# Hardware Implementations of NIST Lightweight Cryptographic Candidates: A First Look

Behnaz Rezvani, Flora Coleman, Sachin Sachin, and William Diehl

Virginia Tech, Blacksburg, VA 24061, USA
email: {behnaz, googly2, sachin255701, wdiehl}@vt.edu

**Abstract.** Achieving security in the Internet of Things (IoT) is challenging. The need for lightweight yet robust cryptographic solutions suitable for the IoT calls for improved design and implementation of constructs such as authenticated encryption with associated data (AEAD) which can ensure confidentiality, integrity, and authenticity of data in one algorithm. The U.S. National Institute of Standards and Technology (NIST) has embarked on a multi-year effort called the lightweight cryptography (LWC) standardization process to evaluate lightweight AEAD and optional hash algorithms for inclusion in U.S. federal standards. As candidates are evaluated for many characteristics including hardware resources and performance, obtaining results of hardware implementations as early as possible is preferable. In this work, we implement six NIST LWC Round 2 candidate ciphers, SpoC, GIFT-COFB, COMET-AES, COMET-CHAM, Ascon, and Schwaemm and Esch, in the Artix-7, Spartan-6, and Cyclone-V FPGAs. Implementations are compliant with the newly-released hardware (HW) applications programming interface (API) for lightweight cryptography and are tested in actual hardware. We also provide the average power and energy per bit of our implementations at 40 MHz. Results indicate that SpoC has the smallest area and power, while Ascon has the highest throughput-to-area (TPA) ratio.

**Keywords:** NIST · Lightweight cryptography · FPGA · Implementation · Authenticated encryption

## 1 Introduction

Increasing the use of very lightweight devices that form the basis of the Internet of Things (IoT) necessitates the development and adaptation of lightweight cryptographic algorithms. IoT devices, like all other means of information technology, are vulnerable to theft of privacy information, and are subject to potentially more destructive attacks such as replay or man-in-the-middle attacks. To protect IoT devices against the range of such attacks, cryptographic solutions should ensure *confidentiality* (i.e., where an adversary cannot read private communications), *integrity* (i.e., where any change in transmitted data can be detected), and *authenticity* (i.e., where a receiver can verify the identity of the sender). Authenticated encryption with associated data (AEAD) can ensure all of the above services

using a single algorithm, while realizing savings in cost and performance, and by avoiding security pitfalls of interactions with separately-designed ciphers and hashes.

In August 2018, the U.S. National Institute of Standards and Technology (NIST) issued a call for specifications of lightweight AEAD and optional hashes, to be subjected to several rounds of evaluations as part of the lightweight cryptography (LWC) standardization process, and eventually incorporated into U.S. federal information processing standards (FIPS) [35]. Submissions of specifications were permitted until February 2019, and 56 qualified Round 1 candidates were publicized in April 2019. In August 2019, 32 candidates were selected for Round 2, which is expected to last a year.

NIST LWC candidates are evaluated based on several criteria, including cost (e.g., area, memory, energy consumption) and performance (e.g., latency, throughput (TP), power consumption) in resource-constrained environments representative of emerging IoT devices. All submissions were required to include software reference implementations. While several submissions included synthesis or implementation results from the authors' own hardware submissions in ASIC or FPGA, the NIST LWC evaluation process specifically assigns higher weights to the 3rd-party implementations, i.e., those that have been provided by parties other than the ciphers' authors.

In this paper, we provide the first medium-scale comparion of hardware implementations of selected NIST LWC candidate submissions. We select six ciphers from the Round 2 selections for evaluation: SpoC [3], GIFT-COFB [8], COMET-AES [27], COMET-CHAM [27], Ascon [20], and Schwaemm and Esch [10] (Schwaemm is the name for the AEAD process and Esch is the name for the hash function). The rationale for selection of these ciphers can be summarized as follows:

1. At least 29 NIST LWC Round 2 submissions are composed of block cipher or block cipher-like primitives, out of which at least 15 are sponge-based. SpoC, Ascon, and Schwaemm and Esch are sponge-based, while GIFT-COFB, COMET-AES, and COMET-CHAM are based on block ciphers.
2. Considering all 32 candidates and their members, 13 ciphers use 4-bit S-boxes, 7 ciphers use 8-bit S-boxes, 3 ciphers use 5-bit S-boxes, 3 ciphers use addition, rotation, and XOR (ARX) operations, and at least 9 ciphers use a logical AND or multiplication for non-linear transformations. GIFT-COFB uses 4-bit S-boxes, COMET-AES uses 8-bit S-boxes, Ascon uses 5-bit S-boxes, COMET-CHAM and Schwaemm and Esch use ARX, and SpoC uses a Simeck box for non-linear transformations.
3. AES and GIFT are frequently encountered, as 5 candidates use GIFT and 4 candidates use AES as their underlying ciphers.
4. COMET is a block cipher that used Beetle mode of operation but replaces the sponge permutation with a block cipher. The mode of operation of COMET is a mixture of Beetle and counter modes.
5. There are 3 ARX algorithms in Round 2: COMET-SPECK, COMET-CHAM, and Schwaemm and Esch. We choose CHAM instead of SPECK since CHAM

is a new lightweight block cipher that shows slightly better performance than SPECK in software, and better results than SIMON in hardware [30]. SCHWAEMM and ESCH is the only sponge-based ARX design in Round 2.
6. Two of the selected ciphers have hashes: ASCON and SCHWAEMM and ESCH and we provide a subcomparison between them. We choose ASCON since it is the winner of the CAESAR lightweight use-case, while ESCH is the only hash function that uses ARX operations as its non-linear function.
7. No weaknesses or errors in specification have been publicly identified in the selected ciphers.

Except for COMET-CHAM, we implement the authors' primary recommendations for selected ciphers and use register transfer level (RTL) methodology in Verilog or VHDL with basic-iterative (i.e., round-based) architecture. Since each round of the ciphers is executed in one clock cycle, this implementation offers a reasonable latency and throughput with a medium area. Although latency and energy consumption are also important considerations in the LWC [5], there are certain trade-offs between them. As discussed in [5, 11], basic-iterative implementation is an efficient architecture and a good candidate for applications with low energy consumption.

All implementations are fully compliant with the newly-released LWC hardware applications programming interface (LWC HW API) [29], and use the LWC hardware development package (LWC HW DP) at [41]. Implementations are functionally verified in Xilinx Vivado simulator, and results are generated for Xilinx Artix-7 and Spartan-6, and Intel Cyclone-V FPGAs. Our choice of FPGAs is motivated by popular target platforms, specifically, the NewAE CW305 Artix-7 target board, the Digilent Nexys 3 (Spartan-6), and the Terasic DE1-SoC (Cyclone-V). Post place-and-route results for the Artix-7 are further optimized using the Minerva automatic hardware optimization tool [22].

For candidates including both AEAD and hash, we follow recommendations in [29] to produce two implementations: AEAD alone (AEAD), and AEAD and hash (AEAD + Hash) incorporated together in one core. Implementations are compared according to maximum frequency, area, TP, and throughput-to-area (TPA) ratio. Furthermore, the average power and energy per bit ($E$/bit) of ciphers, measured on the Artix-7 target board at 40 MHz, are also provided.

Our contributions in this work are summarized as follows:

1. We provide the first medium-scale comparison of hardware implementations of selected and representative Round 2 NIST LWC candidates.
2. We offer the first intensive validation of the LWC HW API and DP on a wide range of candidates, including AEAD and hash.
3. We compare our implementations with selected past and present authenticated cipher implementations in order to bridge the gap between CAESAR and NIST LWC processes, as well as LWC HW API-compliant and LWC HW API-non-compliant implementations.
4. We show examples of features of cipher implementations which are resource-intensive and reduce performance, in order to illustrate design pitfalls and help guide later-round tweaks.

The paper is organized as follows: We first provide background on previous hardware implementations, authenticated ciphers, the LWC HW API and associated developer's package in Section 2. We then describe implementations of our chosen cipher candidates in Section 3, and present implementation results in Section 4. A fair comparison with previous results along with the important takeaways are provided in Section 5. We finally conclude the paper in Section 6.

## 2 Background

### 2.1 Hardware implementations in cryptographic competitions

Public competitions for cipher standards began with the NIST call for AES submissions in 1997 [33]. Large-scale comparisons of hardware performance did not emerge until near the end of the selection process, e.g., [24, 21]. This means that hardware implementations were emphasized only in the last third of the competition. As stated in [34], there is typically less data on hardware performance available to evaluators, even going into final deliberations, since the amount of time required to explore each possible implementation is much greater.

Based on lessons-learned from the AES and subsequent SHA-3 competitions, members of the cryptographic community sought to accelerate and standardize fair and efficient benchmarking of hardware implementations of candidates submitted to CAESAR [15]. A standard API for hardware implementations was validated by the CAESAR committee, and CAESAR HW API was made available. Hardware implementations were required from all submission teams beginning with Round 3, out of ultimately 4 rounds of selection [28].

However, the time-frame necessary to understand the design space of hardware implementations was still relatively compressed. For example, Round 1 submissions were due in March 2014, announcement of Round 2 candidates occurred in July 2015, and most hardware submissions appeared only slightly before the announcement of Round 3 candidates in August 2016. As CAESAR finally concluded in February 2019, nearly half of the competition had elapsed before the accumulation of a critical mass of data on hardware implementations.

