurements are taken with low leakage cell library. For comparison purposes, all power figures are measured at frequencies as reported in the original papers [JMPS17, BPP+17]. [a]It has been estimated in [MPL+11] that converting a non-standardized to a standardized circuit requires an additional 20 muxes. The area figures in this row is obtained by adding the area of 20 muxes to the figures in the previous row. [b]GIFT* refers to the slightly modified version of GIFT used in SUNDAE-GIFT [BBP+19].

- Each implementation respects the standard ordering of input and output bits. We do not make a non-standard assumption on the ordering of the bits to reduce the area and latency. Namely, we ensure that an implementation from our paper is readily usable from a NIST LWC candidates without having to modify and deal with the ordering the bits. Some implementations of AES, e.g. [MPL+11, JMPS17, BBR16, BBR17], assume that plaintext and the key is arranged in a row major fashion (which we call non-standard), even though the original specification of AES assumes a column major arrangement [DR02].
- We avoid techniques such as clock-gating, which might sometimes result in timing inconsistencies during synthesis phase and cost additional circuit area. We also use simple D flip-flops that are smaller than their siblings (with added enable or reset functionalities) in some libraries.

In the second part of the paper, we direct our attention to implementation of three AE schemes, one for each blockcipher: SAEAES (1350 GE), Romulus (1779 GE) and SUNDAE-GIFT (1108 GE). We have chosen these candidates, because the mode of operation part of the circuit has the minimal storage requirement, thus leading to very compact implementations. To the best of our knowledge these are the smallest blockcipher-based authenticated encryption schemes reported so far[3].

2 Preliminaries

Notation. We denote the set of integers $\{a, a+1, \ldots, b\}$ with $[a, b]$. Furthermore, we define $(a, b) = [a, b] \setminus \{a, b\}$, $(a, b] = [a, b] \setminus \{a\}$ and $[a, b) = [a, b] \setminus \{b\}$. $[n]$ is a shorthand

[3]It is authors' intention to make the source code available and cite the source. For anonymity reasons, the citation is removed and the HDL source code of the implementations are submitted as supporting document along with this submission.

for $[0, n]$. The bit string concatenation is denoted with $||$.

Pipelines. In a circuit, a bit-serial pipeline is a series of flip flops that are arranged in a way that allows the stored value to be shifted in a fixed direction, one bit position at each clock cycle. In mathematical terms, an n-bit pipeline can be denoted with a series of 1-bit variables $\mathsf{FF}_0, \mathsf{FF}_1, \ldots, \mathsf{FF}_{n-1}$ that support two main operations: *shift* and *swap* (that takes two hard-wired indices u and v as extra input). This corresponds to storage elements (flip flops) in circuit, and we simply write FF to represent this series. The two operations could be better explained with the following algorithmic descriptions, which take the previous state FF and returns the updated state FF':

shift(FF, b_{in}):

$\mathsf{FF}'_{n-1} \leftarrow b_{\mathsf{in}}$
$\mathsf{FF}'_{i-1} \leftarrow \mathsf{FF}_i$ for $i \in [1, n)$
$b_{\mathsf{out}} \leftarrow \mathsf{FF}_0$
return $(\mathsf{FF}', b_{\mathsf{out}})$

swap^{u,v}(FF):

$\mathsf{FF}'_i \leftarrow \mathsf{FF}_i$ for $i \in [0, n) \setminus \{u, v\}$
$\mathsf{FF}'_u \leftarrow \mathsf{FF}_v$
$\mathsf{FF}'_v \leftarrow \mathsf{FF}_u$
return FF'

Here, the shift operation additionally takes a new bit to be loaded into the first flip flop FF_{n-1} of the pipeline, and outputs the bit at the last flip flop FF_0[4]. The swap operation on the other hand, which we simply refer with a tuple (u, v) hereafter, only swaps the contents of FF_u and FF_v and does not touch other values.

What makes these two operations special is that (1) bit-sliding implementations already contain a pipeline, hence a shift operation is already available, and (2) a swap operation (u, v) can be implemented by replacing two regular flip flops with two scan flip flops, which brings a quite small cost to the circuit in terms of area footprint. However, it must be noted that u and v values must be hard coded into the swap operation, as the positions of scan flips are fixed. Therefore, each distinct (u, v) swap brings an extra cost.

An interesting observation is that given any single swap operation (u, v) (with $u \neq v$ and $gcd(u - v, n) = 1$) and the shift operation, one could execute an arbitrary permutation of the bits through a finite calls of shift and swap calls, as pointed out by Banik et al. [BBRV19]. Therefore, in theory, one could implement any permutation layer in any block cipher through a single swap operation as long as the the correct sequence of shift and swap calls are made by the controller circuit. However, this approach would take large number of shift calls, hence lead to unrealistically many number of clock cycles. A good approach therefore is to add few other swap operations to strike a balance between the circuit size and the reasonable latency.

3 AES

For the rest of this section, we assume familiarity with the round function and the key scheduling algorithms of AES [DR02]. Our circuit simply consists of the following circuits in the main hierarchy: (1) a state pipeline, (2) a key pipeline, (3) a controller, (4) a shared S-box.

3.1 State Pipeline

The state in our design uses the following components/techniques:

- nibble-level MixColumns circuit introduced by Jean et al. [JMPS17],
- the smallest known S-box "bonus" from Maximov et al. [ME19],

[4]The indexing might come as counter intuitive at the first sight. However, if we load the bit sequence $b_0, b_1, \ldots, b_{n-1}$ to the pipeline respectively, after n shift calls, each FF_i stores exactly b_i.

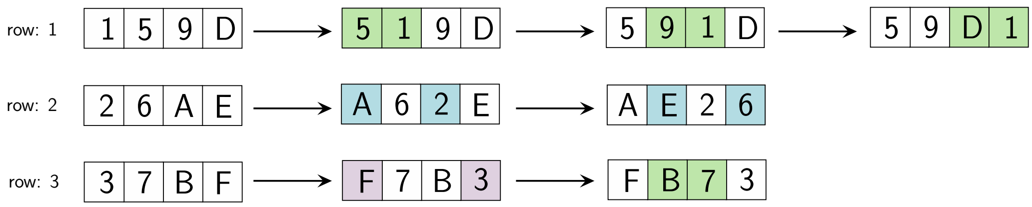

Figure 1: The transition diagram for rows 1, 2, 3; where the colored cells denote the recently-modified values. Note that there are three distinct swap operations, with distance 0, 1 and 2 cells in-between.

One can easily notice that given that state and key bits are stored in a pipelined fashion, AddRoundKey can be performed without much hassle as long as each of the state and key pipelines produces the correct bit per clock cycle. Hence, the main challenges on the state pipeline part is to (1) execute all SubBytes, ShiftRows, MixColumns operations simultaneously, (2) complete the operations in 128 clock cycles, while (3) following the standard ordering of bits for the plaintext and the key. Below, we first describe each layer separately, and show how we can fuse them into one operation that executes over the state pipeline continuously.

3.2 ShiftRows with swaps

Assume that the 128-bit pipeline is defined in the same fashion in Section 2, i.e. the bits are loaded into FF_{127} and they are flushed out by FF_0. We use three swap operations to execute the ShiftRows layer: $(80, 112)$, $(56, 120)$ and $(25, 121)$. The timetable for scheduling these swaps are given in Table 3. Below, we explain how we came up with these swap sequences and the mechanism in which they work for shifting rows correctly.

For simplicity, let us forget about the pipeline and shift operations for the moment, and focus on the nature of ShiftRows in the 16-byte state. We try to express ShiftRows in terms of byte-swaps. Suppose that the values contained in the state are the hexadecimal characters $\mathsf{0}, \mathsf{1}, \ldots, \mathsf{F}$. Considering the standard byte arrangement for loading the initial data [FIP], row 0 contains the values $\mathsf{0}, \mathsf{4}, \mathsf{8}, \mathsf{C}$; row 1 contains the values $\mathsf{1}, \mathsf{5}, \mathsf{9}, \mathsf{D}$ etc. We then devise a sequence of swap operations over the rows 1, 2, 3 to perform ShiftRows. Our three distinct swaps are denoted with distinct colors in Figure 1. This figure shows the movement of the bytes as they arrive to their final position implied by ShiftRows.

We point out two important observations: (1) each byte-swap operation can be executed by a bit-swap circuit through 8 consecutive calls interleaved by shift operation, (2) the swap operations denoted with the same color can actually be executed by a single swap operation as long as it is enabled in the correct clock cycle. Therefore, the choice of swaps and the timetable in Table 3 are straightforward extensions of this example into the 128-bit state pipeline.

