Hardware API for Lightweight Cryptography*

Jens-Peter Kaps[1], William Diehl[2], Michael Tempelmeier[3],
Ekawat Homsirikamol[4], and Kris Gaj[1]

[1] Cryptographic Engineering Research Group
George Mason University, Fairfax, Virginia 22030, USA
email: {jkaps, kgaj}@gmu.edu
[2] Signatures Analysis Lab
Virginia Tech, Blacksburg, Virginia 24061, USA
email: wdiehl@vt.edu
[3] Lehrstuhl für Sicherheit in der Informationstechnik
Technische Universität München, 80333 München, Germany
email: michael.tempelmeier@tum.de
[4] Independent Researcher
email: ekawat@gmail.com

Abstract. In this paper, we define the Lightweight Cryptography (LWC) Hardware Application Programming Interface (API) for the implementations of lightweight authenticated ciphers, hash functions, and cores combining both functionalities. In particular, our API is intended to meet the requirements of all candidates submitted to the NIST Lightweight Cryptography standardization process. The major parts of our specification include minimum compliance criteria, interface, communication protocol, and timing characteristics supported by the LWC core. All of these criteria have been defined with the goals of guaranteeing (a) compatibility among implementations of the same algorithm by different designers, and (b) fair benchmarking of hardware cores implementing an authenticated cipher, a hash function, or both functionalities. Since 2016, a similar API has been successfully used to implement and benchmark all candidates qualified to Rounds 2 and 3 of the CAESAR competition for authenticated ciphers.

1 Introduction

The main reasons for defining a common API for all hardware implementations of candidates submitted to the NIST Lightweight Cryptography standardization project [1] are:

– Fairness of benchmarking,

* This work is supported by the Department of Commerce (NIST) Grant no. 70NANB18H219

– Compatibility among implementations of the same algorithm by different designers, and
– Ease of creating the supporting development package, aimed at simplifying and speeding up the design process.

Among the major cryptographic competitions, the first attempt at defining a hardware API took place during the SHA-3 contest [2, 3]. In the area of high-speed implementations, all 14 Round 2 candidates, all 5 Round 3 candidates, and the previous standard SHA-2 were implemented using the proposed interface and communication protocol by the group from George Mason University (GMU) [2–4]. This interface and protocol were then extended to the case of lightweight applications and applied to the implementations of 13 Round 2 and 5 Round 3 SHA-3 candidates [5]. Alternative interfaces of hash function cores were proposed in [6, 7]. No specific interface was endorsed by NIST as a requirement for all implementations.

During the subsequent CAESAR contest (Competition for Authenticated Encryption: Security, Applicability, and Robustness), conducted in the period 2013-2019, all major decisions were made by the CAESAR Committee, composed of 18 renowned cryptographers, representing multiple institutions worldwide [8].

The first version of the proposed hardware API for CAESAR was reported in [9]. This version was later substantially revised, endorsed by the CAESAR Committee in May 2016, and published as a Cryptology ePrint Archive in June 2016 [10]. A relatively minor addendum was proposed in the same month, and endorsed by the CAESAR Committee in November 2016 [11].

The commonly accepted CAESAR Hardware API provided the foundation for the GMU Development Package, which was released in May and June 2016 [12]. This package included in particular:

a. VHDL code of a generic PreProcessor, PostProcessor, and CMD FIFO, common for all Round 2 and Round 3 CAESAR Candidates (except Keyak), as well as AES-GCM,
b. Universal testbench common for all API-compliant designs (AEAD_TB),
c. Python app used to automatically generate test vectors (aeadtvgen), and
d. Reference implementations of Dummy authenticated ciphers (dummyN).

This package was accompanied by the Implementer's Guide to Hardware Implementations Compliant with the CAESAR Hardware API, v1.0, published at the same time [13]. A few relatively minor weaknesses of this version of the package, discovered when performing experimental testing using general-purpose prototyping boards, were reported in [14].

In December 2017, a substantially revised version of the Development Package (v.2.0) and the corresponding Implementer's Guide were published by the GMU Benchmarking Team [15, 16]. The main revisions included

– Support for the development of lightweight implementations of authenticated ciphers,
– Improved support for the development of high-speed implementations of authenticated ciphers, and

– Improved support for experimental testing using FPGA boards, in applications with intermittent availability of input sources and output destinations.

It should be stressed that at no point was the use of the Development Package required for compliance with the CAESAR Hardware API. To the contrary, [13] clearly stated that the implementations of authenticated ciphers compliant with the CAESAR Hardware API can also be developed without using any resources belonging to the package [12], by just following the specification [10] directly.

In spite of being non-mandatory and the lack of the official endorsement by the CAESAR Committee, the CAESAR Development Package played a major role in increasing the number of implementations developed during Round 2 of the CAESAR contest. Out of 43 implementations reported before the end of Round 2, 32 were fully compliant, and one partially compliant with the CAESAR Hardware API. All fully compliant implementations used the GMU Development Package. The fully and partially compliant implementations covered 28 out of 29 Round 2 submissions (all except Tiaoxin) [15, 17].

In Round 3, the submission of the hardware description language code (VHDL or Verilog) was made obligatory by the CAESAR Committee. As a result, the total number of designs reached 27 for 15 Round 3 candidates. Out of these 27 designs, 23 were fully compliant and 1 partially compliant with the CAESAR Hardware API [10].

Overall, publishing the CAESAR Hardware API, as well as its endorsement by the organizers of the contest, had a major influence on the fairness and the comprehensive nature of the hardware benchmarking during the CAESAR competition.

Several optimized lightweight implementations compliant with this API, and based on v.2.0 of the Development Package, were reported in [18]. In [19–22], several other implementations were enhanced with countermeasures against Differential Power Analysis. In order to facilitate this enhancement, an additional Random Data Input (RDI) port was added to the CAESAR Hardware API.

In this paper, we use the CAESAR Hardware API [10] and its Addendum [11] as a basis for the Lightweight Cryptography (LWC) Hardware API. The detailed differences are summarized in Section 7. The rest of the document is organized as follows. In Section 2, the minimum compliance criteria are defined. In Sections 3, 4, and 5, the proposed interface, communication protocol, and timing characteristics of the LWC core, are described, respectively.

Readers familiar with the CAESAR Hardware API [10, 11], may consider going directly to Section 7, Differences Compared to the CAESAR Hardware API, and reviewing other sections only if needed.

2 Minimum Compliance Criteria

The recommended minimum compliance criteria are listed below. Each criterion is listed with a **heading**, then described in *italics* and followed by a short explanation, including the justification and (optionally) a list of alternatives.

2.1 Encryption/Decryption

Rule: *Authenticated encryption and decryption should be implemented within one core, but only one of these two operations can be executed at a time (half-duplex).*

Justification: This feature demonstrates an algorithm's ability to use shared resources for encryption and decryption.

Alternatives (not recommended):

a) separate cores for encryption and decryption (simplex)
b) authenticated encryption and decryption within one core, with both operations capable of running in parallel (full-duplex).

2.2 Hashing

Rule: *If hashing is supported by a given algorithm, then designers should develop two versions of the LWC core, capable of performing*

a) encryption, decryption, and hashing
b) encryption and decryption only.