With each new cryptographic competition and standardization effort, there is increasing interest on achieving early hardware benchmarks. However, the number of submissions has increased, while early round evaluation periods are short. For example, the NIST LWC standardization process was announced in August 2018, with submission deadlines (including software reference implementations) in February 2019. The public announcement of Round 1 submissions was posted in April 2019, and the Round 2 selections were announced on August 30, 2019.

The NIST call for submissions (like previous competitions) mandated inclusion of software reference implementations, but left open the option to report hardware implementations. Many NIST LWC submissions included author-reported implementations in either ASIC or FPGAs, e.g., ESTATE, SAEAES, Oribatida, etc. [36]. However, [35] specifically highlights the value of 3rd-party evaluations of candidate submissions. This is the first comparison of hardware submissions of its kind following the publication of official specifications in April 2019.

### 2.2 Authenticated encryption with associated data

In both CAESAR and the NIST LWC standardization process, two operations are defined on AEAD, authenticated encryption and authenticated decryption. In encryption, inputs consist of a public message number *Npub* usually defined as a "number used once" (nonce), a secret key $K$, plaintext $PT$, and associated data $AD$ (the secret message number *Nsec* is defined but not expected in NIST LWC submissions). The outputs of authenticated encryption include *Npub*, $AD$, ciphertext $CT$, and *Tag*, which provides for integrity and authenticity of all transmitted data. In authenticated decryption, the inputs are *Npub*, $AD$, $K$, $CT$, and *Tag*. $CT$ is internally decrypted into $PT$, however, an internal $Tag'$ is computed and checked against *Tag* prior to releasing $PT$, in a step called "tag verification."

Properly-defined and engineered authenticated encryption schemes are a way to ensure provision of cryptographic services while avoiding pitfalls of attempting to combine separate ciphers and hashes (e.g., [31]). However, they are much more complex than individual block ciphers or secure hashes, and warrant the focus of the cryptographic community on analysis of their strengths and weaknesses, including physical hardware implementations.

### 2.3 Hardware applications programming interface

As in the case of CAESAR, no analogous hardware API was issued in the call for NIST LWC specifications. In October 2019, the first version of NIST LWC API was proposed and updated later in November. Described at [29], the LWC HW API outlines interface standards and minimum compliance criteria to ensure implementations compatibility of the same algorithms by different designers, fairness, and ease of benchmarking and evaluation.

For example, external interfaces are aligned with the popular AMBA advanced extensible interface 4 (AXI4) standard [6], and consists of three ports: public data (`pdi`), secret data (`sdi`), and data output (`do`). Most data arrives and departs on the public data interface, except for secret keys, which arrive on the secret data interface.

Additionally, a protocol consisting of commands, headers, and data, as well as a prescribed sequence of operations, is used to input and process types of data required for authenticated encryption and decryption, including *Npub*, $AD$, $PT$, $CT$, and *Tag* and for hashing, including message $M$ and message digest or hash $H$.

### 2.4 Hardware developer's package

To facilitate the hardware designer's task of meeting LWC HW API requirements, a hardware developer's package is provided at [41]. The package includes an input processor (PreProcessor) and an output processor (PostProcessor), which are encapsulated in a top-level module called LWC. A designer can place a custom design in a subordinate module called CryptoCore, and use standardized

interfaces to communicate with PreProcessor and PostProcessor. A Python script called `cryptotvgen` is used to generate representative test vectors directly from the software reference implementation, and an accompanying HDL test bench (LWC_TB) automatically verifies test vectors against expected results. An implementer's guide to assist in using the LWC HW DP is also available at [39].

In this work, we use the LWC HW DP (v1.0.1), and develop RTL implementations inside CryptoCore. A representation of CryptoCore, instantiated in LWC with accompanying I/O modules, is shown in Fig. 1. External signals defined in the LWC HW API are the AXI-4 compatible signals associated with `pdi` (public data), `sdi` (secret data), and `do` (data output). Internal input signals to CryptoCore, defined only in the developer's package, include `bdi` (block data input), `key` (key input), `bdi_size`, and `bdi_valid_bytes`. Output signals to the PostProcessor, defined only in the developer's package, include `bdo` (block data output), in which *CT* and *Tag* depart CryptoCore, and `msg_auth` (message authentication), which is used to signal results of tag verification in authenticated decryption. Other signals are defined in [39].

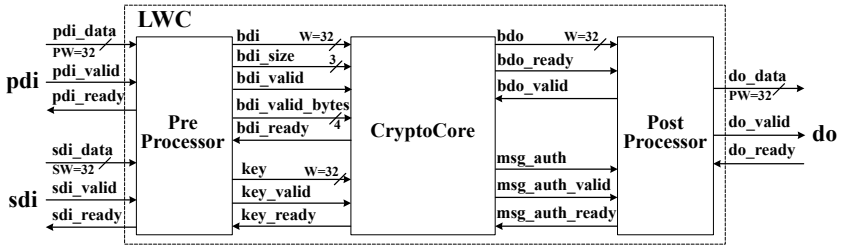


Fig. 1: Instantiation of CryptoCore inside LWC, together with modules and signals from LWC HW DP.

In the LWC HW API and the LWC HW DP, decrypted $PT$ is released during authenticated decryption, regardless of whether or not tag verification succeeds. While this represents a potential analytic vulnerability, it is necessitated by the impracticality of internally buffering potentially long messages and concurrent evaluation by automated test benches.

## 3 Ciphers implemented in this paper

### 3.1 SpoC

**Description.** SpoC, described in [3], refers to "sponge with a masked capacity." It is a sponge-based cipher based on the Beetle mode, where ciphertext is not directed into the permutation, and where combined feedback in the first $r$-bits of rate increases protections against forgery with smaller state size, thus leading to more efficient implementations [18, 12, 13]. In SpoC, capacity is masked with data

blocks instead of rate which improves the security and allows larger rate value per permutation call. We implement one of the authors' primary recommendations, SpoC-64, with capacity $c = 128$ bits, state size $b = 192$ bits, nonce size $|N| = 128$ bits, and tag size $\tau = 64$ bits. In a sponge-based cipher, the rate refers to the number of keystream bits generated per permutation call, and the capacity $c = b - r$.

This cipher is based around the sLiSCP-light[192] permutation; the ACE (with 320 bits of state) and SPIX (with 256 bits of state) NIST LWC candidates also use the sLiSCP permutation [2, 4]. The sLiSCP-light uses a combination of a Type II Generalized Feistel Structure (GFS) and Simeck box (SB), and consists of 18 steps of 6 rounds each. Each step consists of three transformations, namely, SubstituteSubblocks (SSb), AddStepconstants (ASc), and MixSubblocks (MSb). The non-linear operations are applied in the SSb, or SB. SBs consist of XORs, bitwise rotations, and a 48-bit logical and.

In order to follow the basic-iterative construction, we construct two SBs (SB1 and SB3) on 48-bits each, which operate on a total of 96 bits out of 192 bits of state. Round constants are supplied to SB1 and SB3 (corresponding to 48-bit state words $S_1$ and $S_3$, respectively) at the start of each SSb transformation. An SSb transformation requires 6 rounds, each of which executes in one clock cycle. Local state variables, as well as updated round constants, are stored during SSb transformations. The two round constants ($rc_0$ and $rc_1$) and two step constants ($sc_0$ and $sc_1$), each 6 bits, are implemented using look-up tables. The other two state words, $S_0$ and $S_2$, are XORed with step constants in the ASc transformation. Finally, $S_0$ is mixed with $S_1$, and $S_2$ with $S_3$, using XOR in the MSb transformation. One sLiSCP permutation is executed in $18 \times 6 = 108$ clock cycles. The sLiSCP-light[192] permutation is shown in Fig. 2.

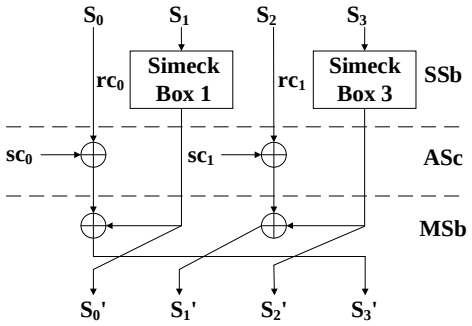


Fig. 2: sLiSCP permutation. Bus widths are 48 bits.

The duplex sponge construction of SpoC is shown in Fig. 3. At each point in time, the state can be divided into a $c$-bit $Y$ and $r$-bit $Z$, and represented as $Y \parallel Z$. The initial state $Y_0 \parallel Z_0$ is formed by interleaving Nonce and Key ($f(N_0, K)$), and performing a permutation. The *Tag* is generated using an interleaved set of

bytes extracted from across the entire 192-bit state. Control bits *ctrl* are 4-bit constants used for domain separation (i.e., to distinguish between authenticated encryption or decryption phases), such as *AD*, *PT*, and *Tag*, and to differentiate between full and partial blocks.

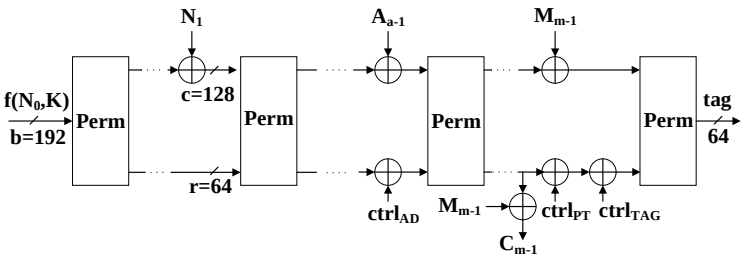


Fig. 3: SpoC duplex sponge construction.

