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

Behnaz Rezvani and William Diehl

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

**Abstract.** 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, i.e., even in round 1, is preferable. In this research, we implement three NIST LWC round 1 candidate ciphers, SpoC, Spook, and GIFT-COFB, in the Artix-7 FPGA. Implementations are compliant with the previously-validated CAESAR Hardware Applications Programming Interface (API) for Authenticated Ciphers, and are tested in actual hardware. Implementations show that GIFT-COFB has the highest Throughput-to-Area (TPA) ratio, by a 4.4 factor margin over Spook. Additionally, the results illustrate hardware implementation challenges associated with integrating multiple cryptographic primitives into one design, as well as complexities due to padding and truncation.

**Keywords:** Lightweight cryptography · FPGA · Authenticated cipher · Encryption

## 1 Introduction

Increasing use of very lightweight devices constituting the Internet of Things (IoT) necessitates the development and adaptation of lightweight cryptographic algorithms. Yet lightweight cryptography should not be less robust than existing protections, as real-world events have shown IoT devices to be a potential "achilles heel" of cybersecurity. For example, in the October 2016 Dyn cyber attack, Mirai malware installed on millions of IoT devices (e.g., web cameras, baby monitors, residential gateways) overwhelmed the Dyn Domain Name Server (DNS) with DNS resolution requests, taking down large parts of major internet-enabled companies in Europe and North America [22, 33].

IoT devices, like all 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 guard 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 in 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 for lightweight AEAD and optional hashes, to be subjected to several rounds of evaluation as part of the Lightweight Cryptography (LWC) Standardization Process, and eventually incorporated into U.S. Federal Information Processing Standards (FIPS) [37]. Submissions of specifications were permitted until February 2019, and 56 qualified round 1 candidates were publicized in April 2019. Round 2 selections are expected as early as August 2019.

NIST LWC candidates are evaluated on several criteria, including cost (e.g., area, memory, energy consumption) and performance (e.g., latency, throughput, 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 weight to 3rd-party implementations.

In this research, we provide an early evaluation of selected NIST LWC AEAD candidate submissions through hardware implementations; optional hash algorithms are not considered in this research. Given the large number of qualified submissions (i.e., 56) and the short period of time between initial publication of specifications and round 2 selection (i.e., less than 4 months), we select three ciphers for evaluation: SpoC, Spook, and GIFT-COFB. The rationale for selection of these ciphers is as follows:

1. According to analysis in [38], at least 47 submissions are composed of block- or block-like primitives, out of which at least 20 are sponge-based. Additionally, at least 22 ciphers (both block and stream) use 4-bit S-Boxes, and at least 9 ciphers use a logical AND or multiplication for non-linear transformations. SpoC and Spook are sponge-based, while Spook and GIFT-COFB use 4-bit S-Boxes, and SpoC uses a logical AND for non-linear transformations.
2. While SpoC and GIFT-COFB have author ASIC implementations, none of the chosen ciphers has author-reported FPGA implementations in the submissions.
3. No weaknesses or errors in specification had been publicly identified in the selected ciphers at the time of implementation.

We implement the selected ciphers using Register Transfer Level (RTL) methodology in Verilog or VHDL. Since no hardware Applications Programming Interface (API) was specified for NIST LWC round 1 submissions, all implementations are fully compliant with the existing CAESAR Hardware Applications Programming Interface (CAESAR HW API) [31, 29], and use the CAESAR Hardware Developers Package (CAESAR HW DP) at [24]. Implementations are functionally verified in Xilinx Vivado Simulator, and verified on actual hardware using the Flexible Open-source workBench fOr Side-channel analysis (FOBOS) [16]. Results are generated in the Artix-7 FPGA by Xilinx Vivado, and further optimized using the Minerva Automatic Hardware Optimization Tool [20]. Implementations are compared according to maximum frequency, area (Look-up Tables (LUTs) and slices), throughput (Mbps), and Throughput-to-Area (TPA) ratios.

Our contributions in this work are as follows:

1. We provide an accelerated look at fully-functional hardware implementations of selected and representative NIST LWC AEAD candidates, before round 2 selections.
2. We provide a comparison with selected past and present authenticated cipher implementations, in order to bridge the space between CAESAR and NIST LWC processes, as well as API-compliant and API-non-compliant implementations.
3. We show examples of features of cipher implementations which may be resource-intensive and reduce performance, in order to illustrate design pitfalls and help guide later-round tweaks.

The paper is organized as follows: We provide background on previous hardware implementations, authenticated ciphers, the CAESAR API and associated Developer's Package in Section 2. We describe implementations of our chosen cipher candidates in Section 3. We present implementation results in Section 4. We compare with previous results and discuss observations in Section 5. We conclude in Section 6 and discuss future work in Section 7.

## 2 Background

### 2.1 Hardware implementations in cryptographic competitions

Public competitions for cipher standards began with the NIST call for AES submissions in 1997 [35]. 15 candidates were submitted to round 1 of the competition by 1998, with five candidates selected as finalists in 1999. The Rijndael algorithm was chosen for AES standardization in 2000. Candidates in round 1 were evaluated primarily based on software, with more intended focus on hardware in round 2. Large-scale comparisons of hardware performance did not emerge until near the end of the selection process, e.g., [23, 19]. This means that hardware implementations were emphasized only in the last third of the competition. As stated in [36], 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 the Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR) [15]. Hardware implementations were required from all submission teams beginning with round 3, out of ultimately 4 rounds of selection. A standard API for hardware implementations was validated by the CAESAR committee, and a hardware developers package designed to help implementers meet API compliance standards was made available. This resulted in CAESAR HW API-compliant hardware implementation of 28 out of 29 round 2 candidates before the end of round 2 [30].

However, the timeframe 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, and 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 call for specifications predicted a nearly one-year process for a "first-phase evaluation" or candidates. However, the public announcement of round 1 submissions, posted in April 2019, announced that round 2 selections would be made "before September 2019," leaving less than 4 months for round 1 evaluations.

The NIST call for submissions (like previous competitions) mandated calls for 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., ACE, CiliPadi, DryGASCON, ESTATE, Gimli, Lilliput, Limdolen, Oribatida, SAEAES, Subterranean, Sycon, TRIFLE, HERN & HERON, WAGE, etc. [38]. However, [37] specifically highlights the value of 3rd-party evaluations of candidate submissions. To the best of our knowledge, these are the first reported 3rd-party hardware implementations of NIST LWC candidates following the publication of official specifications in April 2019.

