Implementer's Guide to Hardware Implementations Compliant with the Hardware API for Lightweight Cryptography version 1.0.3

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

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

October 24, 2020

Contents

1 Introduction

The primary purpose of this publication is to provide support and guidance for hardware designers interested in efficient implementation and benchmarking of submissions to the NIST Lightweight Cryptography Standardization Process [1]. To assure the fairness of benchmarking and compatibility among implementations of the same algorithm by different designers, Hardware API for Lightweight Cryptography (LWC) was created [2]. The major parts of this API include the minimum compliance criteria, interface, communication protocol, and timing characteristics supported by the implemented core. For the purpose of fair comparison with the existing standards, as well as candidates in the earlier CAESAR contest (Competition for Authenticated Encryption: Security, Applicability, and Robustness) [3], conducted in the period 2013-2019, our proposed implementation and benchmarking framework is not limited to submissions to the current NIST standardization process. Instead, it attempts to support lightweight implementations of all authenticated ciphers (a.k.a. authenticated encryption with associated data (AEAD) algorithms) with an optional hash functionality.

In order to speed up the development of multiple implementations necessary for fair evaluation of candidates in the NIST standardization process, we have created the Development Package for Lightweight Cryptography. As a part of this package, the designers are provided with the following support aimed at speeding-up and simplifying the development process:

1. universal top-level block diagram of the main core, called LWC, including four lower-level units called the PreProcessor, CryptoCore, Header FIFO, and PostProcessor

2. universal VHDL code for the PreProcessor, PostProcessor, and Header FIFO

3. hardware interface for all major building blocks, with the special focus on CryptoCore

4. recommended design procedure for the CryptoCore, and its integration with the remaining three units comprising the LWC core

5. reference VHDL code of an example CryptoCore for a dummy authenticated cipher with hash functionality, fully verified for correct functionality

6. universal testbench suitable for full verification of any implementation of an LWC core compliant with the proposed LWC Hardware API

7. universal test vector generator, based on the reference C implementations of the respective authenticated ciphers and hash functions.

In this document, we describe all these supporting materials one by one.

It should be stressed that *implementations of authenticated ciphers and hash functions compliant with the LWC Hardware API can also be developed without using any resources described in this document, by just following directly the specification of the LWC Hardware API.*

Depending on the personal or team preference, the designers can choose one of three major approaches:

1. using only the specification of the LWC Hardware API, and developing the entire design, hardware description language code, and verification framework from scratch

2. using only selected components of the Development Package, e.g., a universal test vector generator and a universal testbench

3. using all resources of the Development Package.

The more that the Development Package is used, the shorter the development time is likely to become. On the other hand, the obtained results, e.g., in terms of resource utilization, maximum clock frequency, latency, and throughput are likely to be very comparable, with only minor gains (typically only in terms of resource utilization) achieved by using Approach 1.

The users following Approach 1 are encouraged to read at least Chapters 2 and 7. The users following Approach 2 are encouraged to read additionally Chapter 6. Finally, the users following Approach 3 should consider getting familiar with the entire document.

This document is, on one hand, a subset of the Implementer's Guide developed during the CAESAR competition [4], as all chapters devoted specifically to high-speed implementations have been eliminated. On the other hand, it also contains substantial extensions and updates compared to the CAESAR's Implementer's Guide, especially in Chapters 5 and 6. Hardware designers familiar with the CAESAR Development Package [5] and the associated Implementer's Guide [4] should consider reading Chapter 8 first.

2 Compliance with the Requirements for Fair Benchmarking

In this chapter, we focus on the requirements that have to be met for the code to be suitable for evaluation and ranking of candidates in the Lightweight Cryptography Standardization Process.

First and foremost, the design must meet all requirements formulated in the specification of the Hardware API for Lightweight Cryptography [2].

However, it is strongly recommended that the hardware description language (HDL) code meets the following additional guidelines:

1. The primary HDL code should be portable among multiple technologies and supported by a wide variety of tools. In particular, this code should be free of any vendor-specific constructs, directives, macros, primitives, etc. The code optimized for a specific subset of devices and/or tools of a particular vendor can be submitted as well, but it will be compared only with the code optimized in the same fashion.

2. The implementation should use only storage elements based on flip-flops, rather than latches, which is necessary to ensure consistent analysis of maximum clock frequency and area. Flip-flops should be active on only one edge of the clock (preferably the rising edge of the clock).