**Implementation.** We implement a basic-iterative architecture based on the sLiSCP permutation, where one round of the SSb transformation executes in a single clock cycle. This requires 108 clock cycles for the permutation. The SpoC authors, who describe an ASIC implementation in [3], likewise use a basic-iterative architecture requiring 108 clock cycles per permutation.

The SpoC algorithm requires one-zero padding ($10^*$ padding) for *AD*, *PT*, and *CT*. $10^*$ padding is not provided as a service in the LWC HW DP, so it is implemented in our CryptoCore based on the number of valid input bytes provided by `bdi_valid_bytes`. Additionally, the *CT* size must be equal to the *PT* size during output, so we are required to mask non-output bytes. A mask consisting of $FF^*$ is left-shifted once per clock cycle to truncate output to the desired length. Mask adjustment is overlapped with permutation, so there is no effect on latency or throughput. Our implementation of SpoC-64 is shown in Fig. 4.

### 3.2 GIFT-COFB

**Description.** GIFT-COFB is based on the combined feedback (COFB) mode of operation with GIFT-128 as the underlying block cipher, which is described in [8]. COFB mode is single-pass (one block cipher call per data block) and inverse-free (no need for block cipher decryption). The GIFT-COFB recommendations are data block size $n = 128$ bits, nonce size $n = 128$ bits, and tag size $\tau = 128$ bits.

GIFT-128 is a substitution-permutation network (SPN) with a 128-bit key length and a 128-bit cipher state length. Several NIST LWC candidates such as SUNDAE-GIFT [7] also use GIFT-128 as their block cipher. This iterative block cipher has 40 rounds and each round consists of 3 transformations, namely, SubCells, PermBits, and AddRoundKey. The cipher state divides into four 32-bit

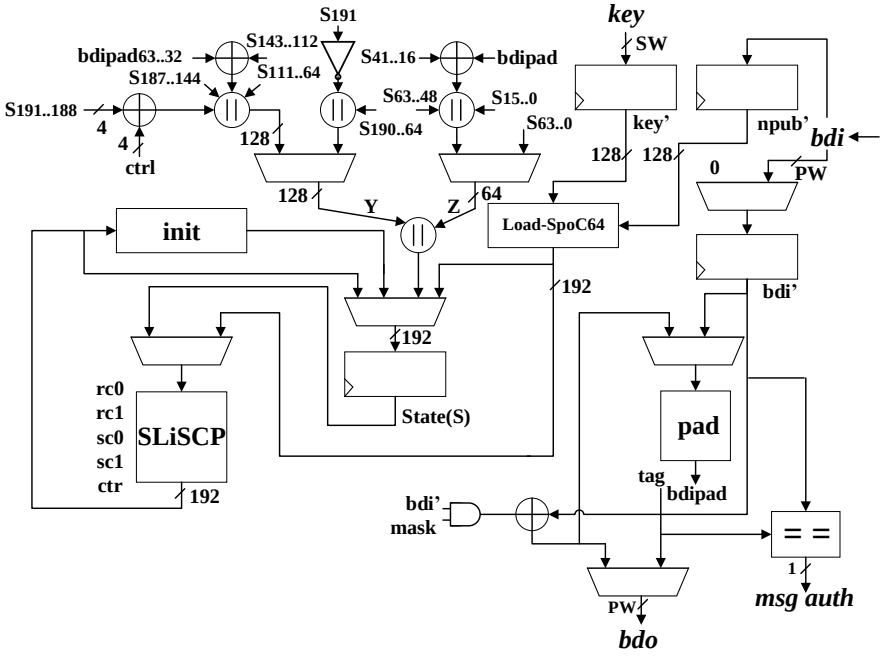


Fig. 4: Block diagram for SpoC-64.

words and the key state divides into eight 16-bit segments. In SubCells, 32 4-bit bitslice S-boxes are applied to every nibble of the state. Then, a 32-bit permutation is applied to every word of the state, which in an iterative implementation, is only bit wiring. In AddRoundKey, the round key is XORed to the second and third words of the state, and a round constant is added to the last word of the state. The addition of the round key is done over only half of the state and the key scheduling is merely a bit permutation. The round constants are generated by a 6-bit LFSR. Based on the above features and as mentioned in [9], GIFT-128 has a low footprint which makes it a good choice for lightweight applications [8].

In Fig. 5, a simplified version of the encryption construction of GIFT-COFB is depicted. At the beginning of the encryption, the state is loaded by a nonce $N$, and then the upper 64 bits of the first $E_K$ output are considered as the delta state that is denoted by $L$ in Fig. 5. Except for the last block of $AD$ and $PT$, the delta state is multiplied by 2 in GF($2^{64}$) for every block of $AD$ and $PT$. For the last block of $AD$ or $PT$, the delta state is multiplied by $3^i$ or $3^{j-i}$, where $i, j - i \leq 4$. The $G$ function is defined as $G(Y) = (Y[2], Y[1] <<< 1)$ [8], which $Y[1]$ and $Y[2]$ are the upper and lower 64-bit of the $Y$ signal, respectively.

**Implementation.** A basic-iterative architecture is used here, i.e., every round of the GIFT round function is executed in one clock cycle. The GIFT-COFB authors also used a round-based design which is implemented in ASIC. GIFT has

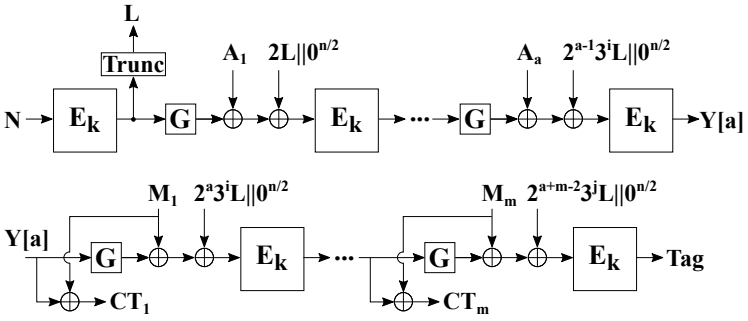


Fig. 5: GIFT-COFB encryption construction [8].

40 rounds, thus it requires 40 clock cycles to process a block of the input data. However, for processing $AD$ and $PT$, we need additional clock cycles due to the delta state requirements. As presented in [8], 4 clock cycles are required for the delta state. In this work, we use 4 clock cycles for processing an $AD$ block, but 2 clock cycles in our finite state machine for processing the plaintext blocks. The reason that we reduce the clock cycles for $PT$ is that the exponent in $3^{j-i}$ does not exceed 2 for an $PT$ block. As a result, we have 40, 44, and 42 clock cycles for processing nonce, an $AD$ block, and a block of message, respectively. Note that these are the number of clock cycles that GIFT needs to process one block of data.

Similar to SpoC, the $10^*$ padding is applied to the partial last block of $AD$ and $PT$. During the decryption, we need additional padding for the plaintext before applying it to the feedback. We use the `bdi_size` to accomplish the padding function. Since modules in the LWC HW DP use 32 bits of `bdi` and `bdo`, we use a counter to track the number of valid bytes that the last block of $PT$ contains. The truncating module is only used for the last 32 bits of a 128-bit $CT$, and masks the $CT$ with the required amount of zeros by utilizing this counter. Our GIFT-COFB implementation is shown in Fig. 6.

### 3.3 COMET

**Description.** COunter Mode Encryption with authentication Tag (COMET) is a block cipher that can be viewed as a mixture of CTR and Beetle modes of operations [27]. Similar to GIFT-COFB, COMET is inverse-free and single-pass. The authors presented three different block ciphers as the COMET underlying cipher, of which we considered AES-128/128 and CHAM-128/128. The primary member of COMET is COMET-128_AES-128/128. The COMET recommendations are key size $k = 128$ bits, data block size $n = 128$ bits, nonce size $n = 128$ bits, and tag size $\tau = 128$ bits.