### 2.2 Authenticated encryption with associated data

Authenticated encryption is a way of ensuring confidentiality, integrity, and authenticity of data by combining the use of a cipher primitive (e.g., block cipher) with a keyed-hash or Message Authentication Code (MAC) to guard against

forgeries. AEAD is defined in [39], where certain header or protocol data should not be encrypted, but should be considered in the integrity and authenticity of transmitted 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., [32]). 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

The top-level function prototypes for AEAD C-language software reference implementations are specified in [37] as follows:

```
// aead_encrypt
int crypto_aead_encrypt( unsigned char *c,unsigned long long *clen,
    const unsigned char *m,unsigned long long mlen, const unsigned char
    *ad,unsigned long long adlen, const unsigned char *nsec, const
    unsigned char *npub, const unsigned char *k);
```

```
// aead_decrypt
int crypto_aead_decrypt( unsigned char *m,unsigned long long *mlen,
    unsigned char *nsec, const unsigned char *c,unsigned long long clen,
    const unsigned char *ad,unsigned long long adlen, const unsigned
    char *npub, const unsigned char *k);
```

Note that in the above prototypes, `unsigned char *m` corresponds to $PT$, `const unsigned char *ad` corresponds to $AD$, etc. (however, `unsigned char *c` actually represents $CT \parallel Tag$). These prototypes form the basis for an API which determines how clients will interact with the program. This ensures compatibility among all similar software implementations, and some fairness in evaluation (other "fairness" measures include language, compiler, and optimization instructions).

However, as in the case of CAESAR, no analogous hardware API was issued in the call for specifications. In Spring 2016, close to the conclusion of CAESAR round 2, the CAESAR committee endorsed the CAESAR HW API. Described at [31, 29], the CAESAR HW API outlines interface standards and minimum compliance criteria to ensure compatibility between 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 [5], 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*. An example of a message, including its representative encoding based on a 32-bit external interface width (i.e., public data bus width $PW = 32$), along with a brief interpretation, is provided in Table 1.

**Table 1.** Description of protocol to load and process one message using the CAESAR Hardware API.

| Protocol | Coding | Description |
|---|---|---|
| instruction = ACTKEY | 70000000 | Activate new key |
| instruction = ENC | 20000000 | Conduct aead_encrypt |
| seg_0_header | D2000010 | Next data is 16 bytes of *Npub*, is the end of type (*Npub*), but is not the end of input |
| seg_0 = *Npub* | 2A9C6AD7<br>F8DE7374<br>A760388E<br>47E557F0 | 128-bit *Npub* (delivered on 4 consecutive reads from the 32-bit pdi_data) |
| seg_1_header | 12000001 | Next data is 1 byte associated data, is the end of type ($AD$), but is not the end of input |
| seg_1=$AD$ | 93000000 | 1 byte of $AD$ (0x93), delivered on 1 read from 32-bit pdi_data |
| seg_2_header | 47000001 | Next data is 1 byte of plaintext, is the end of type ($PT$), and is the end of input |
| seg_2_Msg | DD000000 | 1 byte of $PT$ (0xDD), delivered on 1 read from 32-bit pdi_data |

Since a hardware API has not been defined for the NIST LWC Standardization Process at the time of writing, we implement ciphers using the CAESAR HW API. As NIST LWC AEAD software function prototypes are identical to those used in

CAESAR, we expect similar compatibility with hardware implementations. Of note, an API definition of hash was not included in the CAESAR HW API, but is expected to be included in a future NIST LWC HW API.

### 2.4 Hardware Developer's Package

Whereas external client interface through the software API is handled by typical software tools such as compiler, loader, linker, and operating system, hardware designers are responsible for their own formatted input and output. To facilitate the hardware designer's task of meeting CAESAR HW API requirements, a Hardware Developer's Package is provided at [24]. The package includes an input processor (Pre-Processor) and output processor (Post-Processor), which are encapsulated in a top-level module called AEAD. A designer can place a custom design in a subordinate module called CipherCore, and use standardized interfaces to communicate with Pre- and Post-Processors. A set of Python scripts called `aeadtvgen` is used to generate representative test vectors directly from the software reference implementation, and an accompanying HDL test bench (AEAD_TB) automatically verifies test vectors against expected results. An implementer's guide to assist in using the developer's package is also available at [28].

In this research, we use the HW Developer's Package (v2.0), and develop RTL implementations inside CipherCore. Software reference implementations can be ported directly into `aeadtvgen` for generation of AEAD test vectors. A representation of CipherCore, instantiated in AEAD with accompanying I/O modules, is shown in Fig. 1. External signals defined in the CAESAR HW API are the AXI-4 compatible signals associated with `pdi`, `sdi`, and `do`. Internal input signals to CipherCore, 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 Post-Processor, defined only in the developer's package, include `bdo` (block data output), in which *CT* and *Tag* depart CipherCore, and `msg_auth` (message authentication), which is used to signal results of tag verification in authenticated decryption. Other signals are defined in [28].

In the CAESAR HW API and the accompanying developer's package, 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 practicality of internally buffering potentially long messages (as long as $2^{32} - 1$ bytes in the case of CAESAR) and concurrent evaluation by automated test benches.

## 3 Ciphers implemented in this research

### 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

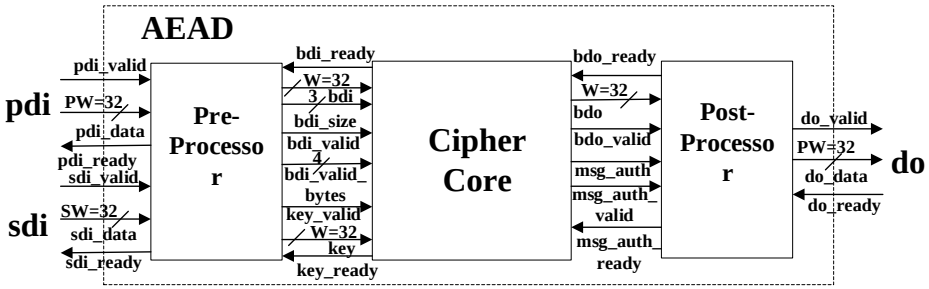


**Fig. 1.** Instantiation of CipherCore inside AEAD, together with modules and signals from CAESAR HW Developer's Package.

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 $t = 64$ 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, 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 Simeck Box (SB). SBs consist of XORs, bitwise rotations, and a 48-bit logical AND. Bitwise rotations are especially advantageous in hardware. The implementation of each SB is shown in Fig. 2.

In order to follow the basic-iterative construction, we construct two SBs (SB1 and SB3) on 48-bits each, which operates 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. 3.