To help understand how the structure helps perform the ShiftRows operation, we note that since the pipeline is always evolving, the shift operation is performed in every clock cycle. To additionally perform the swap $80 \leftrightarrow 112$, in any clock cycle, we need to place scan flip-flops at locations 79 and 111 (and wire the output of 80 to the input of 111; wire the output of 112 to the input of 79) as is shown in Figure 2. So assuming that the bits indexed 0 to 127 enter the pipelines through the location 127. At clock cycle $k \leq 127$, the pipeline stores exactly k bits. For instance, at 56-th clock cycle, the bits indexed 8, 40 are at locations 80, 112 respectively. Effecting swaps for cycles 56 to 63 therefore swaps bits $8, \ldots, 15$ with $40, \ldots, 47$, which is essentially bytes indexed 1 and 5. It can be verified without difficulty that performing the same swap in cycles $[88, 96)$, swaps bytes 1 and 9. Similarly the same swap in cycles $[120, 127)$ swaps bytes 1 with 13, which completes the

pipeline	operation	active cycles
state	swap (80, 112)	$[56,64) \cup [88,96) \cup [120,127] \cup [8,16)$
	swap (56, 120)	$[88,96) \cup [120,127] \cup [0,8)$
	swap (25, 121)	$\{127\} \cup [0,6]$
	load S-box	$\{8k+7 : k \in [0,15]\}$
	load Mix Col.	$[32,40) \cup [64,72) \cup [96,104) \cup [0,8)$
key	swap (96, 128)	$[0,8)$
	swap (40, 72)	$[56,64)$
	load S-box	$\{112\} \cup \{120\} \cup \{0\} \cup \{8\}$
	key XOR	$[0,96)$
	add RC	(look up table)

Table 3: The timetable of operations for bit-serial AES encryption.

ShiftRows operation on row 1. It is not too difficult to verify that the other swaps at cycles as listed in Table 1 faithfully perform the remaining ShiftRows operations.

3.3 The nibble MixColumns

The nibble MixColumns was introduced by Jean et al. [JMPS17]. The multiplication over a single column is completed over 8 clock cycles, updating each nibble at a time. To simplify, we first represent a single column of bytes as 8 vertical nibble vectors as below. Namely, from the pipeline given in Figure 2, the vectors M_i are defined for $0 \le i \le 7$ as below:

$$M_i := \begin{bmatrix} \mathsf{FF}_i \\ \mathsf{FF}_{i+8} \\ \mathsf{FF}_{i+16} \\ \mathsf{FF}_{i+24} \end{bmatrix} \qquad \mathcal{R}(M_0) := \begin{bmatrix} \mathsf{FF}_8 \\ \mathsf{FF}_{16} \\ \mathsf{FF}_{24} \\ \mathsf{FF}_0 \end{bmatrix}$$

The nibble MixColumns architecture employs an additional set of 4 flip-flops to help with the serialized computation of this functionality. Define the vector M_8 to denote this additional internal 4-bit storage this architecture employs. During its 8 clock cycle operation, these flip flops are used to keep the value of the leftmost bit of each one of the four bytes. We define a function *upward* rotation $\mathcal{R}$ that rotates the elements in a given vertical matrix by one position, as exemplified above. The circuit essentially performs the following sequence of operations to derive the new value of M_i for each $i = 0, 1, \ldots, 7$, starting from $i = 0$ respectively:

- if $i = 0$, store $M_8 \leftarrow M_0$ before any of the following computation.
- update $M_i \leftarrow \mathcal{R}(M_i) \oplus \mathcal{R}^2(M_i) \oplus \mathcal{R}^3(M_i) \oplus M_{i+1} \oplus \mathcal{R}(M_{i+1})$
- if $i \in \{3, 4, 6\}$, further update $M_i \leftarrow M_i \oplus M_8 \oplus \mathcal{R}(M_8)$

In other words, at each clock cycle, based on the internal 7-bit counter, we can execute a single slice of the previous computation. In total, it takes 8 clock cycles for a single column, and 32 clock cycles for the whole MixColumns layer. This serial circuit can be realized with with 8 XOR, 8 NAND gates and 4 flip flops (see Figure 1 of [JMPS17]).

3.4 Combined state pipeline

In the controller, the circuit contains an 11-bit counter to keep both the round (4-bit) and the phase (7-bit). We split this counter into two parts and refer to them respectively by variables $0 \le \mathsf{round} \le 10$ for the upper 4-bit and $0 \le \mathsf{count} \le 127$ for the lower 7-bit.

Figure 2: The state (above) and key (below) pipelines of AES-128 encryption with colored scan flip-flops.

In contrast to previous work [JMPS17], we follow the standard ordering of bits in our implementation. That is given a plaintext and a key, the bits are loaded into the circuit starting from the leftmost bits, and following the natural order [FIP]. This becomes a crucial aspect of a blockcipher implementation, if it is meant to be used in a mode of operation that needs to comply with a fixed standard.

At the beginning of its operation, the 11-bit counter is reset to zero. During initialization, i.e. $\mathsf{round} = 0$, the white-colored MUXes in Figure 2 are configured so that the next bit s of the state is received from the input port PT but after the XOR is performed with KEY, which is also being loaded at the same time. For $\mathsf{round} > 0$, we select the state bit to be loaded from the exit of the state pipeline.

SubBytes Meanwhile, we proceed with executing the SubBytes layer, by enabling the S-box at every 8-th cycle. More precisely, the S-box is configured to take $\mathsf{FF}_{121}, \mathsf{FF}_{122}, \ldots, \mathsf{FF}_{127}, s$ as input, and the scan flip flops $\mathsf{FF}_{120}, \ldots, \mathsf{FF}_{127}$ are instructed to load the output from the S-box if $\mathsf{count} \bmod 8 = 7$.

ShiftRows Starting from $\mathsf{count} = 56$, the swap operations become active. Many of the bits need to make a couple of jumps before they are located into their ultimate position implied by ShiftRows, as demonstrated in Figure 1. Hence, position-wise, many bits are incorrectly located and look garbled as they pass through flip flops $\mathsf{FF}_{24}, \ldots, \mathsf{FF}_{120}$. Nonetheless as soon as they exit the last swap position FF_{24}, they are guaranteed to be in their final position. See Table 3 to notice that the last swap operation executed on a layer actually happens when $\mathsf{count} = 15$ in the next round. In other words, performing ShiftRows over the i-th state uses the last 72 cycles of the round i and the first 16 cycles of the round $i + 1$, and it is not aligned with the counter round itself .

MixColumns The input ports to the nibble MixColumns circuit are flip flops FF_i for $i \in \{0, 1, 8, 9, 16, 17, 24, 25\}$, and the output ports are input to the exit MUX of the pipeline and $\mathsf{FF}_7, \mathsf{FF}_{15}, \mathsf{FF}_{23}$ respectively. The MixColumns of round i is performed at $\mathsf{round} = i+1$ and it is active during $0 \le \mathsf{count} \bmod 32 \le 7$, except the last round where MixColumns must be skipped.

Resolving overlaps. Note that there are two clock cycles, i.e. count values, during which two operations modify the same FF simultaneously in Table 3. First, at clock cycle 127 both S-box and swap $(25, 121)$ attempts to overwrite FF_{120}. Here, the operation precedence is given to the S-box (as SubBytes comes before ShiftRows), meaning that the leftmost output bit of the S-box is fed to the swap operation (instead of FF_{120}). A second overlap occurs when $\mathsf{count} = 3$, as MixColumns circuit attempts to read FF_{25} before its value is updated correctly by the swap $(25, 121)$. Here, the precedence is given to the swap operations, meaning that the output of the swap operation is fed as input to MixColumns circuit (instead of FF_{25}).

3.5 Key Pipeline

Suppose that $K_0, K_1, \ldots, K_{15}$ represent the key bytes of a particular round. Then the next round key sequence $K_{16}, \ldots, K_{31}$ is computed as follows:

$$\begin{bmatrix} K_{16} & K_{20} & K_{24} & K_{28} \\ K_{17} & K_{21} & K_{25} & K_{29} \\ K_{18} & K_{22} & K_{26} & K_{30} \\ K_{19} & K_{23} & K_{27} & K_{31} \end{bmatrix} \leftarrow \begin{bmatrix} K_0 & K_4 & K_8 & K_{12} \\ K_1 & K_5 & K_9 & K_{13} \\ K_2 & K_6 & K_{10} & K_{14} \\ K_3 & K_7 & K_{11} & K_{15} \end{bmatrix} \oplus \begin{bmatrix} \mathsf{S}(K_{13}) \oplus \mathsf{rc} & K_{16} & K_{20} & K_{24} \\ \mathsf{S}(K_{14}) & K_{17} & K_{21} & K_{25} \\ \mathsf{S}(K_{15}) & K_{18} & K_{22} & K_{26} \\ \mathsf{S}(K_{12}) & K_{19} & K_{23} & K_{27} \end{bmatrix}$$

where rc denotes the round constant byte.