3. Implementations should not use tri-state buffers or scan-cell flip flops.

4. Coding guidelines regarding reset (synchronous vs. asynchronous, active-high vs. active-low) vary between FPGAs and ASICs, as well as among various vendors. The designers have the freedom to apply different styles, including a hybrid approach, in which some portions

of the circuit treat the reset signal as synchronous and other portions as asynchronous. At the same time, the designers should be aware that this choice may affect the area, maximum clock frequency, and power consumption of their circuit. As a part of evaluating candidates in the NIST standardization process, verification and FPGA benchmarking will be performed under the assumption that the reset is by default synchronous and active high. In particular, our Development Package, containing portions of the code common for multiple ciphers/hash functions, as well as our universal testbench, will support only this type of reset. Other types of reset can be supported as non-default options, for verification and benchmarking using different tools and implementation targets (e.g., ASICs).

5. Each implementation should provide a comma-separated values (CSV) file of generic parameters and allowed combinations of their values. Each column should represent a generic parameter, and each row should be a legitimate combination of the generic values. Although this file is not used by the current version of the Development Package, it may be used in the future by the extended testbench and synthesis scripts to easily iterate over all possible variants of the submitted design.

The code that does not follow these guidelines, with the special focus on compliance with [2], may be flagged during the initial review process as not fully conforming to the requirements of the fair benchmarking process.

3 Top-level Block Diagram

Fig. 3.1 shows the proposed top-level block diagram of the LWC core, implementing an authenticated cipher with or without hash functionality, compliant with the LWC Hardware API. The top-level unit is made of four lower-level units called the PreProcessor, CryptoCore, Header FIFO, and PostProcessor. Ports with names marked in blue are optional. They include:

- ports used only by two-pass algorithms, used for communication between the CryptoCore and the Two-Pass FIFO
- hash and hash_in ports used only by authenticated ciphers with the hash functionality, and
- the do_last output, facilitating the communication with a potential follow-up AXI4-Stream Slave.

3.1 PreProcessor

The PreProcessor is responsible for the following tasks

- parsing segment headers
- loading keys
- passing input blocks to the CryptoCore, along with information required for padding
- keeping track of the number of data bytes left to process.