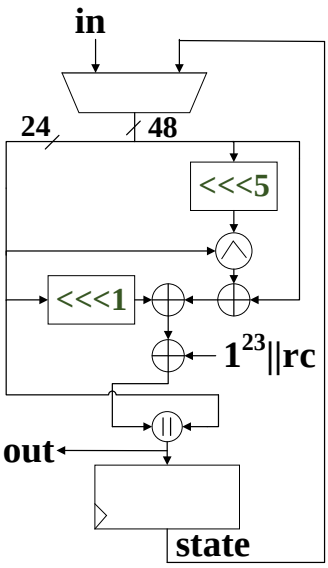


**Fig. 2.** Simeck box construction.

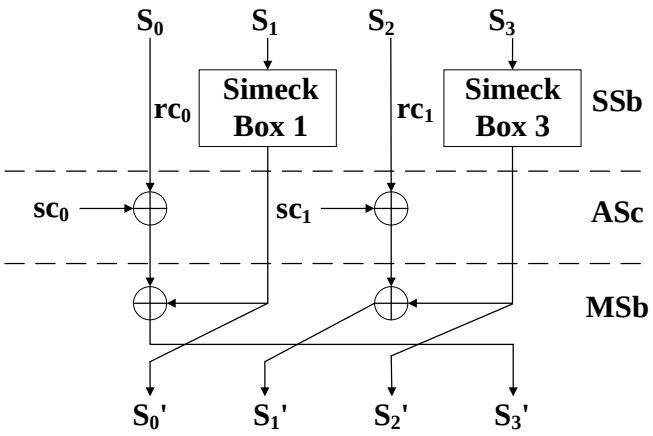


**Fig. 3.** sLiSCP permutation. Bus widths are 48 bits.

The duplex sponge construction of SpoC is shown in Fig. 4. 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 $Y_0 \parallel Z_0$ 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

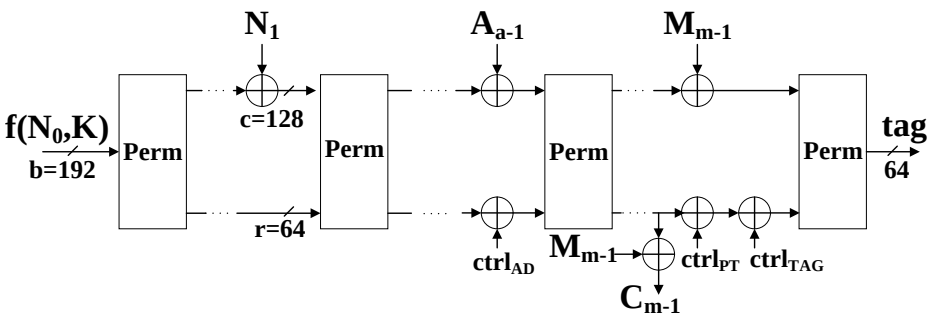


**Fig. 4.** SpoC duplex sponge construction.

**Implementation** We implement a basic iterative architecture based on the sLiSCP permutation, where 1 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 Hardware Developer's Package v2.0, so it is implemented in our CipherCore based on the number of valid input bytes provided by `bdi_valid_bytes`. Additionally, $|CT|$ must equal $|PT|$ 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. 5.

### 3.2 Spook

**Description** Spook uses Sponge one Pass (S1P), and is based on duplex sponge construction [11]. The Spook AEAD algorithm uses two primitives, the Clyde-128 Tweakable Block Cipher (TBC), and the Shadow-512 permutation. The Clyde-128 TBC is similar to the block cipher used in the CAESAR candidate SCREAM in that it is based on Tweakable-LS (TLS) construction, but with new L-Boxes, S-Boxes, and round constants [25, 26]. The tweak is used to generate a tweaked key (or "tweakey"), is updated once per step (with 2 rounds per step), and has a period of 3. Therefore, encryption and decryption both start with the same initial tweak given the authors' primary recommendation of $N_S = 6$ steps.

The Shadow-512 permutation uses the same definitions for L-Boxes and S-Boxes, and similar definition for round constants, as the TBC. However, it is performed across a $b$-bit state (e.g., $b = 512$ in our implementation), and employs encryption only, i.e., there are no inverse L-Boxes or S-Boxes. Additionally, there is a diffusion function, adapted from the Midori cipher, which acts across all $b$-bits of state, and is implemented in 32-bit words according to Alg. 1 [6]. The Shadow-512 permutation typically receives the most invocations in long messages

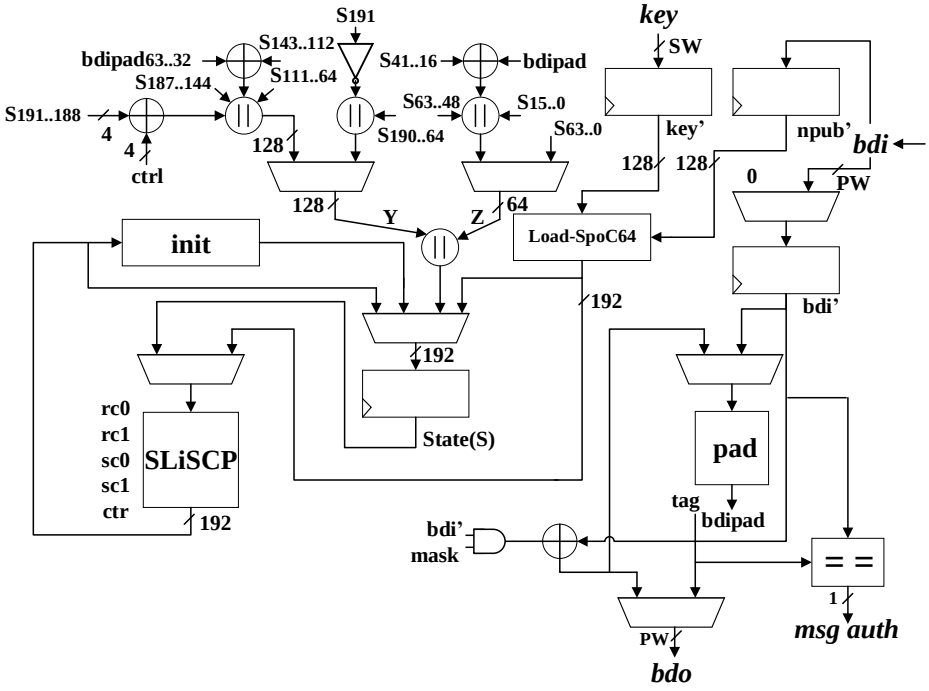


**Fig. 5.** Block diagram for SpoC-64.

consisting of many $AD$ and $PT$ blocks, where as there are always exactly two TBC iterations in each AEAD encrypt or decrypt. A long-term secret key is used only twice per encryption or decryption, which means that side-channel protections can be used in the only primitive which interacts with key (i.e., the TBC), and can be spared in the more expensive permutation.