In summary, the first column requires special treatment, because it involves S-box calls, and the remaining three columns can be updated smoothly (by simply XORing with a neighboring bit). In particular, one can notice the disarrangement in the update of the first column, as it takes the current last columns bytes with a downward rotation (by one byte). If we implement this in a straightforward fashion by updating each byte when they arrive to position 0, we would have to choose the input of the S-box either from the position 13 (for computing K_{16}, K_{17}, K_{18}) or 9 (for computing K_{19}). This means that we would have to put an extra 8-bit MUX to choose which value needs to be fed to the S-box. Instead, we decided to temporarily move the byte K_{12} to position 13 before it is fed to S-box, and then return back to its original position after the S-box operation is done. Therefore the pipeline performs the following operations in sequence:

- In the first 8 clock cycles, we activate the swap $(96, 128)$ so that the key byte K_{12} is temporarily moved such that it comes after K_{15}. Here, FF_{128} actually refers to the new key bit that is about to be loaded into the key pipeline. With this operation, the key pipeline contains $K_{13}, K_{14}, K_{15}, K_{12}$ in this order. Hence, it respects the order they are being used to update the first key column.
- In clock cycles 112, 120 (of the current round) and 0, 8 (of the next round); the S-box is used by the key pipeline. During these cycles, the S-box reads $K_{13}, K_{14}, K_{15}, K_{12}$ from $\mathsf{FF}_{120}, \ldots, \mathsf{FF}_{127}$ in given order. The output from the S-box is XORed with $\mathsf{FF}_{16}, \ldots, \mathsf{FF}_{23}$ and the result is loaded into $\mathsf{FF}_{15}, \ldots, \mathsf{FF}_{22}$.
- The round constant is added as the bit FF_{24} is loaded into FF_{23}. We use a look-up table to decide when the round constant bit is enabled. In total, this bit is enabled 16 times during the whole encryption.
- During the clock cycles $[56, 64)$, we activate the swap $(40, 72)$ to return K_{12} back to its original relative position. Hence the internal ordering of the bytes becomes K_{12}, K_{13}, K_{14}, K_{15} again.
- For the rest of the key bits, by activating $\mathsf{FF}_{31} \leftarrow \mathsf{FF}_0 \oplus \mathsf{FF}_{32}$ during the clock cycles $[0, 96)$.

Library	Area (μm^2)	Area (GE)	Power (μW) @ 10 MHz	Latency round	Latency total	Energy (nJ)	Throughput Mbps	Ref.
STM 90	5562.6	1267	73.6	128	1408	10.4	14.43	Sec. 3
UMC 90	5016.8	1600	65.2	128	1408	9.2	13.45	Sec. 3
TSMC 90	4692.2	1663	56.1	128	1408	7.9	15.57	Sec. 3
Nangate 15	441.8	2247	18.4	128	1408	2.6	315.56	Sec. 3
Nangate 45	1575	1974	143	128	1408	20.1	49.25	Sec. 3

Table 4: AES-128 Enc only

pipeline	operation	active cycles
state	swap (112, 120)	$[112, 120) \cup [120, 127] \cup [0, 8) \cup [64, 72)$
	swap (104, 120)	$[64, 72) \cup [88, 96) \cup [96, 104)$
	swap (96, 120)	$[64, 72)$
	load S-box	$\{8k : k \in [0, 15]\}$
	rc addition.	(look-up table + lfsr)
	load Mix Col.	$[0, 32)$
tweakey 1,2,3	swap (56, 120)	$[72, 127] \cup [0, 8)$
	swap (48, 56)	$[120, 127]$
	swap (24, 56)	$[112, 120) \cup [120, 127] \cup [0, 8)$
	swap (8, 24)	$[120, 127] \cup [0, 8) \cup [24, 32)$
tweakey 2	swap (0, 1)	$[0, 6] \cup [8, 14] \cup [16, 22] \cup [24, 30] \cup [32, 38] \cup [40, 46] \cup [48, 54] \cup [56, 62]$
	lfsr xor	$\{8k : k \in [0, 7]\}$
tweakey 3	lfsr (8-bit)	$\{8k : k \in [0, 7]\}$

Table 5: The timetable of operations for bit-serial SKINNY-128-384 encryption.

4 SKINNY

SKINNY provides six different variants [BJK+16]. In this paper, we consider the the variants that are used by NIST LWC candidates, i.e. these are 128-bit blocksize variants, as given in Table 1. In these variants, the tweakey size is variable, i.e. it can consists of $128z$ bits for $z = 1, 2, 3$. For the remainder of the paper, we refer to these three versions by SKINNY-128-128, SKINNY-128-256 and SKINNY-128-384 respectively.

SKINNY is quite similar to AES in design, but it employs more lightweight operations for the round function. Prominently S-box and MixColumns can be realized with much smaller circuitry compared to AES. The round function consists of SubCells, AddConstants, AddRoundTweakey, ShiftRows, MixColumns. For the fine details of these layers, we refer the reader to the original SKINNY paper [BJK+16].

Hence, our design follows a similar architecture to that of AES. The circuit simply consists of the following parts in the main hierarchy: (1) a state pipeline (which includes a dedicated S-box), (2) a key pipeline, (3) a controller.

4.1 Combined state pipeline

In the controller, the circuit contains an 13-bit counter to keep both the round (6-bit) and the phase (7-bit). We split this counter into two parts and refer to them respectively by variables $0 \leq \mathsf{round} \leq 56$ for the upper 4-bit and $0 \leq \mathsf{count} \leq 127$ for the lower 7-bit.

Because SKINNY is already designed with hardware-friendliness in mind, we load the bits into the circuit starting from the leftmost bits, and following the standard [BJK+16]. In our implementations the key blocks and the plaintext are loaded simultaneously and completed in 128 cycles. This applies to all three versions of SKINNY-128-128, SKINNY-128-256, SKINNY-128-384.

At the beginning of its operation, the 13-bit counter is reset to zero. Then during initialization, i.e. $\mathsf{round} = 0$, the plaintext is loaded through 1-bit input port, and the key is loaded through z-bit input port into their respective pipelines without modification. Each tweakey block has its own dedicated input port. These ports are denoted with PT (for plaintext) and KEY1, KEY2, KEY3 for the tweakey. Below, we describe the layers of operations executed on the state pipeline, in an order observed by the incoming bits.

SubCells SubCells layer is executed by enabling the S-box at every 8-th cycle. More precisely, the S-box is configured to read $\mathsf{FF}_{120}, \mathsf{FF}_{121}, \ldots, \mathsf{FF}_{127}$ as input, and the scan flip flops $\mathsf{FF}_{119}, \ldots, \mathsf{FF}_{126}$ are instructed to be loaded with the S-box output if $\mathsf{count} \bmod 8 = 0$.

AddConstants The round constants are added right after the S-box operation. An XOR gate is placed between FF_{119} and FF_{120} and the round constant bit rc is added. We use a 7-bit LFSR circuit (not shown in the figure) to produce the round constant bit.

AddRoundTweakey The key bits are added at the same position with the round constant bit, i.e. between FF_{119} and FF_{120}. In order to synchronize this with the key pipeline, the key bits k_0, k_1, k_2 are read from FF_{120} of the key pipelines. The key addition is active during $8 \leq \mathsf{count} < 72$. This corresponds to adding the first half of each tweakey.

ShiftRows This layer is executed with 3 swap operations, similar to AES, and the timetable of swaps are given in Table 5.Position-wise, bits are incorrectly located and look garbled as they pass through flip flops $\mathsf{FF}_{95}, \ldots, \mathsf{FF}_{119}$, but as soon as they exit the last swap position FF_{95}, they are guaranteed to be in their final position.

MixColumns The input ports to the nibble MixColumns circuit are flip flops FF_i for $i \in \{0, 32, 64, 96\}$, and the output ports are input to the exit MUX of the pipeline and $\mathsf{FF}_{31}, \mathsf{FF}_{63}, \mathsf{FF}_{95}$ respectively. The MixColumns is active during the first 32 clock cycles of a round.

Resolving overlaps. Note that during clock cycles $64 \leq \mathsf{count} < 72$ three swaps $(112, 120)$, $(104, 120)$, $(96, 120)$ are active at the same time and overlap at the same flip flop FF_{120}. The order of execution here is $(96, 120)$, $(104, 120)$ and $(112, 120)$ respectively.

4.2 Key Pipeline

SKINNY can have up to three block of tweakey, referred to as $TK1, TK2, TK3$ [BJK+16]. The key schedule algorithm is quite similar in all three key blocks. More precisely, suppose that $K_0, K_1, \ldots, K_{15}$ represent the key bytes of a particular tweakey block. Then the next round key sequence $K_{16}, \ldots, K_{31}$ is computed as follows:

$$\begin{bmatrix} K_{16} & K_{17} & K_{18} & K_{19} \\ K_{20} & K_{21} & K_{22} & K_{23} \\ K_{24} & K_{25} & K_{26} & K_{27} \\ K_{28} & K_{29} & K_{30} & K_{31} \end{bmatrix} \leftarrow \begin{bmatrix} \mathcal{L}_i(K_9) & \mathcal{L}_i(K_{15}) & \mathcal{L}_i(K_8) & \mathcal{L}_i(K_{13}) \\ \mathcal{L}_i(K_{10}) & \mathcal{L}_i(K_{14}) & \mathcal{L}_i(K_{12}) & \mathcal{L}_i(K_{11}) \\ K_0 & K_1 & K_2 & K_3 \\ K_4 & K_5 & K_6 & K_7 \end{bmatrix}$$

where the operation $\mathcal{L}_i$ are 8-bit permutations given below:

$$\mathcal{L}_1(x_7||x_6||x_5||x_4||x_3||x_2||x_1||x_0) := x_7||x_6||x_5||x_4||x_3||x_2||x_1||x_0$$
$$\mathcal{L}_2(x_7||x_6||x_5||x_4||x_3||x_2||x_1||x_0) := x_6||x_5||x_4||x_3||x_2||x_1||x_0||(x_7 \oplus x_5)$$
$$\mathcal{L}_2(x_7||x_6||x_5||x_4||x_3||x_2||x_1||x_0) := (x_0 \oplus x_6)||x_7||x_6||x_5||x_4||x_3||x_2||x_1$$

Therefore, our key pipelines do the following operations in sequence. First, we swap the first and the last eight bytes by using the swap $(56, 120)$. Then we perform the local byte permutations on the upper half (i.e. the first 8 bytes) of the key through swaps $(48, 56)$, $(24, 56)$, $(8, 24)$. Finally we apply the 8-bit permutation $\mathcal{L}_2$ through another swap $(0, 1)$ for $TK2$, and a dedicated 8-bit LFSR circuit for $\mathcal{L}_3$ in $TK3$.