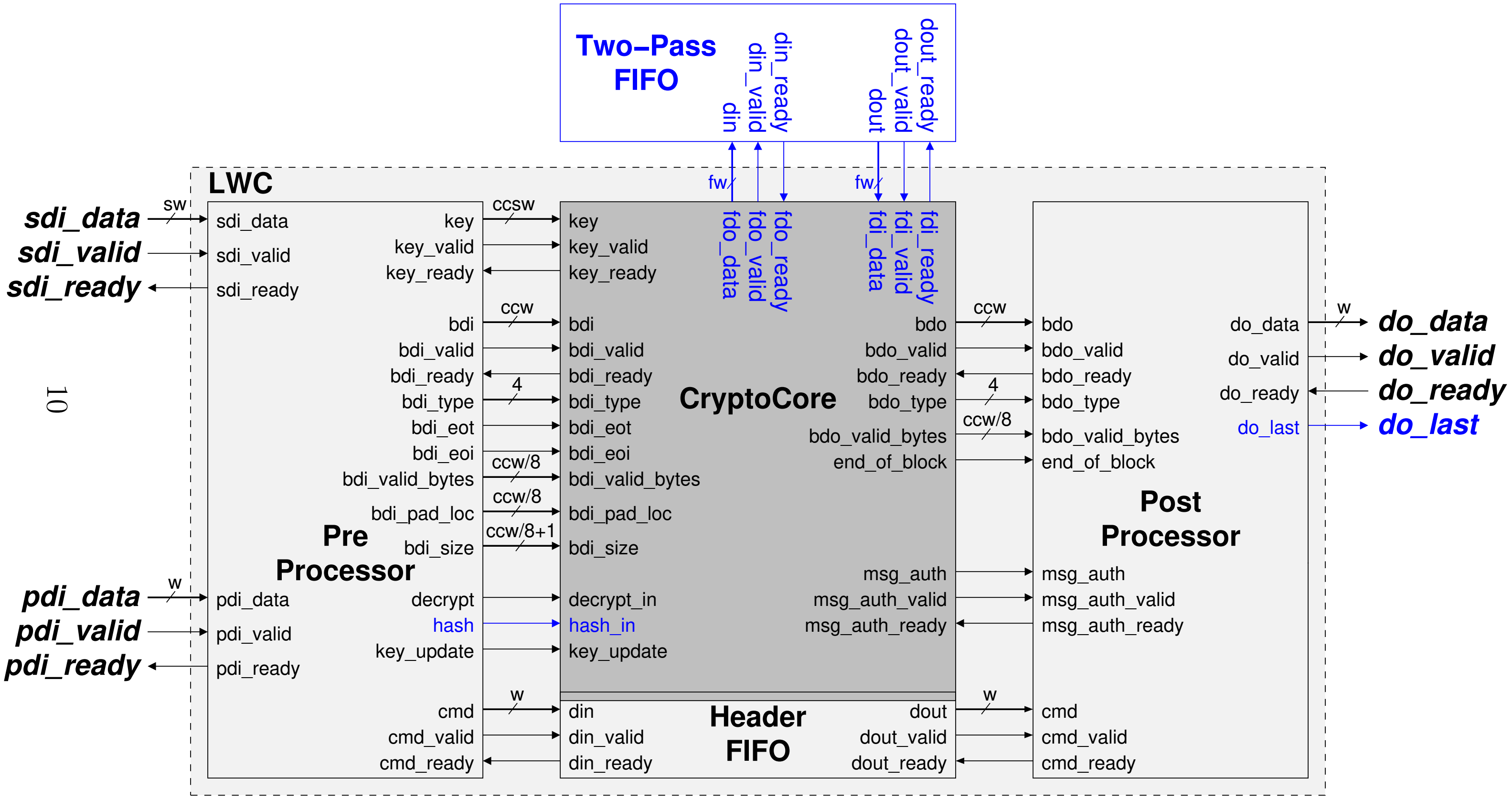

Figure 3.1: Top-level block diagram of the LWC core

It is assumed that padding is performed within the CryptoCore, based on the information provided by the PreProcessor. The signal `bdi_type` specifies the type of data on the `bdi_data` bus. Table 5.2 lists the encoding for different data types.

3.2 PostProcessor

The PostProcessor is responsible for the following tasks:

- clearing any portions of output words not belonging to the ciphertext or plaintext (invalid bytes are set to zero)
- generating the header for the output data blocks
- generating the status block with the result of authentication.

3.3 Header FIFO

The Header FIFO is a small 4 x w FIFO that temporarily stores all segment headers that need to be passed to the output.

4 LWC Core Development

4.1 Introduction

The development and benchmarking of a lightweight implementation of a selected authenticated cipher, with or without hash functionality, can be performed using the following major steps, described in the subsequent chapters of this guide:

1. Develop the CryptoCore (Chapter 5)
2. Generate test vectors (Section 6.1)
3. Verify the LWC design using simulation (Section 6.2)
4. Verify the LWC desing using hardware testbeds (Section 6.3.2)
5. Generate optimized results for LWC using FPGA tools (Chapter 7).

4.2 The LWC Configuration

The entity declaration of LWC for lightweight implementations is available as a part of the Development Package in the file

```
$root/hardware/LWC_rtl/LWC.vhd
```

There are two constants (W and SW) that can be changed to configure the external bus width. The default value for both is 32, as this provides the most flexibility for designers. The type of reset signal can switch from synchronous active-high(default) to asynchronous active-low to support ASIC development by setting ASYNC_RSTN to true. These constants should be set in the corresponding package file

`$root/hardware/LWC_rtl/NIST_LWAPI_pkg.vhd`

It is assumed that all other constants are not changed. Instead, all necessary configurations for the CryptoCore should be performed in the design-specific package file. For an example see:
`$root/hardware/dummy_lwc/src_rtl/v1/design_pkg.vhd`

In any case, the cipher specific constants `TAG_SIZE`, `HASH_VALUE_SIZE`, `CCSW`, `CCW`, and the derived `CCWdiv8` must be set there. They are needed and read by the LWC. As those constants are cipher specific, they have no default values, to ensure that the designer explicitly sets them.

Table 4.1 lists all expected parameters for LWC, PreProcessor, and PostProcessor. Deprecated constants are left for compatibility with the CipherCores developed during the CAESAR competitions [6]. They should be left unchanged for any new designs.