**Algorithm 1:** Sequence for computing Shadow-512 diffusion based on 32-bit inputs $w$, $x$, $y$, and $z$ and outputs $a$, $b$, $c$, and $d$

1: $u \leftarrow w \oplus x$;
2: $v \leftarrow y \oplus z$;
3: $a \leftarrow x \oplus v$;
4: $b \leftarrow w \oplus v$;
5: $c \leftarrow u \oplus z$;
6: $d \leftarrow u \oplus y$;

We implement the authors' primary recommendation, Spook$[128; 512; su]$, with parameters block size $n = 128$, rate $r = 256$, capacity $c = 256$, tag size $\tau = 128$, and state size $b = 512$. The "su" denotes "single user;" the specification also calls for "multiple user" (mu) instantiations, which are not addressed in this research. We also used the recommended parameters for the TBC, consisting of 6 steps of 2 rounds each, and 12 rounds of Shadow-512 per permutation. Since

the rate determines the amount of plaintext converted to ciphertext in every block, we use an external block size of 256 bits for generation of test vectors and computation of throughput. There are $m$ blocks of plaintext (or message) $M$, and $a$ blocks of associated data $A$. There is a $\tau$-bit nonce $N$ (128 bits), a $key = K \parallel P$, which consists long term secret $K$ (128 bits), and public tweak $P$. Following the software reference implementation, $P = 0$ in this hardware implementation.

In TLS cipher constructions, linear transformations are computed using linear L-Boxes consisting of rotations and XORs, and non-linear S-Boxes. The L-Box is an interleaved transformation applying jointly to pairs of 32-bit words, i.e., half of the 128-bit state is processed in each L-Box. L-Boxes can be implemented using look-up tables or by arithmetic calculations. We follow the author's formula for L-Box calculations described in Alg. 2.

**Algorithm 2:** Sequence for computing L-Box based on 32-bit inputs $w$, $x$, $y$, and $z$ and intermediate variables or outputs $a$, $b$, $c$, and $d$, where $L^i$ denotes left rotation by $i$ bits.

1: $a \leftarrow x \oplus L^{12}(x)$;
2: $b \leftarrow y \oplus L^{12}(y)$;
3: $a \leftarrow a \oplus L^{3}(a)$;
4: $b \leftarrow b \oplus L^{3}(b)$;
5: $a \leftarrow a \oplus L^{17}(x)$;
6: $b \leftarrow b \oplus L^{17}(y)$;
7: $c \leftarrow a \oplus L^{31}(a)$;
8: $d \leftarrow b \oplus L^{31}(b)$;
9: $a \leftarrow a \oplus L^{26}(d)$;
10: $b \leftarrow b \oplus L^{25}(c)$;
11: $a \leftarrow a \oplus L^{15}(c)$;
12: $b \leftarrow b \oplus L^{15}(d)$;

The S-Boxes are a variant of the 4-bit S-Box used in the Skinny block cipher [10], and are implemented with look-up tables in this research.

The duplex sponge-based computational flow for authenticated encryption is shown in Fig. 6. Authenticated decryption is not depicted, but is similar in nature. The TBC is used twice – during initialization and tag generation. Initialization consists of loading the upper 256 bits of the state variable with $P \parallel 0^* \parallel N \parallel 0^*$, and computing $B = E_K^P(N)$, where $N$ is a 128-bit nonce. Upon TBC completion, the lower 256 bits of the state are loaded with $0 \parallel B$. *Tag* is formed as $Z = E_K^{V||1}(U)$, where $U$ is the upper 128 bits of state after the last permutation, and $V$ is the next highest 127 bits of state. During authenticated decryption, the supplied tag $Z'$ must be decrypted as $U' = D_K^{V'||1}(Z')$, and compared to $U$ (i.e., $U == U'$) for tag verification. Inverse L-Boxes $LBox^{-1}$ and S-Boxes $SBox^{-1}$ are required for decryption.

In between the initial and final TBC operations, the Shadow-512 permutation is computed once to initialize the state, and once following the processing of each block of $A$ or $M$. $10^*$ padding *pad* is applied to each final partial block of $A$, but padding is not directly applied to a final partial block of $M$. In this case, the resulting upper 256 bits of state are loaded as $C_{m-1} \parallel \{State_{512-|C_{m-1}|-1..256}\} \oplus 01 \parallel 0^*$. State truncation is performed by remembering the number of valid bytes loaded in the last block of plaintext (provided by `bdi_valid_bytes`) and by applying variable masks to $C_{m-1}$ and $\{State_{512-|C_{m-1}|-1..256}\} \oplus 01 \parallel 0^*$.

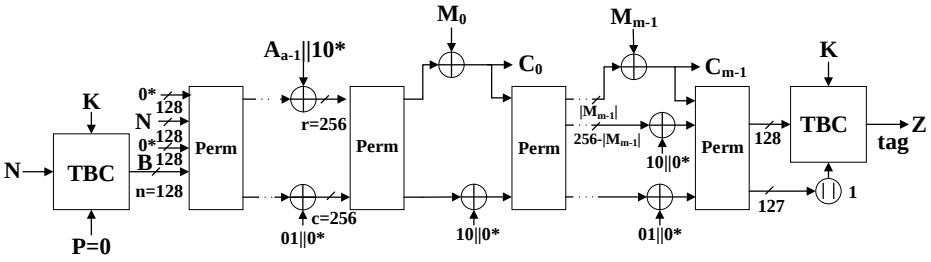


**Fig. 6.** Spook duplex sponge construction.

Domain separation between blocks of $A$ and $M$ is accomplished by $State_{255..0} \oplus \{01, 10, 11\} \parallel 0^*$, where the two-bit combination depends on whether the block is processing $A$ or $M$, and whether or not the last block is partial or full.

**Implementation** Since the authors state that "the need for two primitives implies a larger cost in hardware," we expect a more challenging task of implementing and integrating multiple primitives in one design. The authors also state that the use of the same S-box and L-box in Clyde-128 and Shadow-512 should allow resource sharing. Therefore, we base our design strategy on an attempt to reuse components such as L-Boxes, S-Boxes, and internal state registers, and construct the equivalent of TBC or permutation calls using an Arithmetic Logic Unit (ALU)-like microarchitecture approach.

We use a basic-iterative architecture with reference to the TBC. This means that one round of TBC (encryption or decryption) will execute in one clock cycle, including one set of 128-bit L-Boxes, 128-bit S-Boxes, and round constants. The tweakey is updated at the end of each step, or every other round. This results in 12 clock cycles per TBC.