The COMET encryption process is shown in Fig. 7. The state consists of an $n$-bit state $Y$ and a $k$-bit state $Z$ that can be concatenated to form a $b$-bit state $(Y||Z)$, similar to a sponge-based permutation. The $\varphi$ block in Fig. 7 refers to a

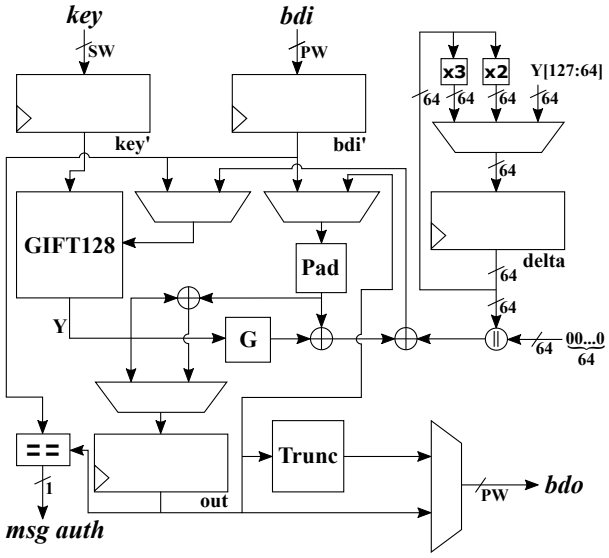


Fig. 6: Block diagram of GIFT-COFB implementation. Bus widths are 128 bits unless indicated.

permutation over $\{0,1\}^k$ and defined as $\varphi(Z) = (Z_1, 2 \times Z_0)$, where $Z_1$ and $Z_0$ are upper and lower 64 bits of the $Z$-state, respectively, and the multiplication is defined over GF($2^{64}$). Inside the $\varrho$ block, during the $AD$ process, the $Y$-state is XORed with the $AD$, but in the encryption process, the $Y$-state words are shuffled first and then XORed with the $PT$ to produce the $CT$. The encryption process starts by using the nonce $N$ and secret key $K$ to create the initial state $(Y_0, Z_0)$, where $Y_0 = K$ and $Z_0 = E_k(N)$. In Fig. 7, the control bit $\text{Ctrl}_{\text{ad}}$ indicates the start of the non-empty $AD$ and in case of the partial last block of $AD$, the control bit $\text{Ctrl}_{\text{p-ad}}$ is used. The $PT$ process is similar to the $AD$ process, but using different domain separation signals. Moreover, we extract out a block of $CT$ for each block of message. At the end of the encryption process, the control signal $\text{Ctrl}_{\text{tg}}$ indicates the tag generation call.

**AES Description.** AES-128/128 is an SPN with 128-bit key and 128-bit cipher state [38]. The algorithm is made up of 10 rounds and each round consists of 4 transformations, namely, SubBytes, ShiftRows, MixColumns, and AddRoundKey. Note that the final round skips the MixColumns step. The initial state is divided into a $4 \times 4$ matrix with 8-bit elements. In SubBytes, the 8-bit S-boxes are applied over each byte of the state matrix. During the ShiftRows, the $i$-th row is rotated $i$ bytes to the left. In the MixColumns step, the state is multiplied by an invertible MDS matrix. Finally, the 128-bit round key is XORed with the state. In this paper, we use the AES code (v1.3) available at [25].

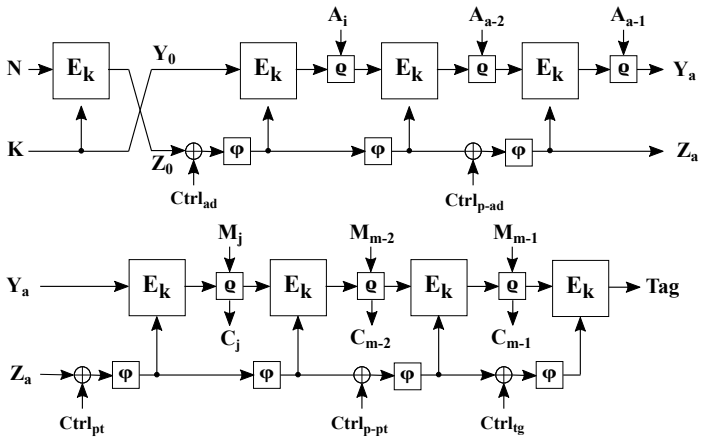


Fig. 7: COMET encryption construction [27].

**CHAM Description.** CHAM is a 4-branch generalized Feistel structure with modulo addition, rotation, and XOR (ARX) operations [30]. The key state does not need an update since CHAM exploits the stateless-on-the-fly key schedule, which makes it a lightweight cipher. In this paper, we considered CHAM-128/128 with 128-bit key, 128-bit block size, and 80 rounds. The key schedule and two consecutive round functions of CHAM are depicted in Fig. 8 and Fig. 9, respectively. The secret key $K$ is divided into four 32-bit words ($w = 32$) and the eight round keys are generated as in Fig. 8. The initial state is also divided into 32-bit words and updated as in Fig. 9. Note that the number of rotations to the left for both key scheduling and round function are different for even and odd rounds.

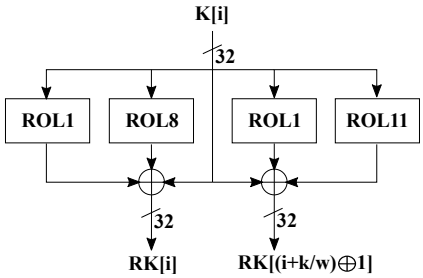


Fig. 8: Key schedule of CHAM [30].

**Implementation.** The COMET authors used little-endian format of indexing and assumed all binary strings are byte-oriented [27]. As we mentioned before, our implementations are based on basic-iterative architecture. Therefore, each

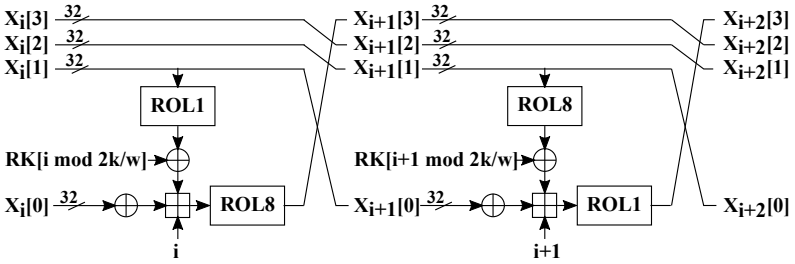


Fig. 9: Two consecutive round functions of CHAM [30].

round of AES and CHAM are executed in one clock cycle and each of them require 10 and 80 clock cycles, respectively, to complete the process of one input block. We use the same datapath and controller for both COMET-AES and COMET-CHAM, where the implementation is shown in Fig. 10. COMET has two states, $Y$ and $Z$ that are updated for each block of input. Similar to SpoC and GIFT-COFB, the $10^*$ padding is applied to partial last block of $AD$ and $PT$. Moreover, the truncation (chop module in Fig. 10) is also applied on the partial last block of $CT$ to have the same size as its $PT$. Both the padding and truncation modules exploit the `bdi_size` signal as the selector of their multiplexers.

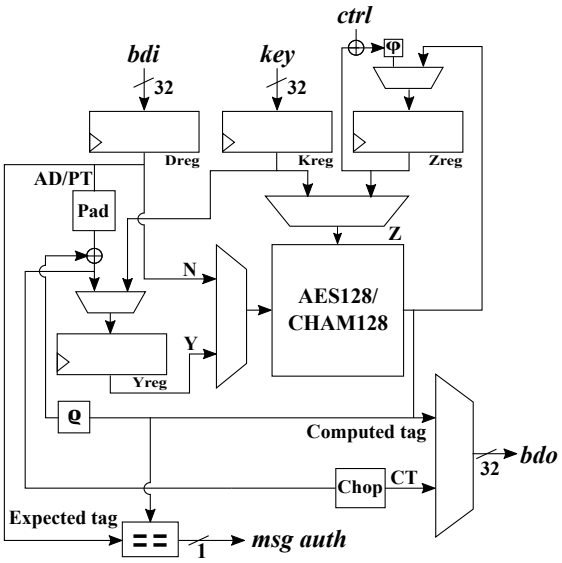


Fig. 10: Block diagram of COMET implementation. Bus widths are 128 bits unless indicated.

### 3.4 Ascon

**AEAD Description.** Ascon, chosen as the primary recommendation for lightweight authenticated encryption in the final portfolio of the CAESAR competition, is a permutation-based cipher with sponge and duplex modes of operation [20]. The authors primary member is Ascon-128 with key size $k = 128$ bits, nonce size $|N| = 128$ bits, tag size $\tau = 128$ bits, state size $b = 320$ bits, rate size $r = 64$ bits, and capacity size $c = b - r = 256$ bits.

Fig. 11 shows the encryption structure of Ascon with the duplex mode of operation. The permutation $p$ is made up of 12 or 6 rounds, where each round consists of three steps, namely: Addition of constants, substitution layer, and linear diffusion layer. The 320-bit state is divided into five 64-bit words. In addition of constants, the third word is XORed with the appropriate round constant. Then, 5-bit S-boxes are applied over the state in the substitution layer step. Finally, each word of the state in the linear diffusion layer is differently permuted with XOR and rotation functions. The round number is 12 for initialization and finalization and is 6 for processing $AD$ and $PT$.

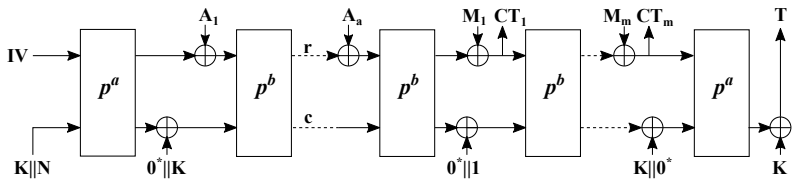


Fig. 11: Ascon encryption construction [20].

The encryption starts with the initialization step. The initial state is formed by the initialization vector (IV) and concatenation of the secret key $K$ and the nonce $N$, followed by a permutation over the state. At the end of the initialization, $K$ is XORed with the lower 128-bit of capacity. In the finalization, $K$ is first XORed with the upper 128 bits of capacity and then a permutation is executed over the state. After the permutation, the tag is extracted from XORing the lower 128 bits of capacity with $K$.