Table 4.1: LWC configuration

Name	**Type**	**Default Value**	**Definition**
Constants (set in `design_pkg.vhd`)			
TAG_SIZE	Integer	–	Size of AEAD-Tag
HASH_VALUE_SIZE	Integer	–	Size of hash value.
CCSW	Integer	–	internal key width (8, 16, 32).
CCW	Integer	–	internal data width (8, 16, 32).
CCWdiv8	Integer	–	CCW / 8
Constants (set in `NIST_LWAPI_pkg.vhd`)			
W	Integer	32	external data width
SW	Integer	W	external key width
ASYNC_RSTN	Boolean	False	Asynchronous active low when true. Synchronous active high when false.
Deprecated constants (set in `NIST_LWAPI_pkg.vhd`)			
TAG_INTERNAL	Boolean	True	Verification must be done by the LWC Core

4.3 I/O Port Widths

Consistently with the specification of the LWC Hardware API the external I/O port widths (pdi/do and sdi) can be set to 8, 16, and 32 bits in the package `NIST_LWAPI_pkg.vhd`, using the constants W and SW. The internal

I/O port widths (bdi/bdo and key) are implementation specific an can be set to 8, 16 or 32 bits in the core configuration package `design_pkg.vhd`, using CCW and CCSW.

The following combinations (w, ccw) are supported in the current version of the Development Package: (32, 32), (32, 16), (32, 8), (16, 16), and (8, 8). The following combinations (sw, ccsw) are supported: (32, 32), (32, 16), (32, 8), (16, 16), and (8, 8). However, w and sw must be always the same.

4.4 Limitations

The current implementation of the Pre- and PostProcessor do not support the following features:

- Ciphertext||Tag segment
- Intermediate tags
- multiple segments of the same type separated by segments of another type, e.g. header and trailer, treated as two segments of the type AD, separated by message segments.
- data blocks are never split across two segments as shown in Figs. 4.1, and 4.2.

Additionally, there is no error handling for protocol errors. However, in simulation, multiple assertions ensure that the simulation is stopped if an unexpected header or data type is received.

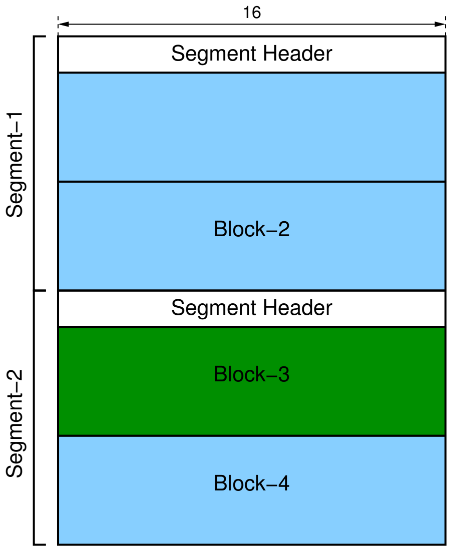

Figure 4.1: Correct way of splitting blocks

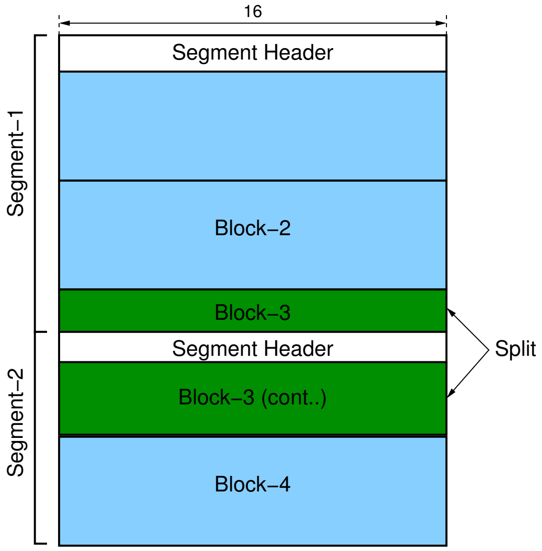

Figure 4.2: Incorrect way of splitting blocks

5 CryptoCore Development

5.1 Byte Order

All data is assumed to be represented in big endianness.

5.2 Interface

The interface of the CryptoCore is shown in Figure 5.1. Ports marked in blue are optional and used only if required. This approach allows the synthesis tool to trim the unused ports and the associated logic from the design, resulting in a better resource utilization.

Data input ports are limited to `key` and `bdi` (block data input). The key port is controlled using the handshake signals `key_valid` and `key_ready`. `key_update` is used to notify the CryptoCore that it should update the internal key prior to processing the next message.

The `bdi` port is controlled using the `bdi_valid` and `bdi_ready` handshake signals.