However, our target implementation is not strictly basic-iterative with respect to the permutation, since we instantiate only 128-bit L-Boxes and S-Boxes, but must call each module 4 times across a 512-bit state. With 6 steps and 2 rounds per step, plus the diffusion, we have: Round A = 4 × 128 bit S-Boxes + 4 × 128 bit L-Boxes + 4 round count = 12 clock cycles; Round B = 4 × 128 bit S-Boxes + 4 round count = 8 clock cycles; and Diffusion = 4 × 128 rotations and XORs = 4 clock cycles, for a total of 144 clock cycles per permutation. The permutation

can be reduced to 12 or even fewer clock cycles, but at much greater hardware cost.

We first construct fully-functional but individual instances of TBC and Shadow-512 permutation. Using a wrapper to account for limited output pins on our target FPGA, we note implementation results in Xilinx Vivado. Our 144-cycle permutation requires 6379 LUTs (1753 slices), while the TBC requires 1483 LUTs (518 slices), for a total of 7862 LUTs (2271 slices). We make two observations:

1. With nearly 8000 LUTs required for the two primitives, we should try to reuse components between TBC and Shadow-512, and
2. Given that the 128-bit L-Boxes and S-Boxes require only 320 and 128 LUTs, respectively, the majority of the resources of these two primitives are consumed by non-shareable functions, such as control, routing structures, counters, tweakey generation, etc.

There are other hardware challenges in this cipher. The need for truncation to clear unused blocks of ciphertext prior to output, i.e., ensuring $|C| = |M|$, creates additional overhead in hardware, including generation of masks (e.g., `mask` and $C^*$). The `bdo_valid_bytes` feature in the LW Development package, although designed for ciphertext expansion modes, could have assisted in truncating output strings, but is not implemented in the current development package. In any case, internal padding and state truncation is required during processing of the last partial block of plaintext, as shown in Fig. 6.

Our implementation is shown in Fig. 7. All bus widths are 128 bits, unless indicated. Terminology in addition to that explained above and in Subsection 2.3 include $t'$ (tweak register), $tk$ (tweakey), $r0$, $r1$, $r2$ and $r3$ (128-bit state registers), and $mask'$ (register for $mask$).

### 3.3 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 $|T| = 128$ 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 like Simple [27] and 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 words and the key state divides into eight 16-bit segments. In SubCells, thirty two 4-bit bitslice Sboxes are applied to every nibble of the state. Then, a 32-bit permutation is applied to every word of the state. In AddRoundKey, the round key is XORed to the second and third words of the state, and a round

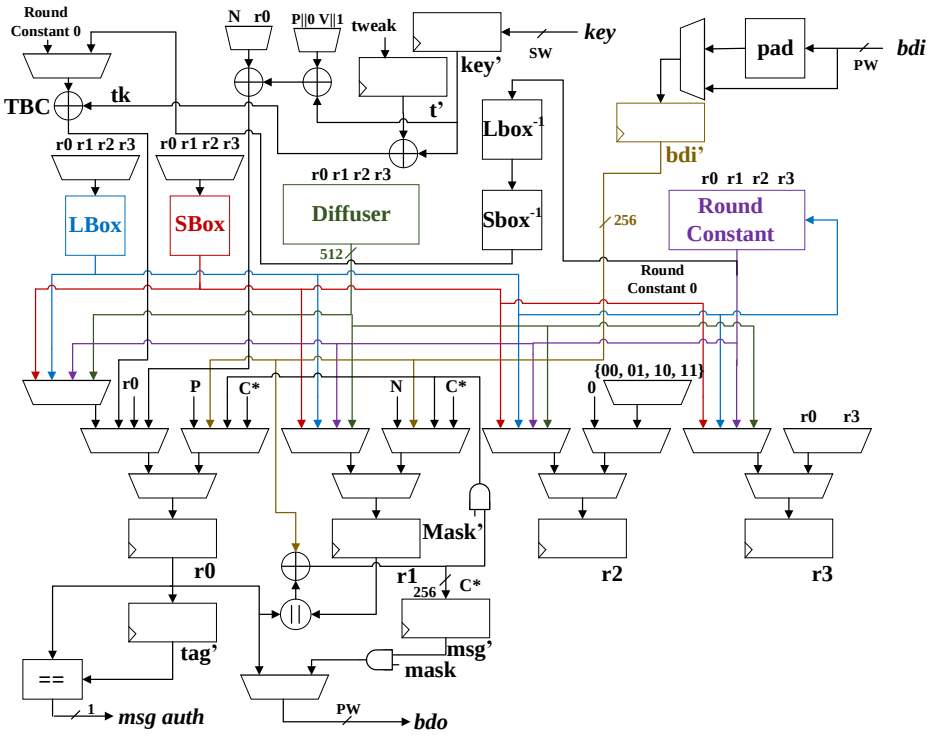


**Fig. 7.** Block diagram for Spook[128;512;su]. Bus widths are 128 bits unless indicated.

constant is added to the last word of the state. The round constants are generated by a 6-bit LFSR. Since the addition of the round key is done over only half of the state and the key scheduling is merely a bit permutation, and as mentioned in [9], GIFT-128 has a low footprint which makes it a good choice for lightweight applications [8]. Fig. 8 shows the GIFT round function.

In Fig. 9, 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 64 MSBs of the first $E_K$ output $L$ are considered as the delta state. Except for the last block of $AD$ and $M$, the delta state is multiplied by 2 in GF($2^{64}$) for every block of $AD$ and $M$. For the last block of $AD$ or $M$, 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].

**Implementation** A basic-iterative (i.e., round-based) 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 40 rounds, thus it requires 40 clock cycles to process a block of the input data. However, for processing $AD$ and $M$, we need additional clock cycles due to the delta state. As presented in [8], the required clock cycles for

delta state is 4. In this work, we used the same 4 clock cycles for processing an $AD$ block and 2 clock cycles in our finite state machine for processing the message blocks. The reason that we reduced the clock cycles for $M$ is that the exponent in $3^{j-i}$ does not exceed 2 for an $M$ 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 cycles that GIFT needs to process one block of data.

Similar to SpoC and Spook, the $10^*$ padding is applied to the last partial block of $AD$ and message. During the decryption, we need additional padding for the plaintext before applying it to the feedback. We used the `bdi_size` to accomplish the padding function. Since modules in the CAESAR Hardware Developer's Package uses 32 bits of `bdi` and `bdo_data`, we used a counter to track the number of valid bytes that the last block of $M$ 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. 10.