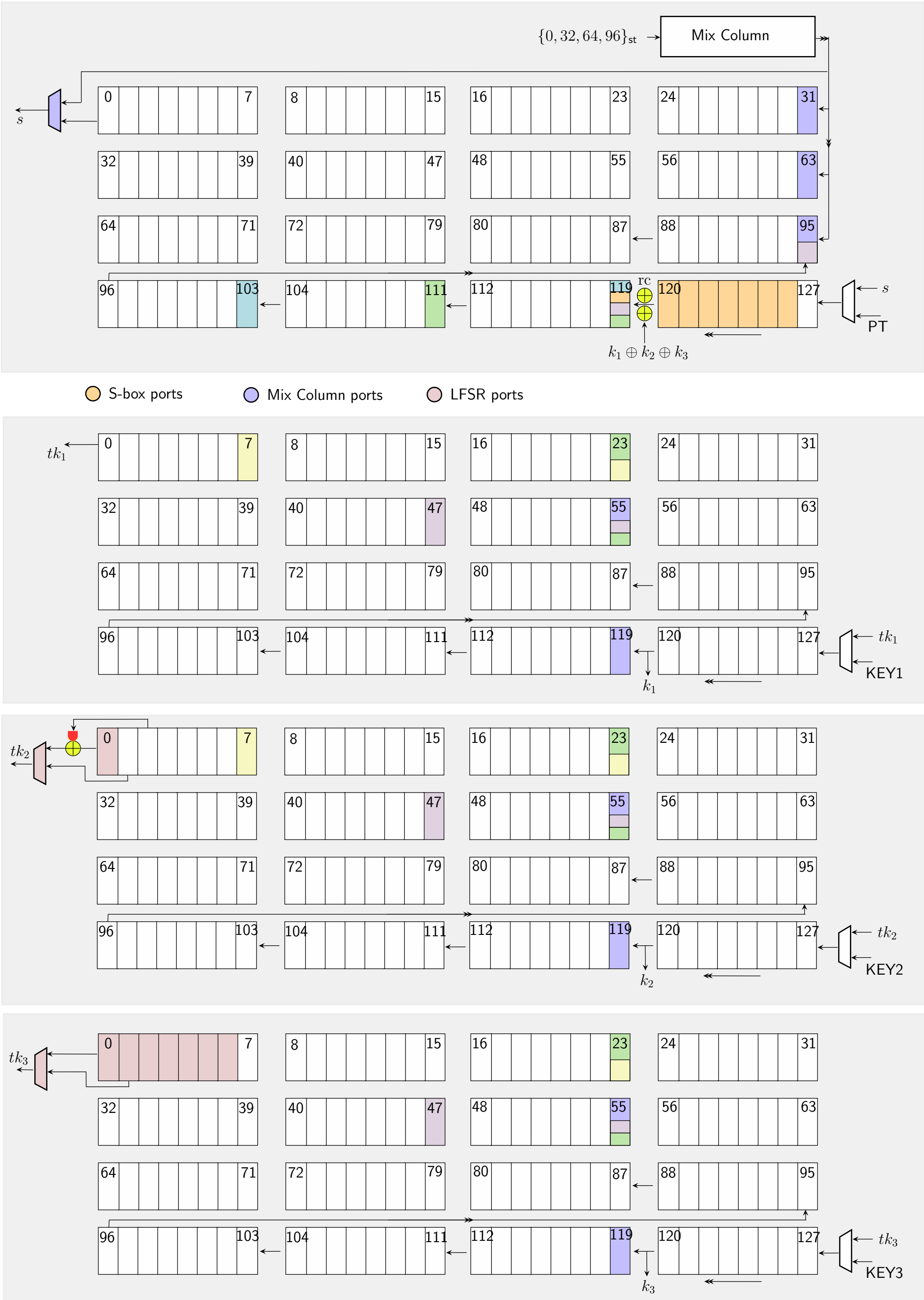

Figure 3: The state (above) and key ($TK1$, $TK2$ and $TK3$ respectively) pipelines of SKINNY encryption.

Library	Area (μm^2)	Area (GE)	Power (μW) @ 10 MHz	Latency round	Latency total	Energy (nJ)	Throughput Mbps	Ref.
			SKINNY-128-128					
STM 90	4697.7	1070	51.47	128	5248	27.0	13.43	Sec. 4
UMC 90	4249.3	1355	52.08	128	5248	27.3	11.51	Sec. 4
TSMC 90	4022.6	1425	47.12	128	5248	24.7	15.42	Sec. 4
Nangate 15	391.7	1992	15.89	128	5248	8.3	277.51	Sec. 4
Nangate 45	1394.9	1748	122.06	128	5248	64.1	41.63	Sec. 4
			SKINNY-128-256					
STM 90	6642.7	1513	75.30	128	6272	47.2	10.72	Sec. 4
UMC 90	6043.9	1927	75.70	128	6272	47.5	9.46	Sec. 4
TSMC 90	5730.9	2030	69.25	128	6272	43.4	12.90	Sec. 4
Nangate 15	561.0	2853	22.99	128	6272	14.4	219.98	Sec. 4
Nangate 45	1996.9	2502	175.28	128	6272	109.9	34.26	Sec. 4
			SKINNY-128-384					
STM 90	8631.5	1966	99.36	128	7296	72.5	6.37	Sec. 4
UMC 90	7895.7	2518	99.73	128	7296	72.8	7.68	Sec. 4
TSMC 90	7465.2	2645	91.38	128	7296	66.7	11.44	Sec. 4
Nangate 15	733.7	3732	30.22	128	7296	22.0	163.32	Sec. 4
Nangate 45	2603.6	3263	229.10	128	7296	167.2	29.94	Sec. 4

Table 6: Encryption only circuits for SKINNY.

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

Table 7: Bit-sliced GIFT* permutation where index 0 is the rightmost bit of a row segment.

5 GIFT*

We will be focusing our efforts on the bitsliced design of the GIFT blockcipher, as utilized in the NIST LWC candidates GIFT-COFB and SUNDAE-GIFT [BCI+19, BBP+19]. We denote it by GIFT* as it differs from the original construction in the way data bits are organized. The specification is laid out in the appendix of [BPP+17]. In this variant, the cipher state is reordered and interpreted as a two-dimensional array, i.e. four 32-bit segments S_0, S_1, S_2, S_3 such that

$$\begin{bmatrix} S_0 \\ S_1 \\ S_2 \\ S_3 \end{bmatrix} = \begin{bmatrix} s_3 & s_7 & \dots & s_{127} \\ s_2 & s_6 & \dots & s_{126} \\ s_1 & s_5 & \dots & s_{125} \\ s_0 & s_4 & \dots & s_{124} \end{bmatrix},$$

where $s_0 s_1 \dots s_{127}$ are the state bits. In this ordering s_3 is the most significant bit in the state pipeline and s_{124} the least significant. The 4-bit S-box is applied column-wise and the permutation layer consists of four independent row-wise permutations as shown in Table 7.

In contrast the key state is not reordered and remains unchanged with respect to the regular GIFT^* specification, i.e.

$$\begin{bmatrix} K_0 \mid\mid K_1 \\ K_2 \mid\mid K_3 \\ K_4 \mid\mid K_5 \\ K_6 \mid\mid K_7 \end{bmatrix} = \begin{bmatrix} k_0 & k_1 & \dots & k_{31} \\ k_{32} & k_{33} & \dots & k_{63} \\ k_{64} & k_{65} & \dots & k_{95} \\ k_{96} & k_{97} & \dots & k_{127} \end{bmatrix},$$

where $k_0k_1\dots k_{127}$ are the key bits. The key schedule consists of two 16-bit internal rotations and an external rotation that spans the entire state. More specifically, K_6 is rotated 2 positions to right, K_7 is rotated 12 position to the right and the entire key state is rotated concurrently by 32 positions to the right.

$$\left[K_0, K_1, K_2, K_3, K_4, K_5, K_6, K_7\right] = \left[K_6 \gg 2, K_7 \gg 12, K_0, K_1, K_2, K_3, K_4, K_5\right].$$

In each round 64-bits of the key are mixed into the cipher state as follows

$$S_2 = S_2 \oplus U$$
$$S_1 = S_1 \oplus V,$$

where $U = K_2 \mid\mid K_3$ and $V = K_6 \mid\mid K_7$. Alongside the round keys, in each round there is also a 6-bit $(c_5, c_4, c_3, c_2, c_1, c_0)$ round constant that is added to the state and updated such that

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

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

5.1 State Pipeline

The bitwise nature of both the GIFT^* and GIFT permutation complicates matters in a swap-and-rotate setting, since each state bit needs to be moved to its designated position individually. As a consequence, a simple solution with few swaps as devised for the AES ShiftRows procedure, detailed in Section 3.2 is not achievable.