The correct values of `bdi_valid_bytes`, `bdi_pad_loc` and `bdi_size` for various numbers of valid bytes within a 4-byte data block are shown in Table 5.1, where:

- Case A: Either not the last block or the last block with all 4 bytes valid.
- Case B: The last block with 3 bytes valid.
- Case C: The last block with 1 byte valid.
- Case D: The last block with no valid bytes.

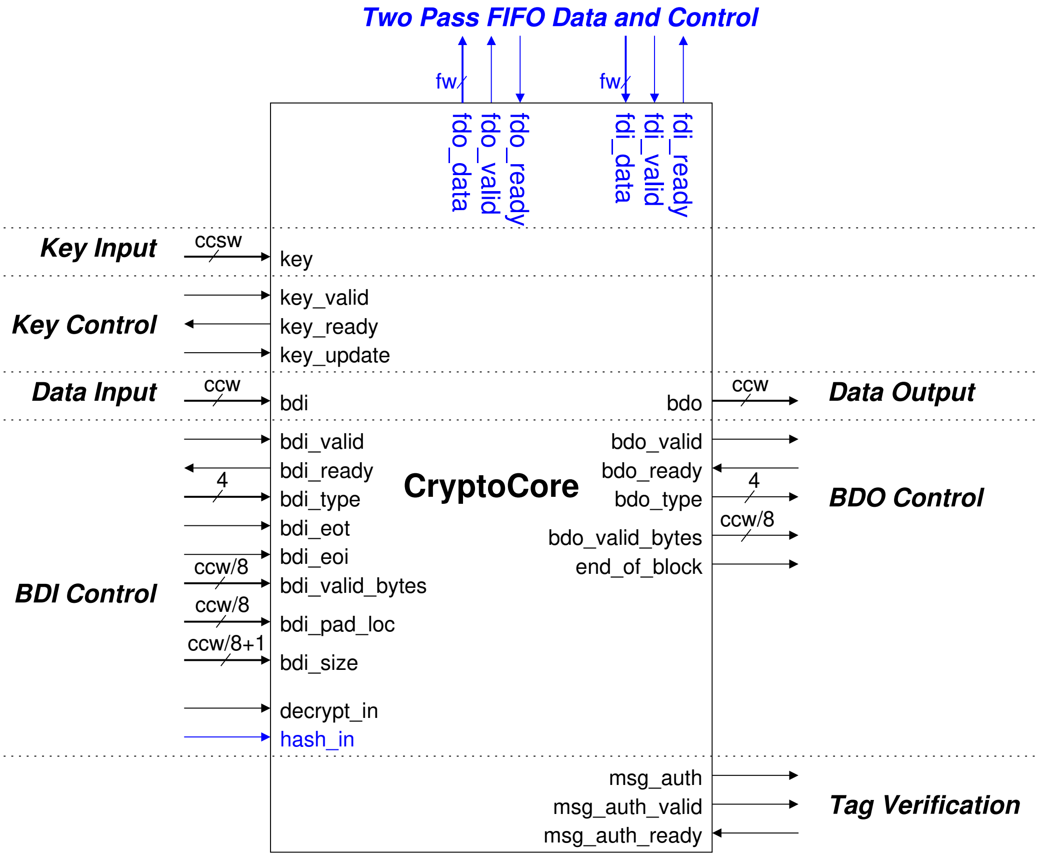

Figure 5.1: Interface of the CryptoCore.

The signal bdi_eot indicates that the current BDI block is the last block of its type. This signal is used only when the type is either AD, Plaintext, Ciphertext, or Hash Message. The signal bdi_eoi indicates that the current BDI block is the last block of input other than a block of the Length segment, a block of the Tag segment, or a block of padding.

The input and output data types are indicated by bdi_type and bdo_type using the encoding shown in Table. 5.2.

When processing authenticated encryption with associated data (AEAD), the input decrypt_in informs the core whether the operation is encryption or decryption. The input hash_in informs the core that a current operation is a hash, or an encryption/decryption.

It must be noted that all ports of the BDI control group and bdi are

Table 5.1: Values of the special control signals bdi_valid_bytes, bdi_pad_loc, and bdi_size for the bdi bus with a width of 32 bits. *Byte Validity* represents the byte locations in bdi that were the part of input (AD, PT, CT, or hash message) before padding.