Justification: Option a) should be implemented to demonstrate the capability of sharing resources between authenticated encryption and hashing. Option b) should be implemented to enable comparison with algorithms that do not support hashing.

2.3 Variants

Rule: *Only a variant indicated in the algorithm specification as the primary recommendation has to be implemented. Other variants, if implemented, should be selectable at the time of synthesis. The implementation of multiple variants should not affect benchmarking results for any of them.*

Justification: Supporting multiple variants within the same core (with the capability of switching during runtime) may lead to substantial overheads compared to any particular variant, complicating ranking of algorithms.

An alternative (not recommended):

– implementing multiple variants within the same LWC core, with the capability of switching among them during runtime.

2.4 Key scheduling

Rule: *Key scheduling of authenticated ciphers should be fully implemented within the LWC core.*

Justification: This approach takes into account very different contributions of the key scheduling unit to the entire cipher hardware implementation area, which are specific for each algorithm.

An alternative (not recommended):

– generation of round keys outside of the cipher hardware implementation, e.g., in software.

2.5 Incomplete blocks

Rule: *The LWC core should properly handle incomplete blocks in associated data, plaintext, hash message, and ciphertext.*

Justification: Handling of incomplete blocks substantially increases the core area for multiple candidates, due to the large area required for variable shifts.

An alternative (not recommended):

– handling only associated data, plaintext, hash message, and ciphertexts composed of full blocks.

2.6 Padding

Rule: *Padding should be implemented in hardware.*

Justification: Padding cost, in terms of area, is algorithm dependent and not negligible. In some algorithms, padding in software may need to be reversed in hardware because the tag calculation uses an unpadded last block.

Alternatives (not recommended):

a) Padding in hardware, assuming that an unused portion of the last *block* is filled with zeros.
b) Padding in software, followed, if needed, by modifications of the last blocks in hardware.

2.7 Unused portions of the last block

Rule: *Any unused portions of the last block released to the output should be cleared (filled with zeros).*

Justification: Any portions of the last block that do not belong to the proper output, such as ciphertext or decrypted plaintext, can potentially leak the results of intermediate calculations involving the key, and thus also facilitate cryptanalysis.

An alternative (not recommended):

– potentially leaking some key-related data using unused portions of the last block or word released to the output.

2.8 Decrypted plaintext release

Rule: *The decrypted plaintext blocks should be released immediately, without waiting for the result of authentication.*

Justification: We assume that the delayed release of decrypted data, dependent on the result of authentication, will be handled by an external circuit, which is FIFO-based and similar for each candidate.

An alternative (not recommended):

– storing a decrypted plaintext internally, until the result of the verification is known.

Pros: More complete functionality.
Cons: Complicates the design and benchmarking. Makes the calculation of the output latency and throughput dependent on the output buffer size and implementation details (e.g., support for simultaneous reading and writing).

2.9 Empty AD/plaintext/ciphertext/hash message

Rule: *The core should support empty associated data, plaintext, hash message, ciphertext, and any meaningful combination thereof, unless the zero length of any of the aforementioned input parts or their combinations is explicitly excluded in the algorithm specification.*

Justification: Empty AD and empty plaintext could be used together with the public message number, Npub, for user authentication. Empty AD, plaintext, and hash message are supported by the specifications of the majority of authenticated ciphers and hash functions, as well as their software implementations.

Alternatives (not recommended):

a) not allowing empty associated data

b) not allowing empty plaintext/ciphertext
c) not allowing empty hash message
d) not allowing empty input.

2.10 Supported maximum size of AD, plaintext, ciphertext, and hash message

Rule: *For the purpose of benchmarking, the LWC core should support at least the following maximum sizes of associated data, plaintext, and hash messages:*
For single-pass algorithms:
Sa) $2^{16}-1$ *: default; used for comparison with implementations of two-pass algorithms*
Sb) $2^{32}-1$ *: kept for compatibility with the CAESAR API; practical only for single-pass algorithms*
Sc) $2^{50}-1$ *: minimum limit established by NIST for algorithms submitted to the Lightweight Cryptography standardization process.*
For two-pass algorithms:
Ta) $2^{16}-1$ *: default; used for comparison with implementations of single-pass algorithms*
Tb) $2^{11}-1$ *: kept for compatibility with the CAESAR API*
Tc) $2^{50}-1$ *: minimum limit established by NIST for algorithms submitted to the Lightweight Cryptography standardization process.*

The core should also support the corresponding ciphertext sizes. However, the size limit $2^{16}-1$ *should be treated as a default, and the implementers should do their best to eliminate (or at least minimize to negligible) the influence of the remaining size limits on the*

1. *maximum clock frequency*
2. *total number of clock cycles for short messages*
3. *throughput for long messages.*

Justification: *If these conditions are met, the timing performance results could be assumed to be identical for all three cases associated with a given type of cipher (and thus only one ranking would need to be generated and analyzed). Any experimental verification of performance could be then conducted for just a single size limit of* $2^{16}-1$ *bytes (within reach of two-pass algorithms, assuming the storage of all intermediate results on chip). At the same time, the resource utilization of each LWC core would be reported using three different numbers, demonstrating the dependence of the*
a) total area in case of ASICs, and
b) amount of particular FPGA resources (e.g., LUTs, FFs, BRAMs, DSPs)
on the selected size limit.
The NIST Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process [1] define these maximum sizes as $2^{50}-1$ *for both individual AD/plaintext/ciphertext/hash message, as well the total amount of data processed using a given key. Still, we believe that for the purpose of*

a) operation as a part of most-commonly used applications, and b) hardware benchmarking, the implementations can be limited to supporting smaller, more realistic AD/plaintext/ciphertext/hash message sizes. In particular, in the majority of popular communication protocols, only sizes smaller than or equal to 1500 bytes (the maximum transmission unit (MTU) of Ethernet v2) have to be supported. Additionally, for two-pass algorithms, it would be unrealistic to store intermediate results of the size of $2^{50} - 1$ *bytes on chip, or even on the same board. Similarly, for single-pass algorithms, it would be infeasible to store this amount of decrypted plaintext on chip until the result of authentication becomes available. In both cases, factors beyond the definition of the API would influence the actual maximum speed of the core. Therefore, we suggest the submission of the three aforementioned variants of each core, selected at the synthesis time using a single generic or constant. The differences between the implementations with different maximum sizes are expected to concern only their area, rather than the maximum clock frequency, latency, or throughput.*

2.11 Fractions of bytes

Rule: *The LWC core should support only inputs composed of full bytes.*
Justification: The Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process [23] define an authenticated encryption with associated data (AEAD) algorithm as a function with four *byte-string* inputs and one *byte-string* output. Similarly, it defines a hash function as a function with one *byte-string* input and one *byte-string* output. Thus, none of the inputs or outputs can include any fractions of bytes. This constraint may not apply to other authenticated ciphers or hash functions, developed in the past. In particular, it does not apply to the members of the SHA-2 family of hash functions, which accept inputs with an arbitrary number of input *bits*. In case such functions are implemented in agreement with our LWC hardware API, only inputs composed of full bytes should be supported. Allowing inputs of arbitrary size in bits would substantially increase the area required for handling incomplete blocks.

An alternative (not recommended):

- the size of inputs expressed in *bits*.