Nevertheless, the GIFT^* permutation can be partitioned into three layers each can be generated with three separate swaps, thus, in total, we allocate nine swaps.

Layer 1. Consists of the swaps $(\mathsf{FF}_{31}, \mathsf{FF}_{30})$, $(\mathsf{FF}_{31}, \mathsf{FF}_{28})$ and $(\mathsf{FF}_{31}, \mathsf{FF}_{29})$.

Layer 2. Consists of the swaps $(\mathsf{FF}_{28}, \mathsf{FF}_{24})$, $(\mathsf{FF}_{28}, \mathsf{FF}_{26})$ and $(\mathsf{FF}_{28}, \mathsf{FF}_{26})$.

Layer 3. Consists of the swaps $(\mathsf{FF}_{22}, \mathsf{FF}_{4})$, $(\mathsf{FF}_{22}, \mathsf{FF}_{10})$ and $(\mathsf{FF}_{22}, \mathsf{FF}_{16})$.

Due to the column-wise application of the substitution layer in GIFT^*, the s-box ports in the state pipeline are FF_{31}, FF_{63}, FF_{95} and FF_{127} which are active during the cycles 96 to 127.

A graphical depiction of the GIFT^* state pipeline is given in Figure 4.

5.2 Key Pipeline

The bitsliced interpretation of GIFT^* significantly simplifies how the 64-bit round keys are extracted in each round since they are now mixed into a continuous stretch of the cipher state. For this we can assume, without loss of generality, that the master key K is loaded in the following order as to simplify the swapping algorithm.

$$K = \begin{bmatrix} K_0 \mid\mid K_1 \\ K_6 \mid\mid K_7 \\ K_2 \mid\mid K_3 \\ K_4 \mid\mid K_5 \end{bmatrix},$$

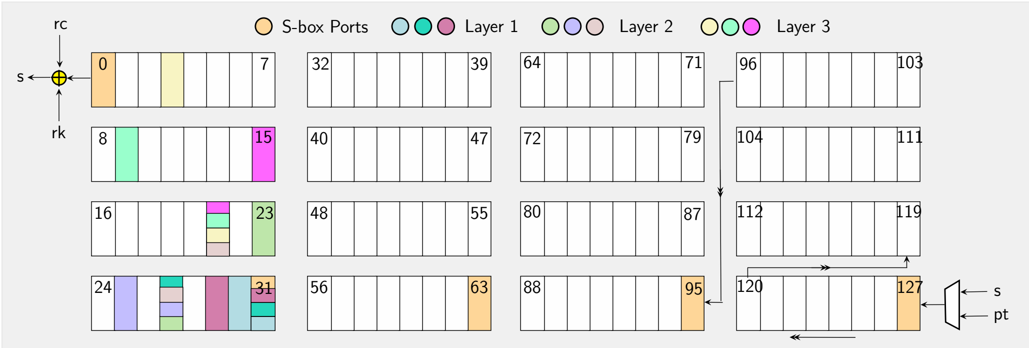

Figure 4: 128-cycle, bit-serial GIFT* key round function implementation using nine swaps.

In this scenario the 64-bit round keys $K_2 \mid\mid K_3 \mid\mid K_6 \mid\mid K_7$ is added to the blockcipher states during the cycles 32 to 96.

The swap sequence for the GIFT* key schedule is partitioned into four phases.

Phase 1 (Rotating the state). We rotate the entire key state by 64 positions to the left. This operation can be achieved with a single swap during 64 active cycles. Preferably, the transformation should occur concurrently with the addition of the round key into the cipher state, i.e. we allocate FF_0 and FF_{64} to perform the rotation during the cycles 32 to 96.

Phase 2 (Swapping the precedence). To achieve a full emulation of the 96-bit rightward rotation of the key schedule it further necessary to swap the precedence of the utilized round key halves, i.e. $K_2||K_3$ and $K_6 \; ||K_7$. This again only requires a single swap during 32 cycles and can be performed subsequently to the first phase, hence we allocate FF_0 and FF_{96} for this second phase.

Phase 3 (Rotating K_6). This transformation can been seen as a 14-bit leftward rotation that can be achieved by composing three leftward rotations of magnitude 8, 4, and 2. The position and the interval of those three swaps can be chosen relatively freely, as $K_6||K_7$ is not part of the current round key, as long as they occur after the second phase has terminated. To simplify the matter we chose to perform them back-to-back during the cycles 32 and 66. More concretely, the 4-bit rotation is done during the cycles 32 to 44 using a swap at register FF_{95} and FF_{99}. Subsequently, we perform the 8-bit rotation during cycles 44 to 52 with the registers FF_{83} and FF_{91}, followed by the 2-bit rotation during cycles 52 to 66 using the registers FF_{75} and FF_{77}.

Phase 4 (Rotating K_7). Phase 3 is followed by a 4-bit leftward rotation of K_7 that is congruent to the 12-bit rightward rotation of the specification. This necessitates a single swap of size 4 for which we can reuse the same swap as utilized in phase 3, i.e. FF_{99} and FF_{95} during the cycles 48 to 60.

A summary of both the key schedule and round function swaps is tabulated in Table 8.

5.3 GIFT

The regular GIFT specification is significantly harder to transform into a low-latency swap-and-rotate circuit due to the fact that the round key bits are not added to cipher state in a continuous stretch. Namely, if $U = K_5||K_4$ and $V = K_1||K_0$ represent the 64-bit round key than its individual bits are mixed into the state S as follows,

$$s_{4i+2} = s_{4i+2} \oplus u_i, \;\; s_{4i+1} = s_{4i+1} \oplus v_i, \;\; \forall i \in \{0, \dots, 31\}.$$

pipeline	operation	active cycles
state	swap (31, 30)	$\{8k+7 : k \in [0,15]\}$
	swap (31, 28)	$\{8k+5 : k \in [0,15]\} \cup \{8k+7 : k \in [0,15]\}$
	swap (31, 29)	$\{8k+5 : k \in [0,15]\}$
	swap (28, 24)	$\{0,1,42,43,50,51,58,59,66,67,104,105,112,113,120,121\}$
	swap (28, 26)	$\{6,7,10,11,14,15,18,19,22,23,26,27,30,31\}$ $\cup\{34,35,72,73,80,81,88,89,96,97\}$
	swap (28, 22)	$\{74,75,82,83,90,91,98,99\}$
	swap (22, 4)	$\{2,3,34,35,66,67,98,99\}$
	swap (22, 10)	$\{4,5,26,27,36,37,58,59,68,69,90,91,100,101,122,123\}$
	swap (22, 16)	$\{6,7,18,19,28,29,38,39,50,51,60,61,70,71\}$ $\cup\{82,83,92,93,102,103,114,115,124,125\}$
	key addition	$[32,96)$
	rc addition	(look up table)
	load S-box	$[96,128)$
key	swap (64, 128)	$[32,96)$
	swap (32, 128)	$[96,128]$
	swap (96, 100)	$[32,44)$
	swap (84, 92)	$[44,52)$
	swap (76, 78)	$[52,66)$
	swap (96, 100)	$[48,60)$

Table 8: The timetable of operations for bit-serial GIFT* encryption.

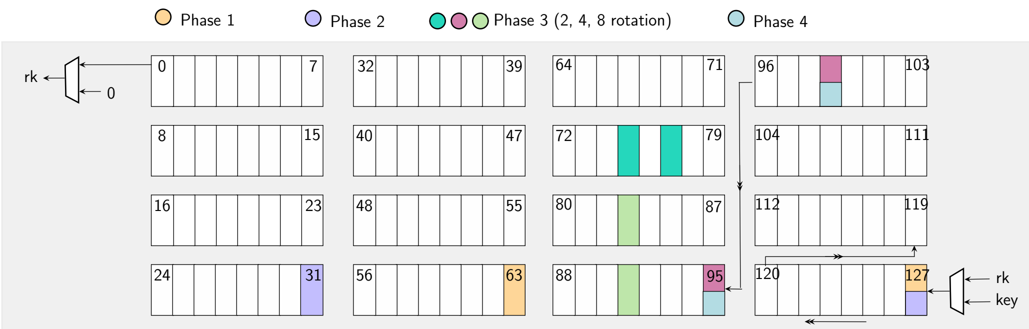

Figure 5: 128-cycle, bit-serial GIFT* key schedule implementation using five swaps.