**Hash Function Description.** Fig. 12 shows the construction of the Ascon hash algorithm. The sizes of rate, capacity, and state are the same as in the Ascon AEAD, however, the IV value is different. The message digest or hash $H$ size is $h = 256$ bits. After the initialization, each block of the message $M$ is absorbed and a permutation is applied over the state. After finishing the message absorption, each block of message digest is extracted from the first 64 bits of the state and after each block extraction, a permutation is applied until the whole 256 bits of hash is obtained. Note that the number of permutation rounds during the hashing is always 12.

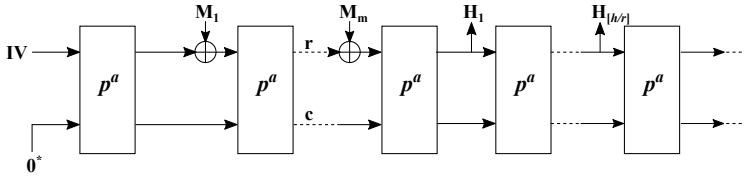


Fig. 12: Ascon hash function construction [20].

**Implementation.** Like the other ciphers, Ascon is implemented based on the basic-iterative architecture. Thus, the initialization and finalization are executed in 12 clock cycles and the processing of $AD$ and $PT$ are executed in 6 clock cycles. We use the same controller and datapath to support both AEAD and hash function, in which the implementation block diagram is shown in Fig. 13. In Ascon, a single one and multiple zeros are added at the end of the $AD$ and message strings before parsing them into $r$-bit blocks. That is, even if their lengths are multiples of $r = 64$ bits, we still have to add a $10^*$ block as the last block of $AD$ or $M$. Moreover, even if the message is empty, we should still add the $10^*$ block. In Ascon, there is a permutation after the last block of $AD$ or $M$ in hash, however, there is no permutation for the last block of message in AEAD (it can be a padded partial block or just a $10^*$ block). The datapath handles these situations by using multiplexers to select the appropriate values in each condition. The truncation is similar to the other ciphers, i.e., we use zero masking for invalid bytes of $CT$. The padding and truncating functions are applied during loading data, thus they do not affect the latency or throughput.

### 3.5 Schwaemm and Esch

**Description** Schwaemm256-128 and Esch256 are the primary recommended AEAD and hash candidates suggested by Beierle et al. [10]. Schwaemm256-128 processes input data in 256-bit blocks using a key size of 128 bits, a nonce size of 256 bits, and it provides a 128-bit output tag. Esch256 processes input data in 128-bit blocks and provides a 256-bit hash output.

The permutation used by both of the candidates is Sparkle384 which is SPN-based with an internal state size of 384 bits. The internal state is processed as 6 branches with each branch consisting of 2 consecutive words. The permutations run for 7 (STEPS_SLIM) or 11 (STEPS_BIG) steps depending on which block of the input is being processed. Each step of the permutation uses four rounds of an ARX box called Alzette and then applies a linear layer. Each of the six branches of the state are processed by Alzette using a specific round constant. The rotation amounts used in Alzette vary based on the current round. The use of this ARX box effectively achieves the substitution needed for the network. After the completion of four rounds of Alzette, the linear layer is used to provide diffusion. The linear layer employs a Feistel function, a rotation to the 3 branches on the rightmost side of the state, and finally, a swap between the leftmost 3

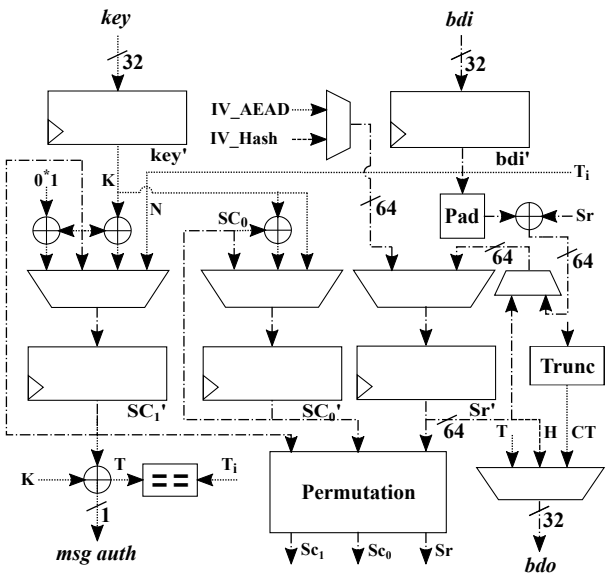


Fig. 13: Block diagram of Ascon implementation. Bus widths are 128 bits unless indicated. The dash-dotted lines are exploited in both AEAD and hash, the dotted lines are only used in AEAD, and the dashed lines are only used in the hash function.

branches and the rightmost 3 branches. The design of this linear layer allows for long trails by leaving half of the words in the state unmodified.

Both the AEAD and the hash recommendations rely upon a sponge-based mode of operation. For AEAD, a slight variation of Beetle is used. The 384-bit state has a rate of 256 bits and a capacity of 128 bits. Prior to executing the permutation, the rate portion of the state is updated by a FeistelSwap function, and a combined feedback function, rho. A constant is injected for domain separation and then finally a rate whitening layer is applied to prevent attackers from gaining partial information from the permutation. For hashing, a more standard sponge mode of operation is used. The 384-bit state has a rate of 128 bits and a capacity of 256 bits. Before the start of each permutation, the Feistel function mentioned in the preceding paragraph is applied to the rate portion of the state.

**Implementation** This implementation of Schwaemm and Esch also uses a basic-iterative architecture. Each of the 4 rounds of the substitution layer requires one clock cycle and this implementation allocates one additional clock cycle for the linear layer. Based on the number of steps, the total clock cycles for the permutation will vary, with 35 cycles needed for STEPS_SLIM and 55 cycles needed for STEPS_BIG. To perform the SPARKLE384 permutation, 6 separate instances of the ARX box Alzette are used. The structure of the permutation

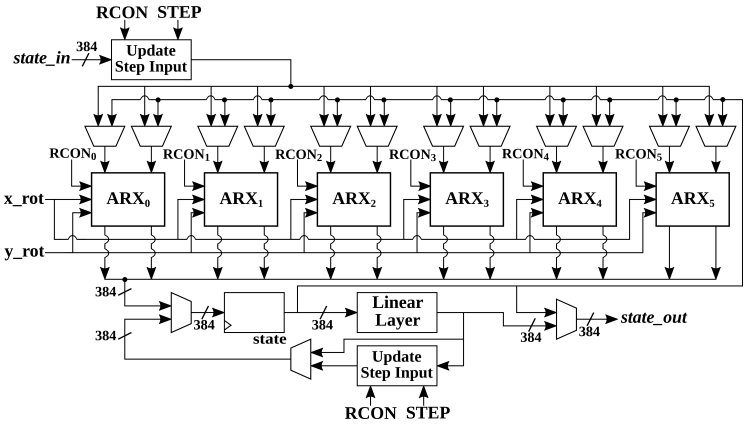


Fig. 14: The SPARKLE384 permutation. Bus widths are 32 bits unless otherwise indicated.

can be seen in Figure 14. For both AEAD and hashing, 10* padding is used to complete partial input blocks. The overall structure of the datapath used for the combined implementation of SCHWAEMM256-128 and ESCH256 can be seen in Figure 15.

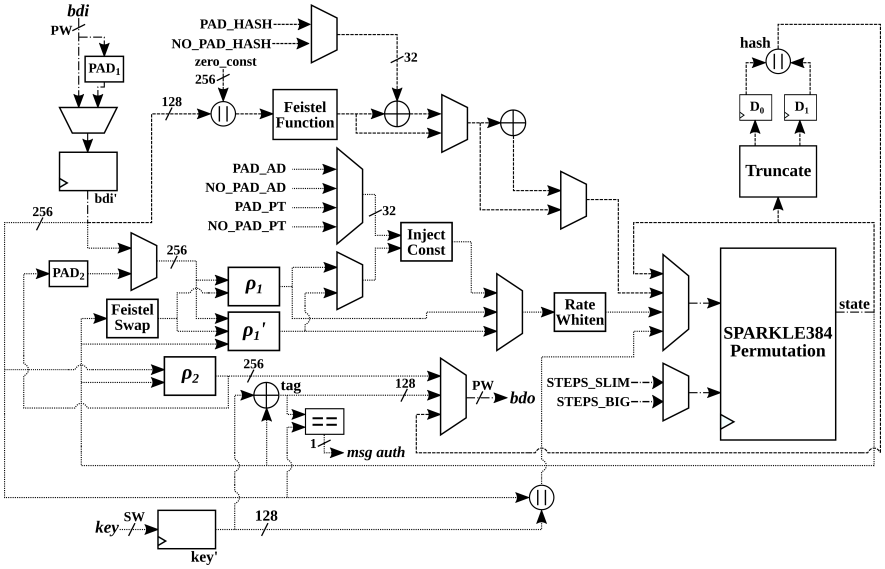


Fig. 15: Block diagram of the combined datapath for SCHWAEMM256-128 and ESCH256. Bus widths are 384 bits unless otherwise indicated. The same dash-dotted, dotted, and dashed line scheme used previously applies here.

### 3.6 Summary

Characteristics of implemented ciphers are summarized in Table 1. All implemented ciphers have nonce and key size of 128 bits, except for SCHWAEMM that has a 256-bit nonce.