2.12 Maximum number of independent streams of data processed in parallel

Rule: *The LWC core should process only one stream of data (i.e., a single independent input understood as composed of any subset of Npub, AD, plaintext, hash message, ciphertext, and tag, supported by the encryption, decryption, or hashing operation of a given algorithm) at a time (without an overlap). We refer to such a core as a single-stream implementation. The single-stream implementation may still take advantage of parallel processing for blocks belonging to the*

same input/stream.

Justification: In the multi-stream implementations:

- Area and power requirements are greater than in single-stream implementations.
- Throughput is limited only by the maximum circuit area.
- Multiple plaintexts/ciphertexts processed in parallel would require multiple public data input (PDI) and data outputs (DO) ports. See Section 3 for the detailed descriptions of these ports.

An alternative (not recommended):

- a multi-stream implementation that supports processing of multiple independent inputs/streams in parallel.

2.13 External memory

Rule:

Single-pass algorithms:	No
Two-pass algorithms:	Yes (but only for results of the first pass)

Justification: For single-pass algorithms, no external memory should be used. Two-pass algorithms can typically benefit from external memory, used to store intermediate values utilized as inputs to the second pass. An alternative is to provide an entire input for the second time to the data inputs of the LWC core. However, doing that is typically less efficient in terms of throughput. Additionally, providing the same input twice through the same port complicates the input circuit, e.g., by requiring two costly DMA transfers, or placing external memory and the associated control logic before the data input ports. The memory used to store intermediate results in two-pass algorithms can also be shared with the memory used to store outputs from authenticated decryption before the AD and ciphertext are fully authenticated. Since the latter memory is required for both single-pass and two-pass algorithms, and it is an external memory for single-pass algorithms, it would be unfair to treat the intermediate-data memory in two-pass algorithms as internal.

2.14 One clock domain

Rule: *The core should have only one clock input and one internal clock signal. The implementation should be able to operate at the maximum clock frequency determined by the critical path located entirely inside of the LWC core.*

Justification: Using a single clock domain simplifies static timing analysis, generation of post-place and route results, and optimization of FPGA tool options. Additionally, the internal datapath width is typically the same or smaller than

the input and output port widths. As a result, there is no advantage from reading inputs or writing outputs with higher clock frequency.

An alternative (not recommended):

– separate clocks for the input module, output module, and cipher.

2.15 Passing unchanged parts of the input to the output

Rule: *Parts of the data inputs that are not changed by encryption or decryption operations, respectively, should not be passed to the output. In particular, Npub and AD should not be a part of the output from either encryption or decryption. See Fig. 5.*

Justification: This assumption removes the need for any bypass FIFO necessary to pass any unchanged data to the output. Any formatting of output from encryption / decryption, for the purpose of transmission through the network or subsequent decryption / encryption, respectively, is assumed to be performed outside of the LWC core.

An alternative (not recommended):

– Passing unchanged parts of the input to the output.

Pros: More complete functionality.
Cons: The design time and area overhead for adding standard functionality that may be implemented in a coherent way outside of the authenticated cipher hardware implementation.

2.16 Permitted widths of external data ports (in bits)

Rule: *The permitted widths of the data buses for the Public Data Input (PDI), Secret Data Input (SDI), and Data Output (DO) ports are as follows:*
PDI and DO: w = 8, 16, or 32
SDI: sw = 8, 16, or 32.

See Section 3 and Fig. 1 for the exact meaning of PDI, SDI, DO.

Justification: 8-bit, 16-bit, and 32-bit processors are among the most popular processors used in embedded systems, especially in resource-constrained environments. An LWC core needs to be able to communicate with at least one of these processors. Hardware architectures of lightweight ciphers and hash functions often use the internal datapath width equal to 8, 16, or 32 bits. It is quite natural (although not required) for an external data bus width to match the internal datapath width. The permitted widths of external data buses also match those defined in the CAESAR Hardware API [10]. This feature makes all existing lightweight implementations of CAESAR Candidates compatible with the proposed hardware API, which provides many helpful reference points for the comparison of results, as well as many helpful open-source examples.

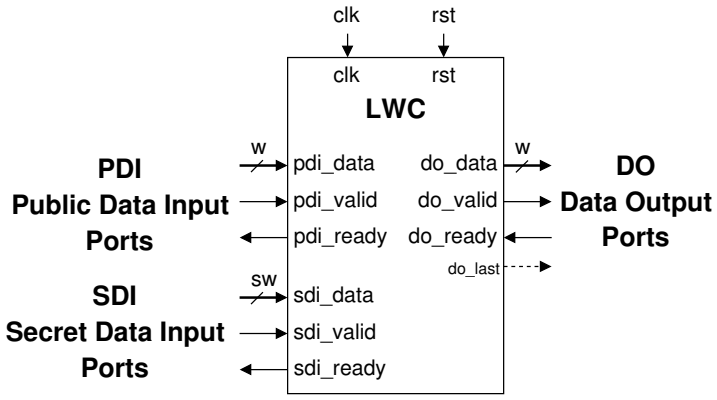


Fig. 1: LWC interface for single-pass algorithms

3 Interface

The proposed interface of the LWC core is shown for

- single-pass algorithms in Fig. 1,
- two-pass algorithms in Fig. 2.

This interface includes three major data buses for:

- Public Data Inputs (`PDI`)
- Secret Data Inputs (`SDI`), and
- Data Outputs (`DO`),

as well as the corresponding handshaking control signals, named *valid* and *ready*. The *valid* signal indicates that the data is ready at the source, and the *ready* signal indicates that the destination is ready to receive it. The signal *do_last* is an optional signal which simplifies the connection to an AXI4-Stream Slave (see below).

In the case of two-pass algorithms, the TWO-PASS FIFO Data Input and Output Ports are added.

The physical separation of Public Data Inputs (such as the plaintext, associated data, public message number, etc.) from Secret Data Inputs (such as the key) is dictated by the resistance against any potential attacks aimed at accepting public data, manipulated by an adversary, as a new key.

The handshaking signals are a subset of major signals used in the AXI4-Stream interface [24]. As a result, LWC can communicate directly with the AXI4-Stream Master through the Public Data Input, and with the AXI4-Stream Slave through the Data Output, as shown in Fig. 3. At the same time, LWC is also capable of communicating with much simpler external circuits, such as FIFOs, as shown in Fig. 4.