Library	Area (μm^2)	Area (GE)	Power (μW) @ 10 MHz	Latency round	Latency total	Energy (nJ)	Throughput Mbps	Ref.
				GIFT*				
STM 90 nm	4863.5	1108	48.7	128	5248	25.5	9.76	Sec. 5
UMC 90 nm	4410.8	1332	49.8	128	5248	26.1	10.49	Sec. 5
TSMC 90 nm	4176.5	1480	45.1	128	5248	23.7	13.43	Sec. 5
Nangate 15 nm	402.3	2047	15.4	128	5248	8.1	192.12	Sec. 5
Nangate 45 nm	1432.1	1791	122.3	128	5248	64.2	32.02	Sec. 5
				GIFT				
STM 90 nm	5334.3	1215	51.3	128	5248	26.9	7.35	Sec. 5
UMC 90 nm	4801.2	1531	51.8	128	5248	27.2	6.32	Sec. 5
TSMC 90 nm	4507.3	1597	45.9	128	5248	24.1	8.61	Sec. 5
Nangate 15 nm	430.5	2190	16.1	128	5248	8.4	146.92	Sec. 5
Nangate 45 nm	1528.4	1915	131.8	128	5248	69.2	23.34	Sec. 5

Table 9: GIFT* bit-serial, low-latency synthesis figures.

pipeline	operation	active cycles
state	swap (39, 71)	$[0,8) \cup [8,16) \cup [16,24) \cup [121,128)$
	swap (38, 22)	$[0,9] \cup [58,73] \cup [122,127]$
	swap (98, 110)	$[10,13] \cup [34,37] \cup [54,57] \cup [74,77] \cup [98,101] \cup [118,121]$
	swap (109, 85)	$[7,10] \cup [51,54] \cup [71,74] \cup [115,118]$
	swap (108, 72)	$[4,7] \cup [68,71]$ $\cup \{34,35,72,73,80,81,88,89,96,97\}$
	swap (121, 117)	$\{16k+5 : k \in [0,15]\} \cup \{16k+13 : k \in [0,15]\} \cup \{16k+15 : k \in [0,15]\}$
	swap (122, 114)	$\{16k+1 : k \in [0,15]\} \cup \{16k+3 : k \in [0,15]\}$
	swap (123, 111)	$\{16k+1 : k \in [0,15]\}$
	swap (22, 4)	$\{2,3,34,35,66,67,98,99\}$
	swap (22, 10)	$\{4,5,26,27,36,37,58,59,68,69,90,91,100,101,122,123\}$
	swap (22, 16)	$\{6,7,18,19,28,29,38,39,50,51,60,61,70,71\}$ $\cup \{82,83,92,93,102,103,114,115,124,125\}$
	key addition	$\{4k+1 : k \in [0,31]\} \cup \{4k+2 : k \in [0,31]\}$
	rc addition	(look up table)
	load S-box	$\{4k+3 : k \in [0,31]\}$
key	swap (120, 128)	$\{4k : k \in [3,16]\}$ if $\mathsf{round} \bmod 4 = 0$ $\{4k-1 : k \in [3,16]\}$ if $\mathsf{round} \bmod 4 = 1$ $\{4k+1 : k \in [3,16]\}$ if $\mathsf{round} \bmod 4 = 2$ $\{4k-2 : k \in [3,16]\}$ if $\mathsf{round} \bmod 4 = 3$
	swap (112, 128)	$\{1\} \cup \{4k-2 : k \in [5,16] \cup [21,32]\}$ if $\mathsf{round} \bmod 4 = 0$ $\{4k : k \in [5,16] \cup [21,31]\}$ if $\mathsf{round} \bmod 4 = 1$ $\{1\} \cup \{4k-1 : k \in [5,16] \cup [21,32]\}$ if $\mathsf{round} \bmod 4 = 2$ $\{4k+1 : k \in [5,16] \cup [21,31]\}$ if $\mathsf{round} \bmod 4 = 3$
	swap (1, 128)	$\{4k : k \in [1,31]\}$ if $\mathsf{round} \bmod 4 = 0$ $\{0\}$ if $\mathsf{round} \bmod 4 = 1$ $\{4k+2 : k \in [0,31]\}$ if $\mathsf{round} \bmod 4 = 3$
	swap (1, 33)	$\{4k+1 : k \in [0,7]\}$ if $\mathsf{round} \bmod 4 = 0$ $\{4k+3 : k \in [0,7]\}$ if $\mathsf{round} \bmod 4 = 1$ $\{4k+2 : k \in [0,7]\}$ if $\mathsf{round} \bmod 4 = 2$ $\{4k : k \in [1,8]\}$ if $\mathsf{round} \bmod 4 = 3$
	swap (4, 5)	$\{4k : k \in [0,31]\}$ if $\mathsf{round} \bmod 4 = 0$
	swap (2, 3)	$\{4k : k \in [0,31]\}$ if $\mathsf{round} \bmod 4 = 3$

Table 10: The timetable of operations for bit-serial GIFT encryption.

By reordering the key bits in this manner such that the bits of U and V exit the pipeline during the correct cycles we can reuse the rotation techniques to obtain a key schedule with 6 different swaps.

However, we can transfer the intuition for the state pipeline of Section 5.1 in order to generate the swap sequence for the GIFT round function. The summary of all GIFT key schedule and round function swaps is tabulated in Table 10.

6 AEAD

As standalone blockciphers are not ready-to-use primitives they are usually wrapped in mode of operation. In this section, we investigate three NIST LWC candidates which are bootstrapped with the improved bit-slided implementations of AES, SKINNY and GIFT* presented in the previous sections: SUNDAE-GIFT, Romulus and COMET. For all three schemes we report the hitherto smallest blockcipher-based authenticated encryption circuits in the literature.

The choice of these three particular candidates in our work is influenced by the observation that the area of a blockcipher is determined, to a large extend, by the amount of storage elements, rather than how lightweight the round operations are. This is more evident when one compares SKINNY-128-256, whose round function comprises lightweight operations, to AES, whose S-box and Mix columns circuits are significantly large. Nevertheless, AES beats SKINNY-128-256 in terms of area, because the latter requires 256-bits of storage for the tweakey, whereas the former only needs 128-bit for the key.

Because an authenticated encryption scheme produces a tag besides ciphertext blocks, it is natural to expect a particular value that is initialized at the beginning and updated repetitively after processing each new block of data. We refer to this value as *the running state*. The running state is eventually used to compute the tag, so that all blocks contribute to its value. From the area perspective, an important question is whether storing the running state requires an extra register or not. For the chosen candidates, the running state is not actually a separate value, but it is passed between consecutive encryption calls. In other words, we can use the state register inside the blockcipher to keep this value temporarily until the next encryption starts. It is precisely the reduction in the storage area that yields the impressive area results for the three candidates.

In the special case of Romulus, which actually defines six different variants, we chose the one that is likely to use the smallest area in ASIC circuit. This is Romulus-N3, because it uses the smaller SKINNY-128-256, while its nonce-based siblings all use SKINNY-128-384.

Another important detail about our AEAD implementations, which directly concerns the hardware API, is that we assume the padding is done a priori to AEAD call. In other words, our implementations leave padding task to the caller, and assume that the associated data and message bits are well aligned with block boundaries. This is in contrast to CAESAR Hardware API, which assumes the padding as the responsibility of the circuit [HDF+16]. Hence, our reported area figures should be carefully interpreted, if one happens to compare them with other implementations which contain the padding circuit.

AEAD mode of operations generally treat the last, empty or partial blocks specially through some allocated bits in the domain separator. Hence, when assuming that the associated data and message are properly chopped into blocks and passed to the circuit, information lost during the padding must also be passed along. In our lightweight API, we use few input signals to indicate if the current data block must be specially processed, e.g. whether the current data block is the last block of associated data, or a padded block.

6.1 SUNDAE-GIFT

The SUNDAE-GIFT AEAD scheme was proposed by Banik et al. and is based on the SUNDAE mode of operation and features the GIFT* blockcipher at its core [BBP+19, BBLT18]. It is a bare-bones construction that does not require any additional registers aside the ones used within the blockcipher. After the encryption of the nonce each data block is mixed into the AEAD state between the encryption calls. A field multiplication over $GF(2^{128})$ is applied after the last associated data has been added to the state. Analogously, the same multiplication is performed for the last message block. The multiplication is either $\times 2$ when the last AD or message block have been padded or $\times 4$ whenever the last blocks are complete. More formally, the multiplication $\times 2$ is encoded as a byte-wise shift and the addition of the most significant byte into other bytes of the state such that if $B_0||B_1||\ldots||B_{15}$ represent the 16 bytes of the intermediate AEAD state with B_0 being the most significant byte we have that

$$2 \times (B_0||B_1||\ldots||B_{15}) = B_1||B_2||\ldots B_{10}||B_{11} \oplus B_0||B_{12}||B_{13} \oplus B_0||B_{14}||B_{15} \oplus B_0||B_0,$$

and $4 \times (B_0||B_1||\ldots||B_{15}) = 2 \times (2 \times (B_0||B_1||\ldots||B_{15}))$. The tag is produced after processing all AD and message blocks and the ciphertext blocks are generated by reprocessing the message blocks afterwards. A schematic of the SUNDAE-GIFT is depicted in Figure 6.

The simplicity of SUNDAE-GIFT can be exploited in a bit-serial implementation to attain a circuit with very low overhead in terms of area. In fact, aside a more involved control logic the sole addition to the GIFT* circuit presented in Section 5 is the field multiplication.