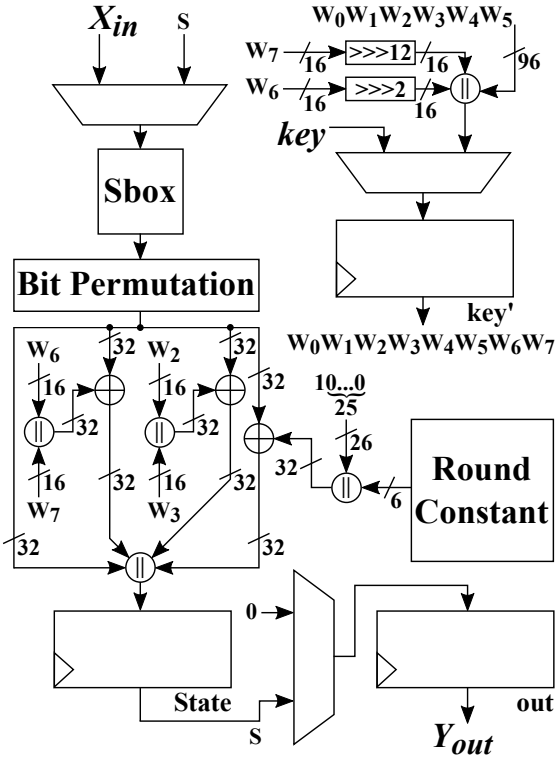


**Fig. 8.** Block diagram for GIFT-128 round function. Bus widths are 128 bits unless indicated.

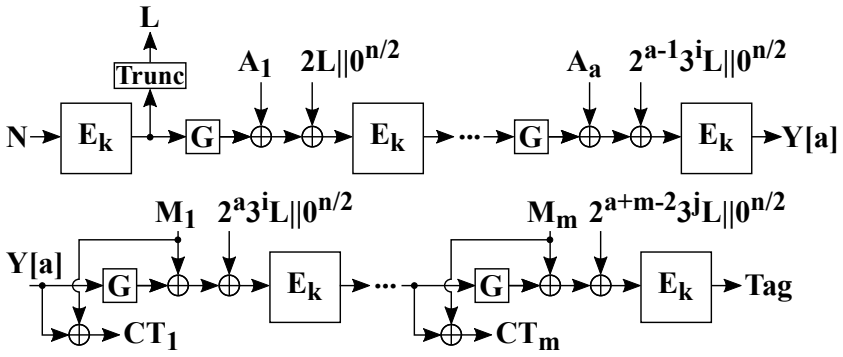


**Fig. 9.** GIFT-COFB encryption construction.

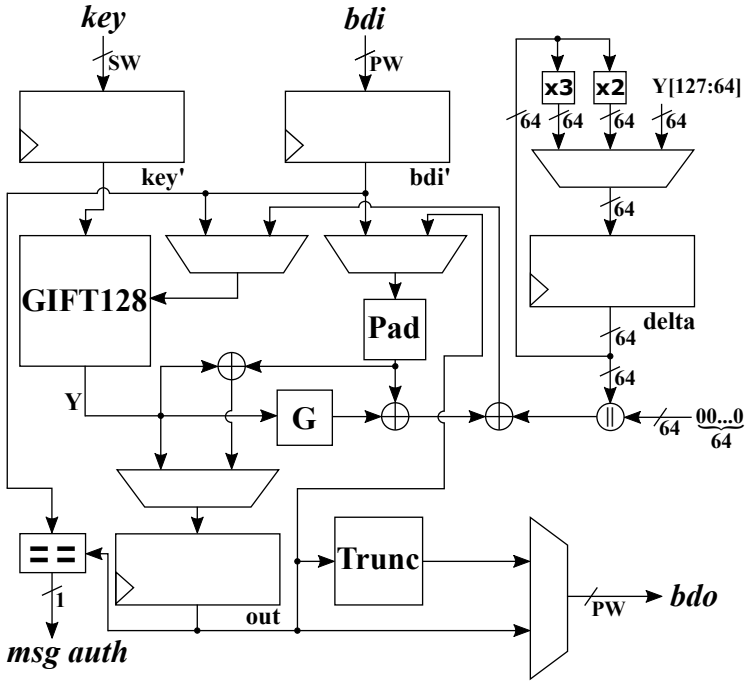


**Fig. 10.** Block diagram for GIFT-COFB. Bus widths are 128 bits unless indicated.

### 3.4 Summary

Characteristics of implemented ciphers are summarized in Table 2. All implemented ciphers have nonce and key size of 128 bits.

Latency and throughput formulas for the implemented ciphers are shown in Table 3. Clock cycles for AEAD encryption and decryption operations are shown as $\alpha + \beta \times A + \gamma \times M$, where $\alpha$ represents the sum of any initiation

**Table 2.** Characteristics of implemented ciphers.

| Cipher | block (bits) | steps | rounds | steps×rounds | state (bits) | rate (bits) | capacity (bits) | tag (bits) |
|---|---|---|---|---|---|---|---|---|
| SpoC-64 | 64 | 18 | 6 | 108 | 192 | 64 | 128 | 64 |
| Spook | 256 | 6 | 2 | 12 | 512 | 256 | 256 | 128 |
| GIFT-COFB | 128 | | | 40 | 128 | | | 128 |

and tag generation cycles, $\beta$ is the number of cycles to process 1 block of $AD$ ($A$), and $\gamma$ is the number of cycles to process 1 block of $PT$ ($M$). Latency is the number of clock cycles required to process 1 block of $PT$ from start to finish. Throughput (TP) is computed as maximum achieved clock frequency fclk × (bits/block)/(cycles/block).

**Table 3.** Latency and throughput formulas for selected ciphers.

| | SpoC | Spook | GIFT-COFB |
|---|---|---|---|
| Encrypt | $219 + 109 \times A + 111 \times M$ | $169 + 145 \times A + 145 \times M$ | $112 + 50 \times A + 53 \times M$ |
| Decrypt | $219 + 109 \times A + 111 \times M$ | $169 + 145 \times A + 145 \times M$ | $112 + 50 \times A + 53 \times M$ |
| Latency | 330 | 314 | 165 |
| TP | fclk × 64/111 | fclk × 256/145 | fclk × 128/53 |

## 4 Results

FPGA implementations in this research are developed in Verilog or VHDL using RTL design methodology. They are compliant with the CAESAR HW API, and include modules in the CAESAR HW Developer's Package (v2.0). Results are implemented in Xilinx Vivado 2018.3 for the Xilinx Artix-7 FPGA (xc7a100tcsg324-3), and optimized for TPA ratio using the Minerva Automated Hardware Optimizaton Tool, introduced at [20] and available for download at [1].

Our implementations are also verified in actual hardware (xc7a100tftg256-3) using the FOBOS [16] and representative test vectors generated by `aeadtvgen` in the Developer's Package. As discussed in [41], 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.

Post-optimization results are shown in Table 4. Additionally, cipher implementations are available for inspection at [40]. As a disclaimer, our implementations are baseline representations of select 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.