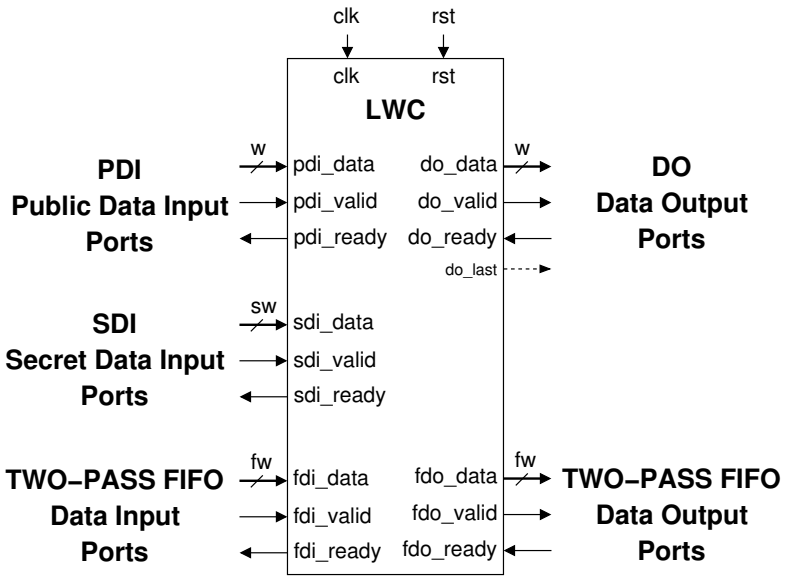


Fig. 2: LWC interface for two-pass algorithms

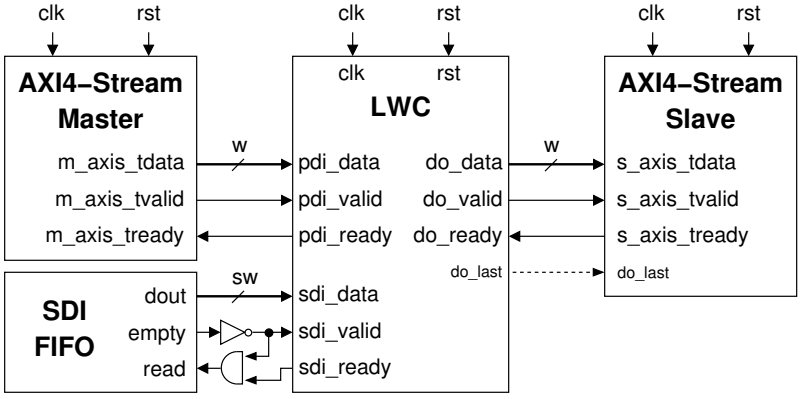


Fig. 3: Typical external circuits: AXI4-Stream IPs

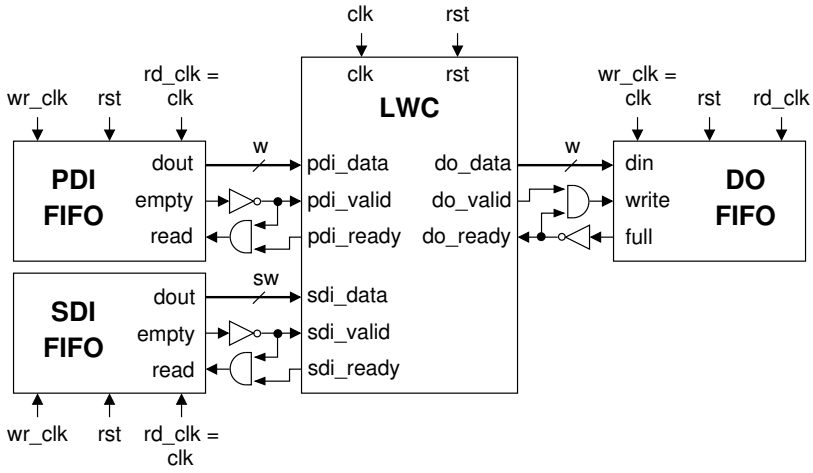


Fig. 4: Typical external circuits: FIFOs

In both cases, the Secret Data Input is connected to a FIFO, as the amount of data loaded to the LWC core using this input port does not justify the use of a separate AXI4-Stream Master, such as DMA.

An additional advantage of using FIFOs at all data ports is their potential role as suitable boundaries between two clock domains, one used for communication and one for computation. This role is facilitated by the use of separate read and write clocks, shown in Fig. 4 as `rd_clk` and `wr_clk`, accordingly. For better compatibility with the AXI communication interface, all FIFOs mentioned in our description are assumed to operate in the First-Word Fall-Through mode (as opposed to the standard mode).

The reset input can be either synchronous or asynchronous, and either active-high or active-low, depending on the conventions used in a given technology (e.g., FPGA vs. ASIC), as well as the personal preference of the designers.

The recommended interface of two-pass algorithms is shown in Fig. 2. Compared to the interface of single-pass algorithms, shown in Fig. 1, additional ports used for communication with the external Two-Pass FIFO have been added. The width of the data buses of these ports is defined by a constant, denoted in Fig. 2 as `fw`. The value of this constant can be selected freely by the designers, depending on the specific feature of each two-pass algorithm and its implementation.

In modern FPGAs, the Two-Pass FIFO will be implemented using block memories (such as BRAMs of Xilinx FPGAs and embedded memory blocks of Intel FPGAs). A FIFO with a capacity of 2^{16} bytes can be built using a relatively small percentage of the total size of on-chip block memories. Thus, the two-pass algorithms are not in any significant way disadvantaged compared to single-pass algorithms.

Additionally, even for single-pass algorithms, a NIST compliant implementation of authenticated decryption is expected to store the deciphered plaintexts

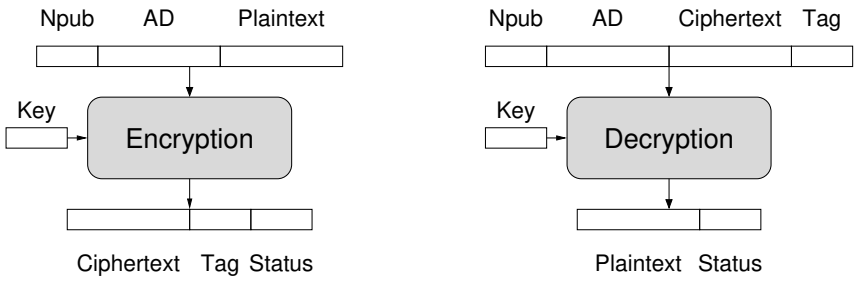


Fig. 5: Input and Output of an Authenticated Cipher. Notation: Npub - Public Message Number, AD - Associated Data

until the authenticity of the plaintext is verified. Only then, the plaintext is allowed to be released. Implementing this feature in hardware would require the memory approximately equal in size to the two-pass FIFO.

Taking these considerations into account, the two-pass FIFO is treated as an external circuit, located outside of the LWC core, and, as such, should not affect either the resource utilization or the maximum clock frequency of the LWC core.

4 Communication Protocol

All parts of a typical input and a typical output of an authenticated cipher are shown in Fig. 5, for encryption and decryption, respectively. Npub denotes Public Message Number, such as a Nonce or Initialization Vector. All parts of input to encryption, other than a key, are optional and can be omitted. If a given part is omitted, it is assumed to be an empty string. Fig. 6 shows the typical input and output format of a hash function.

The proposed format of the Secret Data Input is shown in Fig. 7. The entire input starts with an instruction, which in the case of `SDI` is limited to Load Key (LDKEY). The instruction is followed by segments. Each segment begins with a separate header, describing its type and size. In the case of `SDI`, the only segment type necessary to meet the minimum compliance criteria is: `Key`, denoting a string of bits carrying an authenticated cipher key.