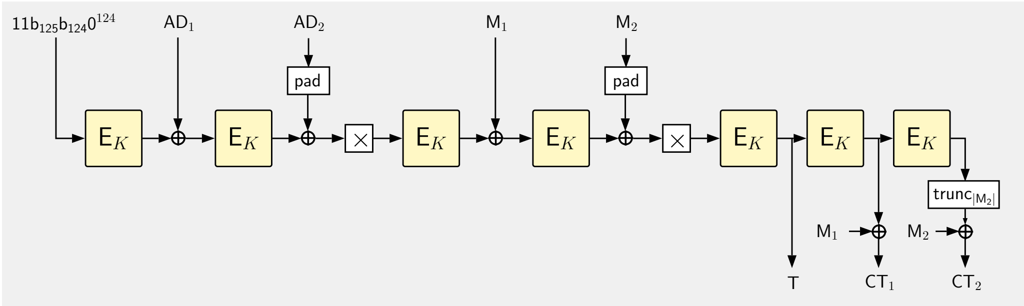

Figure 6: SUNDAE-GIFT high-level overview. The figure depicts the processing of two message and two associated data blocks.

The multiplier can be achieved with a single swap and a one additional 1-bit multiplexer and XOR gate. More concretely, we allocate 128 rounds for the multiplication $\times 2$ and twice the amount for the multiplication $\times 4$ during which the blockcipher round function and key swaps are disabled. In other words while the ciphertext bits exit the last round function computation we swap FF_{120} and FF_0 during the cycles 8 to 127 which rotates the state by 8 positions to the left. Hence in the worst case we require $4 \times 128 = 512$ additional cycles for multiplications. The addition of B_0 into the other state bytes occurs during the cycles 88 to 96, 104 to 112 and 120 to 128. In terms of latency, each new encryption call is loaded with the new plaintext while the ciphertext bits of the previous computation exit the pipeline. As a consequence the very first encryption operates over $41 \times 128 = 5248$ cycles while the remaining encryption each take $40 \times 128 = 5120$ cycles.

After synthesis the resulting SUNDAE-GIFT architecture is the to-date smallest authenticated encryption circuit at around 1200 gate equivalents for the STM 90 nm process which is only a 8 percent increased compared to the bit-serial GIFT* implementation presented in Section 5. The exact synthesis figures for various cell libraries are tabulated in Table 11.

6.2 SAEAES

The SAEAES AEAD scheme was proposed by Naito et al. [NMMaS+19] and uses the AES blockcipher as the underlying encryption core. The SAEAES document proposes a number of parameters according to which the mode can be operated, but the primary candidate of these is `SAEAES128_64_128` which implies keysize of 128 bits, message/AD blocks of 64 bits and tag size of 128 bits. This effectively makes the primary mode of rate 1/2, since 2 blockcipher calls are required per 128 bits of message/AD. However the mode requires no additional state other than those required in the calculation of the blockcipher encryption and so a very compact implementation is possible.

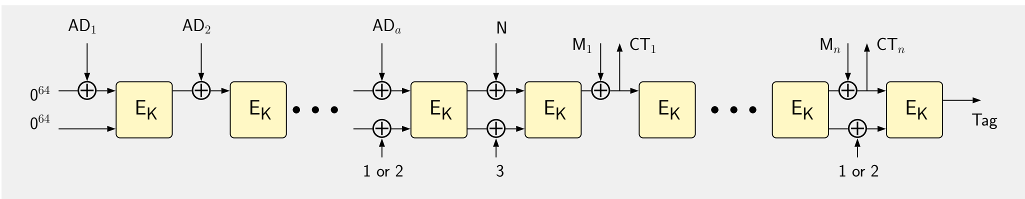

Figure 7: SAEAES high-level overview.

A high level description of the mode of operation is presented in Figure 7. It is easy to

Library	Area (μm^2)	Area (GE)	Power(μW) (@ 10 MHz)	Latency (cycles)	Energy (nJ)	Throughput (Mbit/s)
			SUNDAE-GIFT			
STM 90 nm	5273.9	1201	50.1	92800	464.9	4.80
UMC 90 nm	4729.9	1508	51.1	92800	474.2	5.00
TSMC 90 nm	4444.6	1663	45.9	92800	426.0	5.75
Nangate 15 nm	426.6	2170	15.9	92800	147.6	90.80
Nangate 45 nm	1527.9	1915	130.3	92800	1209.2	15.13
			SAEAES			
STM 90 nm	5938.0	1350	77.2	24448	188.7	6.58
UMC 90 nm	5381.4	1716	66.9	24448	163.6	10.22
TSMC 90 nm	4942.7	1751	56.9	24448	139.1	7.63
Nangate 15 nm	464.3	2362	18.8	24448	46.0	152.69
Nangate 45 nm	1653.5	2067	148.8	24448	363.8	22.76
			Romulus			
STM 90 nm	7812.7	1779	79.1	61447	486.1	5.72
UMC 90 nm	7155.6	2282	81.6	61447	501.3	6.10
TSMC 90 nm	6658.8	2359	74.0	61447	454.4	8.99
Nangate 15 nm	650.8	3310	25.0	61447	153.7	145.14
Nangate 45 nm	2304.1	2887	199.0	61447	1222.8	19.16

Table 11: Low-latency synthesis figures for selected AEAD Schemes. Energy and throughput calculated for processing 1 KB of plaintext and 128 bits of AD

see that the mode does not require additional storage other than the ones required in the blockcipher. From a circuit designer's point of view, it is not very difficult to implement the mode as the only real challenge is to ensure that at the beginning of a particular encryption operation the circuit feeds the correct input vectors to the blockcipher circuit which are as follows

A: $Inp_i = AD_i||0^{64} \oplus E_K(Inp_{i-1})$ or AD_1 (if $i = 1$) during the associated data processing stage, where (Inp_i is the i-th input to the blockcipher).

B: $Inp_a = AD_a||const_{64} \oplus E_K(Inp_{a-1})$ for the last AD block.

C: $IV = N \oplus 3_{128} \oplus E_K(Inp_a)$ before the processing of the plaintext begins.

D: $Inp'_i = M_i \oplus E_K(Inp'_{i-1})$ during the plaintext processing stage, where (Inp'_i is the i-th input to the blockcipher during plaintext processing). It can also be seen that Inp'_i is also incidentally the i-th ciphertext block, and the Tag is simply the outcome of the final encryption call that the mode performs.

A bitwise AES encryption core produces output one bit per clock cycle during the last 128 cycles of the 1408 clock encryption cycle. Since we are using no additional storage blocks, the output bits once produced need to be XORed with the appropriate input signal and concurrently fed back to the blockcipher as inputs to the encryption call. Essentially cycles 1281 to 1408 not only produce the output of the i-th encryption but also serve as the input period for the $(i + 1)$-th encryption. Thus one needs to exercise some more fine grained control over the circuit, to ensure that during cycles 1281 to 1408 the blockcipher circuit is able to perform the dual role. This means that effectively all encryption calls except the first requires 1280 cycles. Hence to process a AD and n plaintext blocks the circuit requires $a + n + 1$ encryption calls and hence $1408 + 1280 * (a + n)$ cycles.

Table 11 tabulates the synthesis figures for the mode with different standard cell libraries. The smallest implementation takes around 1350 GE in the STM 90 library which is only around 83 GE (6.5%) more than the standalone blockcipher circuit.

6.3 Romulus

Romulus is an AEAD scheme designed by Iwata et al. [IKMP19], and uses the SKINNY family of blockciphers. In this work, we choose the member Romulus-N3, because it belongs to the same nonce-based family with the primary member Romulus-N1, but it uses smaller SKINNY-128-256 instead of SKINNY-128-384. Given the measurements in Table 6, our estimate is that the primary member would take an additional 450 GE of area compared to Romulus-N3, which would roughly correspond to 2200-2300 GE.

In order to reduce the number of blockcipher calls, and make use of the large tweakey space, Romulus-N3 makes 1/2 blockcipher call per associated data block, and 1 blockcipher call per message block. Romulus-N3 member admits 128-bit key, 96-bit nonce, variable-length message chopped into 128-bit blocks, and produces 128-bit tag. An interesting design choice here is that associated data has alternating blocksize. In particular for some integer i, AD_{2i-1} blocks are 128-bit, and AD_{2i} blocks are 96-bit. Alternatively, to ease notation and the description, one can actually treat $\mathsf{AD}_{2i-1}||\mathsf{AD}_{2i}$ as a single 224-bit block, assuming that the original padding is preserved during this conversion.

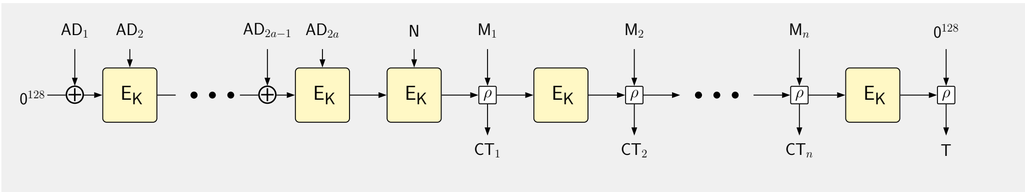

Figure 8: Processing associated data AD_i and message M_i blocks in Romulus-N3.

Figure 8 describes the three phases a full AEAD operation passes through, namely processing of (1) associated data, (2) nonce and (3) messages.