Results indicate that SpoC has the highest maximum frequency of 265 MHz, i.e., lowest sum of logic and routing delays, followed by GIFT-COFB (172 MHz)

**Table 4.** Benchmarking results on Artix-7 FPGA.

| Cipher | SpoC | Spook | GIFT-COFB |
|---|---|---|---|
| Max Freq (fclk)(MHz) | 265 | 141 | 172 |
| Bits/Block | 64 | 256 | 128 |
| Cycles/Block | 111 | 145 | 53 |
| Throughput (Mbps) | 152.8 | 248.9 | 415.4 |
| LUTs | 1344 | 7082 | 2695 |
| Slices | 410 | 1901 | 1090 |
| Registers | 745 | 1805 | 1163 |
| TPA (Mbps/LUT) | 0.114 | 0.035 | 0.154 |

and Spook (141 MHz). In terms of area in FPGA LUTs, SpoC is the smallest with 1344 LUTs, followed by GIFT-COFB with 2695 LUTs, and Spook with 7082 LUTs. GIFT-COFB has the highest TP at 415.4 Mbps, followed by Spook (248.9 Mbps) and SpoC (152.8 Mbps). In terms of TPA ratio, GIFT-COFB is the highest at 0.154 Mbps/LUT, followed by SpoC at 0.114 Mbps/LUT, and Spook at 0.035 Mbps/LUT.

While improvements in any one of more of the above metrics are possible, the fact that there are significant differences in an optimization metric such as TPA ratio show that results of hardware implementations of candidates should be considered as early as possible in any cryptographic contest.

## 5 Analysis

### 5.1 Comparison with selected previous authenticated cipher implementations

Previous hardware implementations during CAESAR and those provided as part of NIST LWC submissions provide some basis for comparison with cipher implementations in this research. However, most CAESAR implementations, even those that were compliant with the CAESAR HW API, used an earlier version of the CAESAR HW Developer's Package designed for High Speed (HS) implementations. The HS package included functionality not used by many ciphers, and exacted a larger toll on area overhead. The Lightweight (LW) Developer's Package only appeared at the end of 2017, and thus there are fewer available examples. Some CAESAR API-compliant examples from [42, 21], implemented using the LW Developer's Package, are included in Table 5 for purpose of comparison.

A full-scale comparison with NIST LWC candidate author implementations is premature, since authors reported results for implementations not compliant with the CAESAR API, and using a variety of FPGA platforms. 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 5. 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 (0.017 to 0.088) for the CAESAR candidates, all round 3 contenders or better, is somewhat analogous to the range of TPA ratios for our implementations (0.035 to 0.154). In contrast, the range of TPA ratios of sampled NIST LWC candidates (0.547 to 0.757) is noticably higher. A judgement as to whether or not these implementations are "better" than either our implementations, or previous CAESAR implementations, is premature, since no uniform standards have been established for benchmarking of hardware implementations in the NIST LWC Standardization Process. For instance, an implementation "compliant with the CAESAR API" is required to include hardware necessary for input and output AEAD data in specified protocol, and must realize "corner cases" (e.g., null blocks, partial blocks, padding, truncating, etc.) which often involve significant resources.

**Table 5.** Comparison with CAESAR lightweight and NIST LWC candidates. The units of Freq, Area, TP, and TPA are MHz, LUT, Mbps, and Mbps/LUT, respectively.

| Cipher | Type | FPGA | Freq | Area | TP | TPA | Ref |
|---|---|---|---|---|---|---|---|
| CAESAR | | | | | | | |
| Ascon-128 | Sponge | Spartan-6 | 216.0 | 684 | 60.1 | 0.088 | [42] |
| CLOC-AES | Block | Spartan-6 | 101.9 | 1604 | 68.7 | 0.043 | [21] |
| SILC-AES | Block | Spartan-6 | 115.1 | 872 | 15.1 | 0.017 | [21] |
| NIST LWC (AEAD) | | | | | | | |
| ESTATE | Block | Virtex-7 | 580.1 | 1413 | 928.3 | 0.657 | [17] |
| SAEAES | Block | Virtex-7 | 145.9 | 348 | 263.3 | 0.757 | [34] |
| Oribatida | Sponge | Virtex-7 | 554.2 | 940 | 514 | 0.547 | [14] |

### 5.2 Observations

The use of two primitives in Spook, i.e., the Clyde-128 TBC and the Shadow-512 primitive, has a detrimental effect on the area of our Spook FPGA implementation. While we have employed a strategy to reuse shared components among the primitives, such as S-Box and L-Box, the increased use of multiplexers, control structures, and resulting routing delays necessary to tie together all components likely outweighs any advantages gained over the use of separate TBC and Shadow permutation primitives in a "black box" approach. The fact that the TBC requires encryption and decryption adds again to implementation complexity.

The necessity of applying one-zero ($10^*$) padding to input words of $AD$ and $PT$ is a known factor in increasing complexity. However, cipher algorithms which can make padding application features as similar as possible for both $AD$ and $PT$ can reduce hardware complexity. While this is the case in SpoC and GIFT-COFB, it is not the case in Spook, since the padding in Spook occurs on words of state vice words of input when processing $PT$.

Additionally, truncation is also a well-known feature in cryptographic algorithms, where $|CT|$ should equal $|PT|$ in order to assure invertibility and not leak information. While truncation in software looks innocuous enough, it often catches cryptographers by surprise in hardware. We employed a non-resource intensive method in SpoC, to remove one unwanted byte at a time from ciphertext output in a serial fashion. However, in Spook, truncated ciphertext is required to be passed to the state input for subsequent permutations, in addition to being routed directly to cipher output, which increases implementation complexity.

While the above issues primarily affect area (e.g., LUTs or slices), they can also affect performance, as resulting routing delay in FPGA necessary to connect more basic elements decreases the maximum achievable frequency, even without a corresponding increase in logic delay.

## 6 Conclusion

We provided an accelerated look at fully-functional FPGA implementations of selected NIST LWC Standardization Process round 1 candidates. Candidates examined in this research, SpoC, Spook, and GIFT-COFB, are representative of the many of the 56 NIST LWC round 1 accepted submissions. Implementations are fully-compliant with the existing CAESAR Hardware API for Authenticated Ciphers, use the associated Hardware Developer's Package, are optimized using the Minerva Automated Hardware Optimization Tool, and are verified to operate in actual hardware using the FOBOS test bench.

Our results show that GIFT-COFB has the highest TPA ratio (4.4 × more than Spook), followed by SpoC (3.3 × more than Spook). The magnitude of differences in TPA ratios supports the rationale for gaining information on hardware implementations as early as possible in any cryptographic contest. The TPA ratio results of the implemented ciphers are similar to results reported for CAESAR HW API compliant late-round CAESAR candidates, however, have TPA ratios which are significantly less than TPA ratios reported for a select group of NIST LWC submission author implementations of ciphers of similar construction. However, 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.