Table 1: Characteristics of implemented ciphers.

| Cipher | block (bits) | steps | rounds | state (bits) | rate (bits) | tag (bits) | hash (bits) |
|---|---|---|---|---|---|---|---|
| SpoC-64 | 64 | 18 | 6 | 192 | 64 | 64 | - |
| GIFT-COFB | 128 | - | 40 | 128 | - | 128 | - |
| COMET-AES | 128 | - | 10 | 128 | - | 128 | - |
| COMET-CHAM | 128 | - | 80 | 128 | - | 128 | - |
| ASCON | 64 | - | 6/12 | 320 | 64 | 128 | 256 |
| SCHWAEMM | 256 | 7/11 | 4 | 384 | 256 | 128 | - |
| ESCH | 128 | 7/11 | 4 | 384 | 128 | - | 256 |

Latency and throughput formulas for the implemented ciphers are shown in Table 2. Clock cycles for AEAD encryption operation are shown as $\alpha + \beta \times AD + \gamma \times PT$, where $\alpha$ represents the sum of any initiation and tag generation cycles, and $\beta$ and $\gamma$ are the number of cycles to process one block of $AD$ and $PT$, respectively. For the hash function, the number of clock cycles are shown as $\delta + \epsilon \times M$, where $\delta$ is the sum of cycles of all initialization, finalization, and extraction of the 256-bit $H$ and $\epsilon$ is the number of cycles to absorb one block of message. Latency for both AEAD and hash is defined as the number of clock cycles required to process one block of message from start to end. For long messages, throughput is computed as TP = fclk $\times$ (bits/block)/(cycles/block), where fclk is the maximum achieved clock frequency and cycles/block refers to $\gamma$ and $\epsilon$ for AEAD process and hashing, respectively.

## 4 Results

FPGA implementations in this paper are developed in Verilog or VHDL using RTL design methodology and based on a basic-iterative architecture. Our implementations are compliant with the LWC HW API, and include modules in the LWC HW DP (v1.0.1). Results are implemented in Xilinx Vivado 2018.3 for the Xilinx Artix-7 FPGA (xc7a100tcsg324-3), and optimized for throughput-to-area (TPA) ratio using the Minerva automated hardware optimization tool, introduced in [22] and available for download at [1]. TPA ratio is a useful metric for comparing cipher hardware implementations, since it captures any attempt by the implementer to significantly optimize one characteristic (e.g., area or throughput) at the expense of another. Our implementations are also verified in actual hardware (xc7a100tftg256-3) using the FOBOS [16] and representative

Table 2: Latency and throughput formulas for selected ciphers.

| | Encrypt | Latency | TP |
|---|---|---|---|
| SpoC | $219 + 109 \times AD + 111 \times PT$ | 330 | fclk × 64/111 |
| GIFT-COFB | $102 + 50 \times AD + 53 \times PT$ | 155 | fclk × 128/53 |
| COMET-AES | $37 + 16 \times AD + 20 \times PT$ | 57 | fclk × 128/20 |
| COMET-CHAM | $179 + 87 \times AD + 91 \times PT$ | 270 | fclk × 128/91 |
| Ascon-AEAD | $42 + 9 \times AD + 10 \times PT$ | 52 | fclk × 64/10 |
| Schwaemm | $114 + 43 \times AD + 52 \times PT$ | 166 | fclk × 256/52 |
| | Hash | Latency | TP |
| Ascon-Hash | $60 + 15 \times M$ | 75 | fclk × 64/15 |
| Esch | $63 + 39 \times M$ | 102 | fclk × 128/39 |

test vectors generated by `cryptotvgen` in the LWC HW DP. As discussed in [40], verification in actual hardware is important, since unverified implementations might contain conditions (such as combinational loops or latches) which render them ineffective on actual platforms. Moreover, we implement the ciphers in Xilinx ISE for the Xilinx Spartan-6 FPGA and in Quartus Prime Lite for the Intel Cyclone-V FPGA.

In this work, the power of the selected Round 2 candidates is measured during operation on the NewAE CW305 Artix-7 target board at three different frequencies, i.e., 10, 25, and 40 MHz. The results at 40 MHz are shown in Table 3 and the capturing system architecture is depicted in Fig. 16. In Table 3, $P_{\text{mean}}$ represents the average power over the entire traces, $P_{\text{max}}$ is the highest captured power during any trace, $\Delta P$ is obtained as $\Delta P = (|P_{\text{max}} - P_{\text{mean}}|/P_{\text{mean}}) \times 100$, energy per bit is derived as $E/\text{bit} = P_{\text{mean}}/\text{TP}$, and we obtain the power gradient ($\text{d}P_{\text{mean}}/\text{dFreq}$) using the power results at 10, 25, and 40 MHz frequencies. From this table, we observe that SpoC and Ascon-AEAD have the lowest (30.8 mW) and the highest (46.2 mW) powers at 40 MHz, respectively. COMET-AES has the smallest $E/\text{bit}$, 0.16 nJ/bit and the next lowest is Ascon-AEAD with 0.18 nJ/bit. The maximum $E/\text{bit}$ belongs to SpoC with 1.34 nJ/bit. In the case of the AEAD-Hash implementations, we measured power by considering hash test vectors that contain only message and hash values.

The results of our implementations are shown in Table 4. Additionally, these implementations are available for inspection at [37]. As a disclaimer, our implementations are baseline representations of selected ciphers using basic-iterative architecture; further optimizations may be possible along one or several dimensions, including but not limited to improved throughput or reduced area.

Based on the results of Table 4, SpoC has the highest maximum frequency of 268.0 and 224.4 MHz in Artix-7 and Cyclone-V, respectively, while Ascon-AEAD has the highest frequency of 174.4 MHz in Spartan-6. The lowest frequency in Artix-7 and Spartan-6 belongs to Schwaemm and Esch at 106.0 and 61.0 MHz, respectively, and it belongs to Schwaemm at 72.0 MHz in Cyclone-V. In terms of area in FPGA look-up tables (LUTs) or adaptive logic modules (ALMs) in

Table 3: Power and energy characteristics of the selected ciphers at 40 MHz.

| | $P_{\text{mean}}$ mW | $P_{\text{max}}$ mW | $\Delta P$ % | $E$/bit nJ/bit | Gradient d$P$/d$Freq$ |
|---|---|---|---|---|---|
| SpoC | 30.8 | 35.4 | 14.94 | 1.34 | 0.13 |
| GIFT-COFB | 32.1 | 36.7 | 14.33 | 0.33 | 0.17 |
| COMET-AES | 41.6 | 47.7 | 14.66 | 0.16 | 0.39 |
| COMET-CHAM | 32.1 | 36.3 | 13.08 | 0.57 | 0.19 |
| Ascon-AEAD | 46.2 | 53.2 | 15.15 | 0.18 | 0.47 |
| Ascon-Hash | 34.0 | 37.5 | 10.29 | 0.20 | 0.26 |
| Schwaemm | 39.1 | 43.7 | 11.76 | 0.20 | 0.43 |
| Esch | 39.3 | 44.7 | 13.74 | 0.30 | 0.43 |

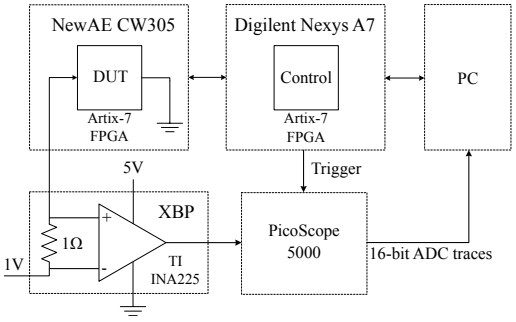


Fig. 16: Power Capturing system architecture.

Cyclone-V, SpoC is the smallest with 1172 LUTs, 1364 LUTs, and 678 ALMs in Artix-7, Spartan-6, and Cyclone-V, respectively. Schwaemm and Esch has the largest area with 5002 LUTs, 6279 LUTs, and 3548 ALMs in Artix-7, Spartan-6, and Cyclone-V, respectively. Ascon-AEAD has the highest TP at 1683.2, 1116.4, and 1295.6 Mbps in Artix-7, Spartan-6, and Cyclone-V, respectively. The lowest TP is held by SpoC at 154.5, 75.6, and 129.4 Mbps in Artix-7, Spartan-6, and Cyclone-V, respectively. In terms of TPA ratio, Ascon-AEAD is the highest at 0.887 Mbps/LUT, 0.584 Mbps/LUT, and 1.233 Mbps/ALM in Artix-7, Spartan-6, and Cyclone-V, respectively. Schwaemm with 0.121 Mbps/LUT in Artix-7, SpoC with 0.055 Mbps/LUT in Spartan-6, and COMET-CHAM with 0.069 Mbps/ALM in Cyclone-V have the smallest TPA ratios. Additionally, the latency of our Ascon-AEAD and COMET-AES implementations, respectively, are only 16% and 17% of the latency of our SpoC implementation. This provides Ascon-AEAD and COMET-AES the advantage of processing very short messages, which is a desirable characteristic for LWC candidates identified in [35].

We do not compute TPA ratios for AEAD+Hash implementations; this is not a relevant metric, since multiple components with different TP share the

same area. However, in terms of comparison between hash implementations, we note that Ascon-AEAD+Hash has 36% of the area, 3.3 × the TP, uses 87% of the power and is 33% more energy efficient than Schwaemm+Esch, where TP, power and energy are measured in terms of hash-only test vectors.