The proposed format of the Public Data Input is shown in Fig. 8. The allowed instruction types are: Activate Key (ACTKEY), Authenticated Encryption (ENC), Authenticated Decryption (DEC), and Hash. The Activate Key instruction, typically directly precedes the Authenticated Encryption or Authenticated Decryption instruction. `Public Data Input (PDI)` is divided into segments. Segment types allowed during authenticated encryption include Public Message Number (Npub), Secret Message Number (Nsec), Associated Data (AD), Npub||AD, AD||Npub, Plaintext, and Length. Segment types allowed during authenticated decryption include Public Message Number (Npub), Encrypted Secret Message Number (Enc Nsec), Associated Data (AD), Npub||AD,

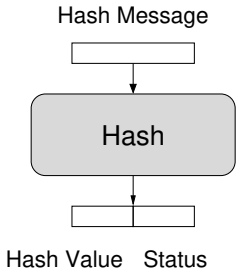


Fig. 6: Input and Output of a Hash Function.

instruction = LDKEY
seg_0_header
seg_0 = Key

Fig. 7: Format of Secret Data Input for loading the key

instruction = ACTKEY
instruction = ENC
seg_0_header
seg_0 = Npub
seg_1_header
seg_1 = AD
seg_2_header
seg_2 = Plaintext

(a)

instruction = ACTKEY
instruction = ENC
seg_0_header
seg_0 = Npub
seg_1_header
seg_1 = AD_0
seg_2_header
seg_2 = AD_1
seg_3_header
seg_3 = Plaintext0
seg_4_header
seg_4 = Plaintext1

(b)

Fig. 8: Format of Public Data Input in case of a) one segment for each data type, b) multiple segments for AD and Plaintext

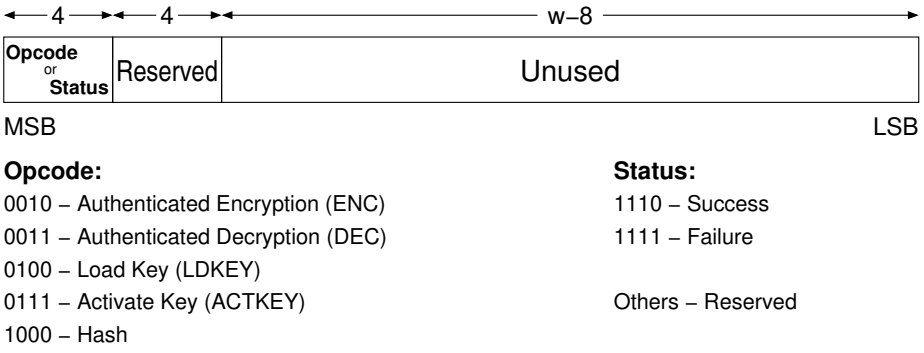


Fig. 9: Instruction/Status Format

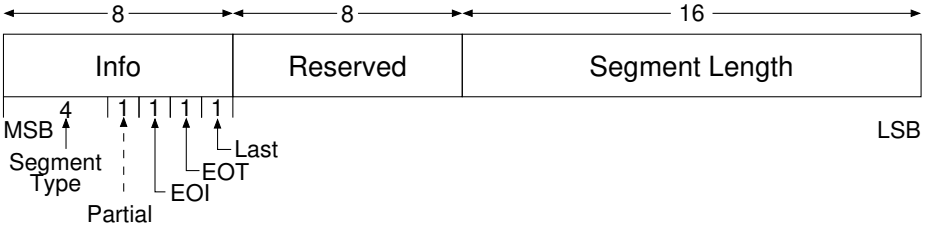


Fig. 10: Segment Header Format

AD||Npub, Ciphertext, Tag, and Length. Hashing allows only the segment types Hash Message and Length.

If no data is to be sent to the LWC core for a segment, in case of AD, Plaintext, Ciphertext, and Hash Message segments, the segment header still has to be sent with the Segment Length field of the respective header set to 0.

The Associated Data, Plaintext, and Hash Message can be (but do not have to be) divided into multiple segments (as shown in Fig. 8b). *The maximum size of each segment is assumed to be* $2^{16} - 1$ *bytes.* The primary reason for dividing AD, Plaintext, and Hash Message into multiple segments is that the full input size may be unknown when the authenticated encryption or hashing starts. Npub can only use one segment, as its size is typically quite small (in the range of 16 bytes).

The instruction/status format is shown in Fig. 9. The size of the instruction/status is always w bits. For instruction, the Opcode field determines which operation should be executed next. For status, the Opcode field is replaced by the Status field, which can be set to only two values, Success or Failure.

Table 1: Segment Type Encoding

Encoding	Type	Encoding	Type
0000	*Reserved*	1000	Tag
0001	AD	1001	Hash Value
0010	Npub||AD	1010	Length
0011	AD||Npub	1011	*Reserved*
0100	Plaintext	1100	Key
0101	Ciphertext	1101	Npub
0110	Ciphertext||Tag	1110	Nsec
0111	Hash Message	1111	Enc Nsec

The segment header format is shown in Fig. 10. The segment header is always 32 bits long and consists of:

- 4-bit *Segment Type* indicates the type of data that the current segment contains. The type encoding is defined in Table 1.
- 1-bit optional *Partial* bit indicates that the current segment contains an incomplete block of plaintext or the corresponding ciphertext. The only authenticated cipher we are aware of that requires this bit is the Round 2 CAESAR candidate AES-COPA.
- 1-bit *EOI* (End-Of-Input) indicates that the current segment is the last segment of input other than the *Length* segment, *Tag* segment, or any empty segment. EOI is set to '0' for segments leaving the cipher, i.e., on Data Output Ports.
- 1-bit *EOT* (End-Of-Type) indicates that the current segment is the last segment of the current *Segment Type*.
- 1-bit *Last* indicates that the current segment is the last segment, i.e., no more segments are associated with the given instruction.
- 8 reserved bits for future extensions.
- 16-bit *Segment Length* to specify the size of data in the given segment in *bytes*.

The majority of the segment types listed in Table 1 are self-explanatory. The meaning and usage of the remaining types are explained below:

Nsec stands for the Secret Message Number and Enc Nsec for Encrypted Secret Message Number. These types are kept primarily for compatibility with the CAESAR Hardware API [10]. Only a few candidates in the early rounds of the CAESAR competition specified these numbers as parts of inputs to encryption and decryption, respectively [8, 10].

The types Npub||AD, AD||Npub, Ciphertext||Tag, combine two consecutive input fields into the same segment, without any separation between these fields. These special segments may be used to simplify the implementations of ciphers in which the size of the former of the two merged fields is not a multiple of the word size. In particular, they may eliminate the need for implementing a variable shift, which is typically quite area-consuming in hardware.

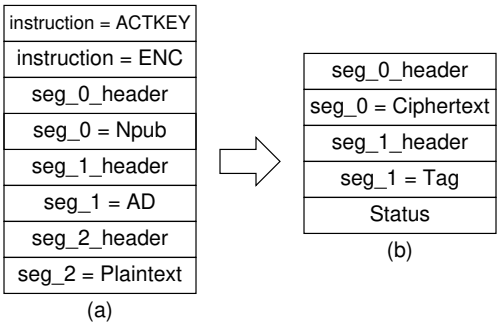


Fig. 11: Format of Public Data Input (PDI) and Data Output (DO) for authenticated encryption a) PDI, b) DO

We restrict the use of the Length Segment to "offline" algorithms, such as AES-CCM, understood as algorithms that require the availability of the lengths of the AD and plaintext (ciphertext) in advance, before the authenticated encryption (decryption) starts. The Length segment must not be used in the implementations of "online" algorithms, such as AES-GCM, in which all lengths can be calculated as the AD/plaintext/ciphertext arrives and is processed. For the "offline" authenticated ciphers, which are permitted to use the Length Segment, we make its format common for all algorithms, and define it as follows:

- Header (32 bits)
- AD length in bytes (32 bits)
- Plaintext/Ciphertext length in bytes (32 bits).