During associated data phase, each combined 224-bit $\mathsf{AD}_{2i-1}||\mathsf{AD}_{2i}$ block is processed with a single blockcipher call $\mathsf{E_K}$. For each of these SKINNY-128-256 calls, the plaintext is AD_{2i-1}, and the tweakey is concatenation of 24-bit counter[5], 8-bit domain separator, 96-bit AD_{2i} block and the 128-bit key K. The output from the blockcipher is treated as the running state, and XORed with each new AD_{2i-1} blocks.

Once, all $\mathsf{AD}_{2i-1}||\mathsf{AD}_{2i}$ combined blocks are processed, the running state is encrypted by using the nonce N itself as part of the tweakey. We refer to this as processing of nonce.

During the message phase, for each of the 128-bit message blocks, the running state and message block M_i are passed through ρ function defined below. Essentially ρ acts as XOR in the lateral direction, hence the running state is XORed with the message blocks as before. Once all message blocks are processed, the final blockcipher output is passed through ρ with 0^{128} to produce the tag. $\rho(S, M) = (S', C)$ is defined as $S' \leftarrow S \oplus M$ and $C \leftarrow G(S) \oplus M$. For each byte, G performs the following operation:

$$G(x_7||x_6||x_5||x_4||x_3||x_2||x_1||x_0) := (x_0 \oplus x_7)||x_7||x_6||x_5||x_4||x_3||x_2||x_1$$

It is clear then how we can use 1-bit serial SKINNY-128-256 to realize Romulus-N3. Except for the computation of the ciphertext blocks through ρ, we can simply reuse the state pipeline of SKINNY-128-256 to store the running state. In order to compute G, we

[5] 24-bit counter is defined with regards to a LFSR (see [IKMP19]), and counts the number of blockcipher calls during a phase.

use two external 7-bit buffer pipelines, which keeps the copy of the last 7 bits that exits the state pipeline and the last 7-bit of message blocks fed to the circuit. This leads to 7 clock cycle of delay in between the time a message block is fed and the time the ciphertext bits become available. This similarly applies to the tag as well, hence a last 7 clock cycle delay must is also considered during latency calculation.

As a concrete example, the circuit would process 2×224 bits of associated data and 1×128 bits of message as follows:

- During the first 128 cycles, the key K, associated data blocks AD_1 are loaded simultaneously. Starting from clock cycle 32, 96-bit AD_2 is also being loaded[6]. After loading is complete, the circuit becomes busy for 47 more rounds (for SKINNY-128-256 encryption), i.e. this takes 47×128 clock cycles. At the last clock cycle, the circuit signals that it is ready for receiving the next data block, be it AD_3 or M_1.

- For the following 128 cycles, the state pipeline XORs its content with AD_3, and at the same time initiates the first round of encryption simultaneously. Again, the key is reloaded starting from cycle 0 and AD_4 is also loaded Starting from clock cycle 32. The circuit becomes busy for 47 rounds to compute the encryption. At the last cycle, the circuit signals that the key and the nonce must be reloaded the following round.

- The running state is encrypted, i.e. the state pipeline reloads its own content and start encryption. No data block needs to be loaded, but the key and the nonce must be loaded simultaneously. Since nonce and 96-bit AD blocks are using the exact same positions in the tweakey, the nonce is loaded starting from clock cycles 32. After 47 rounds, the circuit signals that the next data (i.e. message) block can be loaded.

- The message block is loaded, which happens simultaneously with reloading the key. The nonce also follows the key with 32 clock cycles delay, as before. The ciphertext bits become available with 7 clock cycles of delay. The circuit again takes 47 rounds to perform the final encryption.

- A final ρ operation is performed with the running state and the 0^{128} vector. The tag becomes available with 7 clock cyles.

7 Conclusion

Bit serial implementation of blockcipher and authenticated encryption schemes provide the smallest known implementations, because they reduce the number of gates on the datapath, such as MUXes, XOR gates etc. In this paper we implemented compact bit serial implementations of 3 blockciphers with around 20 % lesser latency when compared to the state of the art, with the added advantage that we do not alter the standard arrangement of bits as recommended in their specifications. As a result they are readily usable in any mode of operation that uses them as an underlying encryption primitive. As a proof of concept, we implement 3 lightweight AEAD schemes from NIST LWC project that also turn out to be the most compact implementation of these schemes reported in literature.

References

[ALP+19] Elena Andreeva, Virginie Lallemand, Antoon Purnal, Reza Reyhanitabar, Arnab Roy, and Damian Vizár. Forkae. *NIST Lightweight Cryptography Project*, 2019.

[6] According to Romulus-N3 specification, the last 96-bit of $TK1$ should receive nonce/associated data blocks. The leftmost 32-bits of $TK1$ are reserved for the counter and the domain separator

[BBLT18] Subhadeep Banik, Andrey Bogdanov, Atul Luykx, and Elmar Tischhauser. SUNDAE: small universal deterministic authenticated encryption for the internet of things. *IACR Trans. Symmetric Cryptol.*, 2018(3):1–35, 2018.

[BBP+19] Subhadeep Banik, Andrey Bogdanov, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, Elmar Tischhauser, and Yosuke Todo. Sundae-gift v1.0. *NIST Lightweight Cryptography Project*, 2019.

[BBR16] Subhadeep Banik, Andrey Bogdanov, and Francesco Regazzoni. 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*, pages 173–190, 2016.

[BBR17] Subhadeep Banik, Andrey Bogdanov, and Francesco Regazzoni. Compact circuits for combined AES encryption/decryption. *Journal of Cryptographic Engineering*, pages 1–15, 2017.

[BBRV19] Subhadeep Banik, Fatih Balli, Francesco Regazzoni, and Serge Vaudenay. Swap and rotate: Lightweight linear layers for spn-based blockciphers. Cryptology ePrint Archive, Report 2019/1212, 2019. https://eprint.iacr.org/2019/1212.

[BCI+19] Subhadeep Banik, Avik Chakraborti, Tetsu Iwata, Kazuhiko Minematsu, Mridul Nandi, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. Gift-cofb v1.0. *NIST Lightweight Cryptography Project*, 2019.

[BJK+16] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The SKINNY Family of Block Ciphers and Its Low-Latency Variant MANTIS. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II*, pages 123–153, 2016.

[BJK+19] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. Skinny-aead. *NIST Lightweight Cryptography Project*, 2019.

[BPP+17] 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.

[CDJ+19a] Avik Chakraborti, Nilanjan Datta, Ashwin Jha, Cuauhtemoc Mancillas Lopez, Mridul Nandi, and Yu Sasaki. Estate. *NIST Lightweight Cryptography Project*, 2019.

[CDJ+19b] Avik Chakraborti, Nilanjan Datta, Ashwin Jha, Cuauhtemoc Mancillas Lopez, Mridul Nandi, and Yu Sasaki. Lotus-aead and locus-aead. *NIST Lightweight Cryptography Project*, 2019.

[CDJN19] Avik Chakraborti, Nilanjan Datta, Ashwin Jha, and Mridul Nandi. Hyena. *NIST Lightweight Cryptography Project*, 2019.

[CDL+19] Anne Canteaut, Sébastien Duval, Gaëtan Leurent, María Naya-Plasencia, Léo Perrin, Thomas Pornin, and André Schrottenloher. Saturnin. *NIST Lightweight Cryptography Project*, 2019.

[CN19] Bishwajit Chakraborty and Mridul Nandi. mixfeed. *NIST Lightweight Cryptography Project*, 2019.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer Verlag, Berlin, Heidelberg, New York, 2002.

[FIP] PUB FIPS. 197, advanced encryption standard (aes), national institute of standards and technology, 2001. *Link in: http://csrc. nist. gov/publications/fips/fips197/fips-197. pdf.*

[GJK+19] Dahmun Goudarzi, Jérémy Jean, Stefan Kölbl, Thomas Peyrin, Matthieu Rivain, Yu Sasaki, and Siang Meng Sim. Pyjamask. *NIST Lightweight Cryptography Project*, 2019.

[GJN19] Shay Gueron, Ashwin Jha, and Mridul Nandi. Comet. *NIST Lightweight Cryptography Project*, 2019.

[HDF+16] Ekawat Homsirikamol, William Diehl, Ahmed Ferozpuri, Farnoud Farahmand, Panasayya Yalla, Jens-Peter Kaps, and Kris Gaj. Caesar hardware api. Cryptology ePrint Archive, Report 2016/626, 2016. https://eprint.iacr.org/2016/626.

[IKMP19] Tetsu Iwata, Mustafa Khairallah, Kazuhiko Minematsu, and Thomas Peyrin. Romulus v1.2. *NIST Lightweight Cryptography Project*, 2019.

[JMPS17] Jérémy Jean, Amir Moradi, Thomas Peyrin, and Pascal Sasdrich. 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*, pages 687–707, 2017.

[ME19] Alexander Maximov and Patrik Ekdahl. New circuit minimization techniques for smaller and faster AES sboxes. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(4):91–125, 2019.

[MPL+11] Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. Pushing the limits: A very compact and a threshold implementation of aes. In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, pages 69–88, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[NMMaS+19] Yusuke Naito, Yasuyuki Sakai Mitsuru Matsui and, Daisuke Suzuki, Kazuo Sakiyama, and Takeshi Sugawara. Saeaes. *NIST Lightweight Cryptography Project*, 2019.