## 5 Analysis

### 5.1 Comparison with selected previous authenticated cipher implementations

Previous hardware implementations during CAESAR and those provided as part of the NIST LWC submissions establish a basis for comparison with cipher implementations in this paper. Since LWC HW API is very similar to CAESAR HW API, we bring some CAESAR API-compliant examples from [23, 19] in Table 4 for comparison. Since these examples use the lightweight CAESAR HW DP, which only appeared at the end of 2017, there are fewer available examples. The earlier version of the CAESAR HW DP was designed for high speed implementations and included functionality not used by many ciphers, and exacted a larger toll on area overhead. The AES-GCM in Table 4 is compliant with the George Mason University (GMU) HW API [26], which is close to the CAESAR HW API. Moreover, it also uses basic-iterative architecture which allows a better comparison with our implementations. The AES-GCM results are obtained with the same optimization and simulation tools as our implementations.

A full-scale comparison with authors' implementations of NIST LWC candidates is premature, since authors reported results for implementations that are not compliant with the LWC API. Some representative examples of block and sponge cipher FPGA implementations, e.g., ESTATE (ESTATE-TweGIFT-128), SAEAES, and Oribatida (Oribatida-256-64), are included in Table 4. All CAESAR and NIST LWC implementations provided for comparison use a 128-bit key; TP is computed based on the processing rate of a large number of blocks of plaintext into ciphertext. The range of TPA ratios (0.043 to 0.073) for the Round 3 CAESAR candidates, i.e., Ascon-small, CLOC-AES, and SILC-AES, is analogous to the range of TPA ratios for some of our Spartan-6 implementations: SpoC (0.055), Schwaemm (0.065), and COMET-CHAM (0.088). However, the Spartan-6 TPA ratios of GIFT-COFB (0.165), COMET-AES (0.269), and Ascon-AEAD (0.584) are noticeably higher. Although the implementations of sampled NIST LWC candidates are based on a different FPGA platform and the range of TPA ratios (0.547 to 0.757) is greater than some of our ciphers, it is close to some other implementations, such as Ascon-AEAD. Regarding AES-GCM, its TPA ratio is higher than most of our implementations, but Ascon-AEAD is yet higher in all three FPGAs. AES-GCM has a high throughput because of its low latency. However, in terms of area, which is a more important feature than throughput in lightweight designs, it is larger than most observed LWC ciphers.

A judgement as to whether or not these implementations are better than either our implementations or previous CAESAR implementations would be premature, since no uniform standard has been established for benchmarking hardware

Table 4: Results of implementations in this work (TW), and comparison with CAESAR lightweight and NIST LWC candidates. The units of Freq, Area, TP, and TPA are MHz, LUTs (ALMs for Cyclone-V), Mbps, and Mbps/LUT (Mbps/ALM), respectively.

| Cipher | Type | FPGA | Freq | Area | TP | TPA | Ref |
|---|---|---|---|---|---|---|---|
| **CAESAR and Reference Implementation** | | | | | | | |
| Ascon-small | Sponge | Spartan-6 | 146.1 | 1640 | 114.0 | 0.070 | [19] |
| CLOC-AES | Block | Spartan-6 | 101.9 | 1604 | 68.7 | 0.043 | [23] |
| SILC-AES | Block | Spartan-6 | 109.0 | 1052 | 76.6 | 0.073 | [23] |
| AES-GCM | Block | Artix-7 | 222.0 | 3268 | 2583.3 | 0.790 | [26] |
| | | Spartan-6 | 144.7 | 3350 | 1683.6 | 0.503 | |
| | | Cyclone-V | 165.9 | 2651 | 1930.7 | 0.728 | |
| **NIST LWC** | | | | | | | |
| **SpoC** | Sponge | Artix-7 | 268.0 | 1172 | 154.5 | 0.132 | TW |
| | | Spartan-6 | 131.0 | 1364 | 75.6 | 0.055 | |
| | | Cyclone-V | 224.4 | 678 | 129.4 | 0.191 | |
| **GIFT-COFB** | Block | Artix-7 | 263.0 | 1932 | 635.2 | 0.329 | TW |
| | | Spartan-6 | 134.3 | 1960 | 324.3 | 0.165 | |
| | | Cyclone-V | 198.2 | 1122 | 478.7 | 0.427 | |
| **COMET-AES** | Block | Artix-7 | 251.0 | 2753 | 1606.4 | 0.584 | TW |
| | | Spartan-6 | 128.6 | 3058 | 822.9 | 0.269 | |
| | | Cyclone-V | 118.0 | 3383 | 755.3 | 0.223 | |
| **COMET-CHAM** | Block | Artix-7 | 201.0 | 2214 | 282.7 | 0.128 | TW |
| | | Spartan-6 | 145.5 | 2338 | 204.7 | 0.088 | |
| | | Cyclone-V | 131.5 | 2696 | 184.9 | 0.069 | |
| **Ascon-AEAD** | Sponge | Artix-7 | 263.0 | 1898 | 1683.2 | 0.887 | TW |
| | | Spartan-6 | 174.4 | 1913 | 1116.4 | 0.584 | |
| | | Cyclone-V | 202.4 | 1051 | 1295.6 | 1.233 | |
| **Ascon-AEAD+Hash** | Sponge | Artix-7 | 242.0 | 2181 | 1032.5 | - | TW |
| | | Spartan-6 | 158.9 | 2188 | 678.1 | - | |
| | | Cyclone-V | 210.5 | 1064 | 898.0 | - | |
| **Schwaemm** | Sponge | Artix-7 | 106.0 | 4313 | 521.8 | 0.121 | TW |
| | | Spartan-6 | 65.8 | 5005 | 323.9 | 0.065 | |
| | | Cyclone-V | 72.0 | 2802 | 354.5 | 0.127 | |
| **Schwaemm+Esch** | Sponge | Artix-7 | 106.0 | 5002 | 347.9 | - | TW |
| | | Spartan-6 | 61.0 | 6279 | 200.1 | - | |
| | | Cyclone-V | 75.5 | 3548 | 247.8 | - | |
| ESTATE | Block | Virtex-7 | 580.1 | 1413 | 928.3 | 0.657 | [17] |
| SAEAES | Block | Virtex-7 | 145.9 | 348 | 263.3 | 0.757 | [32] |
| Oribatida | Sponge | Virtex-7 | 554.2 | 940 | 514 | 0.547 | [14] |

implementations in the NIST LWC standardization process. For instance, an implementation "compliant with the LWC API" is required to include hardware necessary for input and output LWC data in specified protocol, and must also realize "corner cases" (e.g., null blocks, partial blocks, padding, and truncating) which often involve significant resources.

### 5.2 Observations

COMET-CHAM and COMET-AES use the same mode of operation but with different underlying ciphers, i.e., CHAM and AES, respectively. Considering the area results of different FPGAs, we observe that COMET-CHAM is 20% smaller than COMET-AES, which makes it a more lightweight cipher than AES. However, COMET-CHAM needs 80 rounds due to the CHAM ARX round function, thus leading to a smaller TP than COMET-AES.

Based on the results of Table 3 and Table 4, Ascon-AEAD and COMET-AES have both the highest powers and throughput among the rest of the ciphers, and because of high TP, they are the most energy efficient (i.e., smallest E/bit). Although Ascon-AEAD has a smaller area, its power is higher than COMET-AES due to its larger internal state size.

Power gradients are the best predictor of dynamic power consumption at higher frequencies, since increasing power is highly linear with frequency. SpoC, GIFT-COFB, and COMET-CHAM together have the lowest average powers and gradients, which is a positive quality for lightweight ciphers.

Finally, implementations of [19, 23] have lightweight targets, while our implementations use basic-iterative architecture. A full comparison of NIST LWC candidates implemented with lightweight (e.g., reduced area, performance, and power) targets is the scope of future work.

## 6 Conclusion

We provided a medium-scale comparison of 3rd-party FPGA implementations of selected NIST LWC standardization process Round 2 candidates. Candidates examined in this paper, SpoC, GIFT-COFB, COMET-AES, COMET-CHAM, Ascon, and Schwaemm and Esch are representative of many of the 32 NIST LWC submissions accepted to Round 2. Implementations are fully-compliant with the newly-released LWC HW API for lightweight cryptography, use the associated LWC HW developer's package, and are verified to operate in actual hardware using the FOBOS test bench. Furthermore, they are implemented in different FPGAs (Artix-7, Spartan-6, and Cyclone-V), and power and energy per bit of each cipher are measured at 40 MHz on a hardware instance of the Artix-7.

Our results show that SpoC and Ascon-AEAD have the highest maximum frequencies (both 2.5× greater than Schwaemm). Moreover, SpoC has the lowest area (26% of the LUTs or ALMs of Schwaemm) and the lowest power at 40 MHz (33% smaller than Ascon-AEAD). Ascon-AEAD has the highest throughput (TP) (12× greater than SpoC), and the highest throughput-to-area (TPA) ratio

(8.6× more than SCHWAEMM). Meanwhile, COMET-AES and ASCON-AEAD are the most energy efficient, and SpoC, GIFT-COFB, and COMET-CHAM have the lowest increase in dynamic power with increasing frequency.

The magnitude of differences in maximum frequencies, area, power, TP, and TPA ratios supports the rationale for gaining information on hardware implementations as early as possible in any cryptographic contest. The TPA ratio results of some of the implemented ciphers are similar to the results reported for CAESAR HW API compliant late-round CAESAR candidates, but have TPA ratios which are less than TPA ratios reported for a selected group of NIST LWC submission author implementations of ciphers of similar construction. On the other hand, some cipher implementations have TPA ratios better than CAESAR candidates and close to the LWC authors' reports. Thus, no conclusion can be drawn regarding the relative hardware merits of candidates implemented according to different compliance standards, which reinvigorates the need for a standardized hardware API and minimum compliance criteria for the NIST LWC standardization process.