Several examples, illustrating the correct values of the flags EOI, EOT, and Last, for multiple realistic scenarios are shown in Table 2. The bottom part of this table concerns algorithms with ciphertext expansion, i.e., algorithms in which the ciphertext size is greater than the corresponding plaintext size. In case of such algorithms, for the ease of processing, any plaintext/ciphertext of the size greater than or equal to the block size must be divided into two segments PT[0], PT[1] for encryption, and CT[0], CT[1] for decryption. The former of these segments must have the size equal to a multiple of block size (MBS), the latter segment must have the size smaller than the block size (including zero) for encryption and the block size (BS) for decryption. This special formatting is to ensure that the segment length for the last segment of a particular type can be changed without the need to buffer data for the whole segment. Segments PT[0] and CT[0] can be further divided into smaller segments, each of the size equal to a multiple of the block size (MBS).

Figures 11, 12, and 13 present typical format of input (`PDI`) and output (`DO`) of authenticated encryption, decryption, and hash operation, respectively. At the `PDI` ports, an input typically starts with the key activation instruction (ACTKEY), followed by an operational instruction (ENC or DEC). Header and data segments for different types of data subsequently follow. For encryption

Table 2: Examples of correct values of input flags for encryption and decryption operations in different scenarios. BS = block size. MBS = an integer multiple of the block size, PT = plaintext, CT = ciphertext.

Types	Size	EOI	EOT	Last	Types	Size	EOI	EOT	Last	Types	Size	EOI	EOT	Last
Typical														
Example A: AD = 0, Plaintext = 0										**Example B: AD = 0, Plaintext > 0**				
Encryption					**Decryption**					**Encryption**					**Decryption**				
Npub	>0	1	1	0	Npub	>0	1	1	0	Npub	>0	0	1	0	Npub	>0	0	1	0
AD	0	0	1	0	AD	0	0	1	0	AD	0	0	1	0	AD	0	0	1	0
PT	0	0	1	1	CT	0	0	1	0	PT	>0	1	1	1	CT	>0	1	1	0
					TAG	>0	0	1	1						TAG	>0	0	1	1
Example C: AD > 0, Plaintext = 0										**Example D: AD > 0, Plaintext > 0**				
Encryption					**Decryption**					**Encryption**					**Decryption**				
Npub	>0	0	1	0	Npub	>0	0	1	0	Npub	>0	0	1	0	Npub	>0	0	1	0
AD	>0	1	1	0	AD	>0	1	1	0	AD	>0	0	1	0	AD	>0	0	1	0
PT	0	0	1	1	CT	0	0	1	0	PT	>0	1	1	1	CT	>0	1	1	0
					TAG	>0	0	1	1						TAG	>0	0	1	1
Ciphertext Expansion (AD > 0)														
Example E: Plaintext = 0										**Example F: Plaintext < BS**				
Encryption					**Decryption**					**Encryption**					**Decryption**				
Npub	>0	0	1	0	Npub	>0	0	1	0	Npub	>0	0	1	0	Npub	>0	0	1	0
AD	>0	1	1	0	AD	>0	0	1	0	AD	>0	0	1	0	AD	>0	0	1	0
PT	0	0	1	1	CT	BS	1	1	0	PT	<BS	1	1	1	CT	BS	1	1	0
					TAG	>0	0	1	1						TAG	>0	0	1	1
Example G: (Plaintext % BS) = 0										**Example H: (Plaintext % BS) > 0**				
Encryption					**Decryption**					**Encryption**					**Decryption**				
Npub	>0	0	1	0	Npub	>0	0	1	0	Npub	0	0	1	0	Npub	>0	0	1	0
AD	>0	0	1	0	AD	>0	0	1	0	AD	0	0	1	0	AD	>0	0	1	0
PT[0]	MBS	1	0	0	CT[0]	MBS	0	0	0	PT[0]	MBS	0	0	0	CT[0]	MBS	0	0	0
PT[1]	0	0	1	1	CT[1]	BS	1	1	0	PT[1]	<BS	1	1	1	CT[1]	BS	1	1	0
					TAG	16	0	1	1						TAG	16	0	1	1

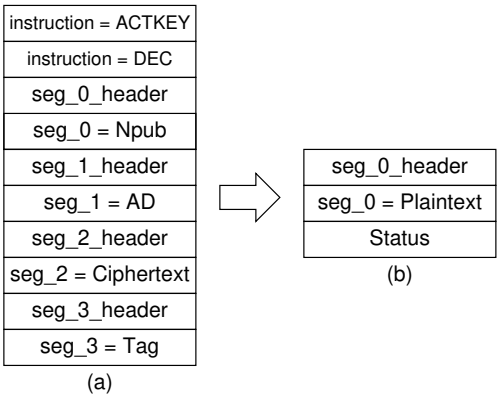


Fig. 12: Format of Public Data Input (PDI) and Data Output (DO) for authenticated decryption a) PDI, b) DO

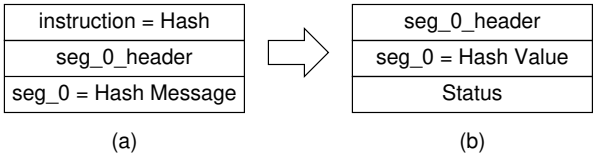


Fig. 13: Format of Public Data Input (PDI) and Data Output (DO) for hashing a) PDI, b) DO

and decryption operation, the order typically is Npub, AD, Data (Plaintext or Ciphertext) and Tag (for decryption only). The order of segment types that can be processed by a given core is a feature of the specific implementation and needs to be clearly documented.

5 Timing Characteristics

Figures 14 and 15 illustrate the timing characteristics of the ports `PDI` and `DO`, respectively. Input ports are shown in **blue** and the output ports in **red**. The contents of data buses are read and acknowledged when `*_valid` and its corresponding `*_ready` are both asserted. Data is assumed to be present at the output of the source module when `*_valid` is asserted. The corresponding signals `*_valid` and `*_ready` can be asserted in arbitrary order, i.e., `*_valid` first, or `*_ready` first, or both at the same time. The implementation should hold `do_data` at a constant value (either 0 or the previous valid output data) when `do_valid='0'`.

The optional `do_last` output indicates that the current output word (w bits), appearing at the `do_data` bus, is the last word of the final output, i.e. the status

word, shown in Fig. 9. The output `do_last` should be asserted only if the output `do_valid` is asserted, too.