Finally, we show that implementation complexities resulting from the need to integrate two cryptographic primitives (e.g., a block cipher and sponge permutation) into one authenticated cipher, as well as padding and truncation strategies, can affect area and performance of resulting implementations, and should be considered by algorithm designers.

## 7 Future work

This was a first-look of NIST LWC FPGA hardware implementations delivered in a compressed time-frame. Future work will include a larger number of implemented

candidates, compliant with a speculated future NIST LWC API, developed with a corresponding Developer's Package, and integrated with optional hash capability. Future research will also experiment with lighter-weight architectures, and include verifiable countermeasures against side-channel attacks and fault analysis.

## 8 Acknowledgements

The authors would like to thank Kris Gaj, Jens-Peter Kaps, and Abubakr Abdulgadir at George Mason University, and Michael Tempelmeier at Technical University Munich, for suggestions and technical support.

## 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-1-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-1-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-1-Candidates
5. ARM: AMBA Specifications, http://www.arm.com/products/system-ip/amba-specifications.php
6. Banik, S., Bogdanov, A., Isobe, T., Shibutani, K., Hiwatari, H., Akishita, T., Regazzoni, F.: Midori: A block cipher for low energy. In: Iwata, T., Cheon, J.H. (eds.) Advances in Cryptology – ASIACRYPT 2015. pp. 411–436. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
7. Banik, S., Bogdanov, A., Peyrin, T., Sasaki, Y., Sim, S.M., Tischhauser, E., Todo, Y.: SUNDAE-GIFT: An Authenticated Encryption and Hash Algorithm Submission to the NIST LWC Competition (Mar 2019), https://csrc.nist.gov/Projects/Lightweight-Cryptography/Round-1-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 Encryption and Hash Algorithm Submission to the NIST LWC Competition (Mar 2019), https://csrc.nist.gov/Projects/Lightweight-Cryptography/Round-1-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) (2017)
10. 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)

11. Bellizia, D., Berti, F., Bronchain, O., Cassiers, G., Duval, S., Guo, C., Leander, G., Leurent, G., Levi, I., Momin, C., Pereira, O., Peters, T., Standaert, F.X., Wiemer, F.: Spook: Sponge-Based Leakage-Resilient Authenticated Encryption with a Masked Tweakable Block Cipher (Mar 2019), https://csrc.nist.gov/Projects/Lightweight-Cryptography/Round-1-Candidates
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., Lpez, C.M., Nandi, M.: The Oribatida Family of Lightweight Authenticated Encryption Schemes (Mar 2019), https://csrc.nist.gov/Projects/Lightweight-Cryptography/Round-1-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-1-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. Elbirt, A., Yip, W., Chetwynd, B., Paar, C.: An FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists (10 2002)
20. Farahmand, F., Ferozpuri, A., Diehl, W., Gaj, K.: Minerva: Automated Hardware Optimization Tool. In: 2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)
21. 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
22. Finjan Cybersecurity: IoT DoS Attacks – How Hacked IoT Devices Can Lead To Massive Denial of Service Attacks (Aug 2018), https://blog.finjan.com/iot-dos-attacks/
23. Gaj, K., Chodowiec, P.: Comparison of the Hardware Performance of the AES Candidates Using Reconfigurable Hardware. pp. 40–54 (01 2000)
24. George Mason University: Development Package for the CAESAR Hardware API, v2.0 (Dec 2017), https://cryptography.gmu.edu/athena/index.php?id=CAESAR
25. Grosso, V., Leuren, G., Standaert, F.X., Varici, K., Durvaux, F., Gaspar, L., Kerckhof, S.: Scream  iscream side-channel resistant authenticated encryption with masking (2014)
26. Grosso, V., Leurent, G., Standaert, F.X., Varıcı, K.: Ls-designs: Bitslice encryption for efficient masked software implementations. In: Cid, C., Rechberger, C. (eds.) Fast Software Encryption. pp. 18–37. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)

27. Gueron, S., Lindell, Y.: Simple: An Authenticated Encryption and Hash Algorithm Submission to the NIST LWC Competition (Mar 2019), https://csrc.nist.gov/Projects/Lightweight-Cryptography/Round-1-Candidates
28. Homsirikamol, E., Diehl, W., Ferozpuri, A., Farahmand, F., Gaj, K.: Implementers Guide to the CAESAR Hardware API v2.0 (Dec 2017), https://cryptography.gmu.edu/athena/CAESAR_HW_API/ CAESAR_HW_Implementers_Guide_v2.0.pdf
29. Homsirikamol, E., Diehl, W., Ferozpuri, A., Farahmand, F., Yalla, P., Kaps, J., Gaj, K.: Addendum to the CAESAR Hardware API v1.0 (Jun 2016), https://cryptography.gmu.edu/athena/CAESAR_HW_API/ CAESAR_HW_API_v1.0_Addendum.pdf
30. 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
31. Homsirikamol, E., Diehl, W., Ferozpuri, A., Farahmand, F., Yalla, P., Kaps, J.P., Gaj, K.: CAESAR Hardware API. Cryptology ePrint Archive, Report 2015/669 (2016)
32. 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)
33. Lewis, D.: The DDoS Attack Against Dyn One Year Later (Oct 2017), https://www.forbes.com/sites/davelewis/2017/10/23/the-ddos-attack-against-dyn-one-year-later/2a2315031ae9
34. Naito, Y., Matsui, M., Sakai, Y., Suzuki, D., Sakiyama, K., Sugawara, T.: SAEAES (Feb 2019), https://csrc.nist.gov/Projects/Lightweight-Cryptography/Round-1-Candidates
35. 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)
36. 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
37. National Institute of Standards and Technology: Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process (Aug 2018), https://csrc.nist.gov/projects/lightweightcryptography
38. Rezvani, B., Diehl, W.: Detailed Characteristics of NIST Lightweight Cryptography Project Round 1 Submissions (v2) (Jul 2019), https://rijndael.ece.vt.edu/wdiehl/
39. Rogaway, P.: Authenticated-encryption with associated-data. p. 98 (01 2002). https://doi.org/10.1145/586123.586125
40. SAL: NIST Lightweight Cryptography Project (Jul 2019), https://rijndael.ece.vt.edu/wdiehl/
41. 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
42. Yalla, P., Kaps, J.P.: Evaluation of the CAESAR Hardware API for Lightweight Implementations. In: International Conference on Reconfigurable Hardware (ReConFig 2017). pp. 1–6 (Dec 2017)