## References

1. Minerva: Automated Hardware Optimization Tool, https://cryptography.gmu.edu/athena/index.php?id=Minerva
2. Aagaard, M., AlTawy, R., Gong, G., Mandal, K., Rohit, R.: ACE: An Authenticated Encryption and Hash Algorithm Submission to the NIST LWC Competition (Mar 2019), https://csrc.nist.gov/Projects/Lightweight-Cryptography/Round-2-Candidates
3. AlTawy, R., Gong, G., He, M., Jha, A., Mandal, K., Nandi, M., Rohit, R.: SpoC: An Authenticated Cipher Submission to the NIST LWC Competition (Feb 2019), https://csrc.nist.gov/Projects/Lightweight-Cryptography/Round-2-Candidates
4. AlTawy, R., Gong, G., He, M., Mandal, K., Rohit, R.: Spix: An Authenticated Cipher Submission to the NIST LWC Competition (Mar 2019), https://csrc.nist.gov/Projects/Lightweight-Cryptography/Round-2-Candidates
5. Andres Lara-Nino, C., Díaz-Pérez, A., Morales-Sandoval, M.: FPGA-Based Assessment of Midori and Gift Lightweight Block Ciphers: 20th International Conference, ICICS 2018, Lille, France, October 29-31, 2018, Proceedings, pp. 745–755 (10 2018). https://doi.org/10.1007/978-3-030-01950-1_45
6. ARM: AMBA Specifications, http://www.arm.com/products/system-ip/amba-specifications.php
7. Banik, S., Bogdanov, A., Peyrin, T., Sasaki, Y., Sim, S.M., Tischhauser, E., Todo, Y.: SUNDAE-GIFT: An Authenticated Cipher Submission to the NIST LWC Competition (Mar 2019), https://csrc.nist.gov/Projects/Lightweight-Cryptography/Round-2-Candidates
8. Banik, S., Chakraborti, A., Iwata, T., Minematsu, K., Nandi, M., Peyrin, T., Sasaki, Y., Sim, S.M., Todo, Y.: GIFT-COFB: An Authenticated Cipher Submission to the NIST LWC Competition (Mar 2019), https://csrc.nist.gov/Projects/lightweight-cryptography/Round-2-candidates
9. 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: International

Conference on Cryptographic Hardware and Embedded Systems (CHES). pp. 321–345 (Sep 2017)
10. Beierle, C., Biryukov, A., Cardoso dos Santos, L., Großschadl, J., Perrin, L., Udovenko, A., Velichkov, V., Wang, Q.: SCHWAEMM and ESCH: An Authenticated Cipher Submission to the NIST LWC Competition (Sep 2019), https://csrc.nist.gov/Projects/Lightweight-Cryptography/Round-2-Candidates
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: Robshaw, M., Katz, J. (eds.) Advances in Cryptology – CRYPTO 2016. pp. 123–153. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
12. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications. In: Selected Areas in Cryptography. pp. 320–337 (2012)
13. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Permutation-based Encryption, Authentication and Authenticated Encryption. DIAC, Stockholm, Sweden (2012)
14. Bhattacharjee, A., List, E., López, C.M., Nandi, M.: The Oribatida Family of Lightweight Authenticated Encryption Schemes (Mar 2019), https://csrc.nist.gov/Projects/Lightweight-Cryptography/Round-2-Candidates
15. CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness: Cryptographic competitions (January 2016), http://competitions.cr.yp.to/index.html
16. CERG: Flexible Open-source workBench fOr Side-channel analysis (FOBOS) (Oct 2016), https://cryptography.gmu.edu/fobos/
17. Chakraborti, A., Datta, N., Jha, A., Lopez, C.M., Nandi, M., Sasaki, Y.: ESTATE (Mar 2019), https://csrc.nist.gov/Projects/Lightweight-Cryptography/Round-2-Candidates
18. Chakraborti, A., Datta, N., Nandi, M., Yasuda, K.: Beetle family of lightweight and secure authenticated encryption ciphers. IACR Transactions on Cryptographic Hardware and Embedded Systems **2018**(2), 218–241 (May 2018), https://tches.iacr.org/index.php/TCHES/article/view/881
19. Diehl, W., Farahmand, F., Abdulgadir, A., Kaps, J.P., Gaj, K.: Face-off between the caesar lightweight finalists: Acorn vs. ascon. 2018 International Conference on Field-Programmable Technology (Dec 2018)
20. Dobraunig, C., Eichlseder, M., Mendel, F., Schläffer, M.: Ascon v1.2 (Mar 2019), https://csrc.nist.gov/Projects/Lightweight-Cryptography/Round-2-Candidates
21. Elbirt, A., Yip, W., Chetwynd, B., Paar, C.: An FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists (10 2002)
22. Farahmand, F., Ferozpuri, A., Diehl, W., Gaj, K.: Minerva: Automated Hardware Optimization Tool. In: 2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)
23. Farahmand, F., Diehl, W., Abdulgadir, A., Kaps, J.P., Gaj, K.: Improved lightweight implementations of caesar authenticated ciphers. pp. 29–36 (04 2018). https://doi.org/10.1109/FCCM.2018.00014
24. Gaj, K., Chodowiec, P.: Comparison of the Hardware Performance of the AES Candidates Using Reconfigurable Hardware. pp. 40–54 (01 2000)
25. George Mason University: Source Code for AES v1.3 (Jun 2016), https://cryptography.gmu.edu/athena/index.php?id=CAESAR_source_codes

26. George Mason University: Source Code for AES_GCM v1.0 (Jun 2016), https://cryptography.gmu.edu/athena/index.php?id=CAESAR_source_codes
27. Gueron, S., Jha, A., Nandi, M.: COMET: An Authenticated Cipher Submission to the NIST LWC Competition (Mar 2019), https://csrc.nist.gov/Projects/Lightweight-Cryptography/Round-2-Candidates
28. Homsirikamol, E., , Diehl, W., Ferozpuri, A., Farahmand, F., Lyons, M.X., Yalla, P., Gaj, K.: Toward Fair and Comprehensive Benchmarking of CAESAR Candidates in Hardware: Standard API, High-Speed Implementations in VHDL/Verilog, and Benchmarking Using FPGAs (Sep 2016), https://cryptography.gmu.edu/athena/presentations/GMU_DIAC_2016_RTL.pdf
29. Kaps, J.P., Diehl, W., Tempelmeier, M., Homsirikamol, E., Gaj, K.: Hardware API for Lightweight Cryptography (Oct 2019), https://cryptography.gmu.edu/athena/LWC/LWC_HW_API.pdf
30. Koo, B., Roh, D., Kim, H., Jung, Y., Lee, D.G., Kwon, D.: Cham: A family of lightweight block ciphers for resource-constrained devices. In: Kim, H., Kim, D.C. (eds.) Information Security and Cryptology – ICISC 2017. pp. 3–25. Springer International Publishing, Cham (2018)
31. Krawczyk, H.: The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?). In: Kilian, J. (ed.) Advances in Cryptology — CRYPTO 2001. pp. 310–331. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
32. Naito, Y., Matsui, M., Sakai, Y., Suzuki, D., Sakiyama, K., Sugawara, T.: SAEAES (Feb 2019), https://csrc.nist.gov/Projects/Lightweight-Cryptography/Round-2-Candidates
33. National Institute of Standards and Technology: Announcing Request for Candidate Algorithm Nominations for the Advanced Encryption Standard National Institute of Standards and Technology (Sep 1997)
34. National Institute of Standards and Technology: Report on the development of the Advanced Encryption Standard (AES) (Oct 2000), http://csrc.nist.gov/archive/aes/round2/r2report.pdf
35. National Institute of Standards and Technology: Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process (Aug 2018), https://csrc.nist.gov/projects/lightweight-cryptography
36. Rezvani, B., Diehl, W.: Detailed Characteristics of NIST Lightweight Cryptography Project Round 1 Submissions (v2) (Jul 2019), https://rijndael.ece.vt.edu/wdiehl/
37. SAL: NIST Lightweight Cryptography Implementations (Jan 2019), https://github.com/vtsal
38. Standard, N.F.: Announcing the advanced encryption standard (aes). Federal Information Processing Standards Publication **197**(1-51), 3–3 (2001)
39. Tempelmeier, M., Farahmand, F., Homsirikamol, E., Diehl, W., Kaps, J., Gaj, K.: Implementer's Guide to Hardware Implementations Compliant with the Hardware API for Lightweight Cryptography v1.0.1 (Nov 2019), https://cryptography.gmu.edu/athena/LWC/LWC_HW_Implementers_Guide.pdf
40. Tempelmeier, M., De Santis, F., Sigl, G., Kaps, J.P.: The caesar-api in the real world — towards a fair evaluation of hardware caesar candidates. pp. 73–80 (04 2018). https://doi.org/10.1109/HST.2018.8383893
41. University, G.M.: Development package for hardware implementations compliant with the hardware api for lightweight cryptography v1.0.1 (Nov 2019), https://cryptography.gmu.edu/athena/index.php?id=LWC