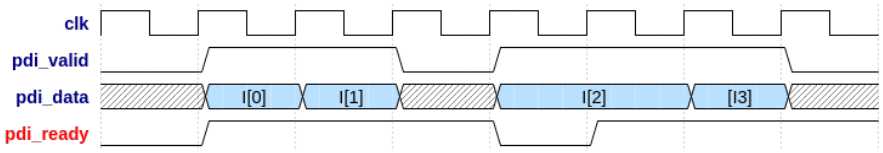


Fig. 14: Example timing diagram for `PDI`

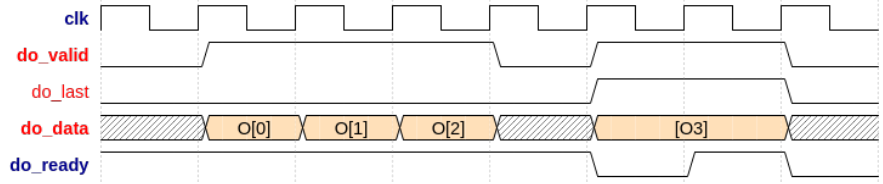
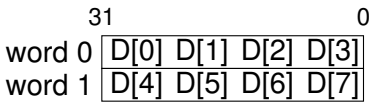


Fig. 15: Example timing diagram for `DO`

Addr	Value
n	D[0]
n+1	D[1]
n+2	D[2]
n+3	D[3]
n+4	D[4]
n+5	D[5]
n+6	D[6]
n+7	D[7]

(a) Memory



(b) 32-bit word representation

Fig. 16: Data representation

6 Side-channel Resistant Implementations

The NIST LWC Standardization Process does not mandate, but does encourage, algorithms and implementations that support effective and efficient side-channel countermeasures [1]. This includes implementations secure against power analysis side-channel attacks (SCA), such as Simple Power Analysis (SPA) and Differential Power Analysis (DPA).

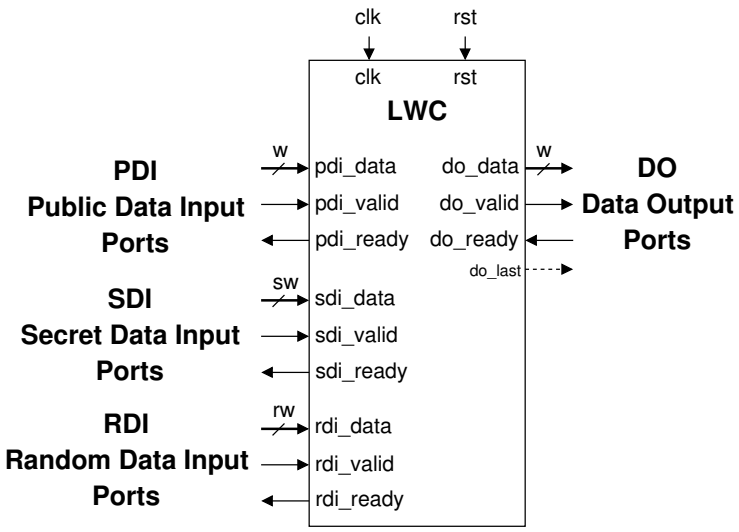


Fig. 17: LWC interface for the single-pass algorithms with random data input

One common requirement for nearly all algorithmic power analysis countermeasures, e.g., Boolean Masking and Threshold Implementations (TI), is the consumption of randomness during cipher operations. An example is the necessity of meeting the TI uniformity property, which often requires so-called refreshing randomness [25, 26].

To facilitate side-channel resistant implementations that require refreshing randomness, we propose an additional `Random Data Input (RDI)` bus, comprised of the signals `rdi_data` of user selectable width rw, `rdi_ready` and `rdi_valid` (see Fig. 17). No protocol support is provided for this optional interface. The protected LWC core simply asserts a `rdi_ready` signal, checks `rdi_valid` and then reads rw bits of random data.

The LWC API makes no assumptions on which kind of SCA protection is being implemented. However, if data has to be separated into shares, this task should be performed in software. All shares should be sent to the LWC cipher through the regular inputs, i.e., all public data should still arrive via `PDI`, and depart via `DO`. For an n-share implementation, the Plaintext segments should then contain shares 1 through n, w bits each, followed by the next w bits of each share, and so on. In the same fashion, separated data should leave the core. Figure 18 shows how an $m \cdot w/8$-byte Plaintext, split into n shares, should arrive at `PDI`.

Justification: Share separation in software facilitates verification of countermeasures in hardware using leakage detection techniques (e.g., t-test or χ^2-test), for which false-positive results could occur if share separation were performed in hardware. Arrival of share-separated data in w-bit shares allows the designer

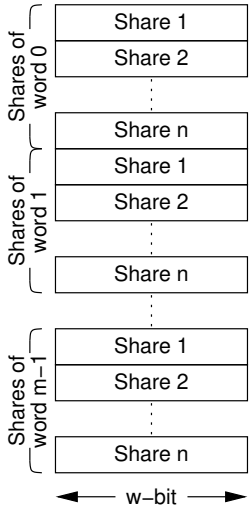


Fig. 18: Plaintext/Ciphertext of $m \cdot w/8$ bytes split into n shares

the choice of immediately processing individual shares, or internally buffering sequential shares as part of a larger calculation.

The implementation designer should write a script that takes as input the test vectors of an unprotected implementation and outputs the share separated data. Furthermore, an additional script should be able to combine share separated outputs back so that they can be compared to the test vectors for the output on an unprotected implementation.

7 Differences Compared to the CAESAR Hardware API

Major differences between the proposed Lightweight Cryptography Hardware API and the CAESAR Hardware API, defined in [10, 11], are as follows:

In terms of the Minimum Compliance Criteria:

1. One additional configuration, encryption/decryption/hashing, has been added on top of the previously supported configuration: encryption/decryption.
2. The maximum sizes of AD/plaintext/ciphertext supported in the CAESAR Hardware API were a) $2^{32} - 1$ for single-pass ciphers, and $2^{11} - 1$ for two-pass ciphers. In the LWC API, we add one additional maximum size $2^{16} - 1$, which should be supported by implementations of both kinds of ciphers, and which can be used for fair comparison among implementations of algorithms belonging to these two distinct cipher groups. The same maximum sizes are also used to limit the length of hash messages.

In terms of the Interface:

1. An additional optional output, `do_last`, has been added to the Data Output Ports.

In terms of the Communication Protocol:

1. In the Instruction/Status, shown in Fig. 9, an additional opcode value, 1000 - hash, has been added, and the size of the Instruction/Status has been changed from 16 bits to w bits, in order to match the PDI data bus width. Even though this minor revision marks a change compared to the formal specification of the CAESAR API [10], it does not change the way this feature was implemented in the Development Package for Hardware Implementations Compliant with the CAESAR Hardware API [12], as well as in the majority of lightweight implementations of CAESAR candidates reported to date.
2. In the Segment Header word, two additional Segment Type values, 0111 - Hash Message and 1001 - Hash Value, have been added.

In terms of Support for Side-Channel Resistant Implementations:

No support for side-channel resistant hardware implementations was provided in the CAESAR Hardware API. This specification addresses this issue in Section 6, by defining

1. An extended interface, shown in Fig. 17,
2. The requirement for the generation and merging of shares outside of the LWC core, and
3. The mechanism for passing the input shares to the core and the output shares from the core, as shown in Fig. 18.

8 Conclusions

We have defined the full specification of the hardware API for lightweight cryptography, suitable for hardware benchmarking of candidates competing in the NIST Lightweight Standardization Process and their comparison with a previous generation of authenticated encryption algorithms, including CAESAR candidates and standards such as AES-GCM and AES-CCM.

Our proposal meets one of the fundamental properties of every properly defined API: If a given algorithm is implemented independently by two different groups using the same API, one should be able to encrypt a plaintext using the first implementation, and decrypt it using the second implementation. To be exact, our assumption is that either

1. Both implementations use the same values of the data port widths w and sw, or
2. Simple reformatting (word width conversion) of the input to decryption is performed outside of the cipher core (in software or hardware).

A similar API, described in [10, 11], has been successfully used to implement and benchmark all Round 2 and Round 3 CAESAR candidates [15].

Acknowledgements

The authors would like to thank the remaining co-authors of the CAESAR Hardware API, Ahmed Ferozpuri, Farnoud Farahmand, and Panasayya Yalla, for fruitful discussions and testing ideas presented in this specification through efficient hardware implementations of multiple CAESAR candidates.

References

1. NIST, "Lightweight Cryptography: Project Overview," https://csrc.nist.gov/projects/lightweight-cryptography, 2019.
2. K. Gaj, E. Homsirikamol, and M. Rogawski, "Fair and Comprehensive Methodology for Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates Using FPGAs," in *Cryptographic Hardware and Embedded Systems, CHES 2010*, ser. LNCS, vol. 6225, Santa Barbara, CA, Aug. 2010, pp. 264–278.
3. E. Homsirikamol, M. Rogawski, and K. Gaj, "Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates Using FPGAs," Cryptology ePrint Archive 2010/445, 2010.
4. K. Gaj, E. Homsirikamol, M. Rogawski, R. Shahid, and M. U. Sharif, "Comprehensive Evaluation of High-Speed and Medium-Speed Implementations of Five SHA-3 Finalists Using Xilinx and Altera FPGAs," Cryptology ePrint Archive 2012/368, 2012.
5. J.-P. Kaps, K. K. Surapathi, B. Habib, S. Vadlamudi, S. Gurung, and J. Pham, "Lightweight Implementations of SHA-3 Candidates on FPGAs," in *12th International Conference on Cryptology in India, Indocrypt 2011*, ser. LNCS, vol. 7107, Chennai, India, Dec. 2011, pp. 270–289.
6. B. Baldwin, A. Byrne, L. Lu, M. Hamilton, N. Hanley, M. O'Neill, and W. P. Marnane, "FPGA Implementations of the Round Two SHA-3 Candidates," in *2010 International Conference on Field Programmable Logic and Applications, FPL 2010*, Milan, Italy, Aug. 2010, pp. 400–407.
7. M. Knezevic, K. Kobayashi, J. Ikegami, S. Matsuo, A. Satoh, Ü. Kocabas, J. Fan, T. Katashita, T. Sugawara, K. Sakiyama, I. Verbauwhede, K. Ohta, N. Homma, and T. Aoki, "Fair and Consistent Hardware Evaluation of Fourteen Round Two SHA-3 Candidates," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 5, pp. 827–840, May 2012.
8. "CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness - web page," https://competitions.cr.yp.to/caesar.html, 2019.
9. E. Homsirikamol, W. Diehl, A. Ferozpuri, F. Farahmand, M. U. Sharif, and K. Gaj, "A universal hardware API for authenticated ciphers," in *2015 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2015*, Riviera Maya, Mexico, Dec. 2015.
10. E. Homsirikamol, W. Diehl, A. Ferozpuri, F. Farahmand, P. Yalla, J.-P. Kaps, and K. Gaj, "CAESAR Hardware API," Cryptology ePrint Archive 2016/626, 2016.
11. ——, "Addendum to the CAESAR Hardware API v1.0," George Mason University, Fairfax, VA, GMU Report, Jun. 2016.
12. E. Homsirikamol, P. Yalla, and F. Farahmand, "Development Package for Hardware Implementations Compliant with the CAESAR Hardware API," 2016.
13. E. Homsirikamol, P. Yalla, F. Farahmand, W. Diehl, A. Ferozpuri, J.-P. Kaps, and K. Gaj, "Implementer's Guide to Hardware Implementations Compliant with the CAESAR Hardware API," GMU, Fairfax, VA, GMU Report, 2016.

14. M. Tempelmeier, F. De Santis, G. Sigl, and J.-P. Kaps, "The CAESAR-API in the real world — Towards a fair evaluation of hardware CAESAR candidates," in *2018 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2018*, Washington, DC, Apr. 2018, pp. 73–80.
15. Cryptographic Engineering Research Group (CERG) at George Mason University, "Hardware Benchmarking of CAESAR Candidates," https://cryptography.gmu.edu/athena/index.php?id=CAESAR, 2019.
16. P. Yalla and J.-P. Kaps, "Evaluation of the CAESAR hardware API for lightweight implementations," in *2017 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2017*, Cancun, Mexico, Dec. 2017.
17. K. Gaj, E. Homsirikamol, W. Diehl, A. Ferozpuri, F. Farahmand, M. X. Lyons, and P. Yalla, "Toward Fair and Comprehensive Benchmarking of CAESAR Candidates in Hardware: Standard API, High-Speed Implementations in VHDL/Verilog, and Benchmarking Using FPGAs," in *Directions in Authenticated Ciphers Workshop, DIAC 2016*, Nagoya, Japan, Sep. 2016.
18. F. Farahmand, W. Diehl, A. Abdulgadir, J.-P. Kaps, and K. Gaj, "Improved Lightweight Implementations of CAESAR Authenticated Ciphers," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2018*, Boulder, CO, Apr. 2018, pp. 29–36.
19. W. Diehl, A. Abdulgadir, F. Farahmand, J.-P. Kaps, and K. Gaj, "Comparison of cost of protection against differential power analysis of selected authenticated ciphers," in *2018 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2018*, Washington, DC, Apr. 2018, pp. 147–152.
20. ——, "Comparison of Cost of Protection against Differential Power Analysis of Selected Authenticated Ciphers," *Cryptography*, vol. 2, no. 3, p. 26, Sep. 2018.
21. W. Diehl, F. Farahmand, A. Abdulgadir, J.-P. Kaps, and K. Gaj, "Face-off between the CAESAR Lightweight Finalists: ACORN vs. Ascon," in *2018 International Conference on Field Programmable Technology, FPT 2018*, Naha, Okinawa, Japan, Dec. 2018.
22. ——, "Face-off between the CAESAR Lightweight Finalists: ACORN vs. Ascon," Cryptology ePrint Archive 2019/184, 2019.
23. NIST, "DRAFT Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process," https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/Draft-LWC-Submission-Requirements-April2018.pdf, 2019.
24. ARM, "AMBA: The Standard for On-Chip Communication," https://www.arm.com/products/silicon-ip-system/embedded-system-design/amba-specifications, 2019.
25. E. Trichina, "Combinational Logic Design for AES SubByte Transformation on Masked Data," Cryptology ePrint Archive 2003/236, Nov. 2003.
26. S. Nikova, C. Rechberger, and V. Rijmen, "Threshold Implementations Against Side-Channel Attacks and Glitches," in *Information and Communications Security, ICICS 2006*, ser. LNCS, vol. 4307. Springer Berlin Heidelberg, 2006, pp. 529–545.