Byte/Bit Position	**3**	**2**	**1**	**0**	**3**	**2**	**1**	**0**
	Case A				**Case B**			
Byte Validity	▨	▨	▨	▨	▨	▨	▨	
bdi_valid_bytes	1	1	1	1	1	1	1	0
bdi_pad_loc	0	0	0	0	0	0	0	1
bdi_size		1	0	0		0	1	1
	Case C				**Case D**			
Byte Validity	▨							
bdi_valid_bytes	1	0	0	0	0	0	0	0
bdi_pad_loc	0	1	0	0	1	0	0	0
bdi_size		0	0	1		0	0	0

synchronized with the bdi_valid input. Their values should be read only when the bdi_valid signal is high. The same scenario also applies to the BDO Control group and bdo, which are synchronized with the value of the bdo_valid output.

The bdo port is controlled using the bdo_valid and bdo_ready handshake signals. bdo_valid_bytes is the encoding of the byte locations in bdo that are valid. It is used to clear any unused portion of bdo in the PostProcessor and uses the same convention as bdi_valid_bytes. The encoding is illustrated in Table 5.1. The end_of_block signal indicates the last word of an output block. bdo_type is not evaluated by the PostProcessor, however, for future extensions, it is highly recommended to implement this feature. There is no penalty in terms of area, as it gets trimmed during synthesis.

The Tag Verification ports (msg_auth_*) are used only during an authenticated decryption operation. The CryptoCore must provide msg_auth to indicate its result and set msg_auth_valid to high until the PostProcessor is ready (msg_auth_ready is active).

The description of all *CryptoCore* ports are provided in Table 5.3. Ports related to the bdi control are categorized according to the following criteria:

COMM A handshake signal.

Table 5.2: bdi_type and bdo_type Encoding

Encoding	Generic	Type
0001	HDR_AD	Associated Data
0100	HDR_PT	Plaintext
0101	HDR_CT	Ciphertext
1000	HDR_TAG	Tag
1100	HDR_KEY	Key
1101	HDR_NPUB	Npub
0111	HDR_HASH_MSG	Hash message
1001	HDR_HASH_VALUE	Hash value

INPUT INFO An auxiliary signal that remains valid until a given input is fully processed. Deactivation is typically done at the end of input.

SEGMENT INFO An auxiliary signal that remains valid for the current segment. Its value changes when a new segment is received via the PDI data bus.

BLOCK INFO An auxiliary signal that is valid for the current input block. Its value changes when a new block is read.

The description of all ports of the *Header FIFO* are provided in Table 5.4.

5.3 Handshakes

This section presents examples of handshakes. All ports in the figures of this section are represented by a blue and red color, for input and output ports, respectively.

The data on the buses is controlled using the handshake signals. The *_valid signals are set to high if the data on the corresponding bus is valid. If the module is ready to receive the data, the corresponding *_ready signals are set to high. These two handshaking signals operate independently.

Fig. 5.2 shows an example of loading a 128-bit key, for $sw = 32$. The key_update signal indicates the update of the key. It is decoupled from key_valid and key_ready and stays high until the key is fully transmitted.

Table 5.3: CryptoCore Port Descriptions.

Name	Direction	Size	Description
Data Input & Output			
key	in	ccsw	Key data
bdi_data	in	ccw	Block data input
bdo_data	out	ccw	Block data output
Key Control			
key_valid	in	1	Key data is valid
key_ready	out	1	LWC core is ready to receive a new key
key_update	in	1	Key must be updated prior to processing a new input
BDI Control			
bdi_valid	in	1	[COMM] BDI data is valid
bdi_ready	out	1	[COMM] LWC Core is ready to receive data
bdi_pad_loc	in	ccw/8	[BLOCK INFO] Encoding of the byte location where padding begins.
bdi_valid_bytes	in	ccw/8	[BLOCK INFO] Encoding of the byte locations that are valid.
bdi_size	in	ccw/8+1	[BLOCK INFO] Number of valid bytes in bdi.
bdi_eot	in	1	[BLOCK INFO] The current BDI block is the last block of its type. Note: Only applies when the type is either AD, Plaintext, Ciphertext, or Hash message.
bdi_eoi	in	1	[BLOCK INFO] The current BDI block is the last block of input other than a block of the Tag segment.
bdi_type	in	4	[BLOCK INFO] Type of BDI data. See Table 5.2.
decrypt_in	in	1	[INPUT INFO] 0=Encryption, 1=Decryption
hash_in	in	1	[INPUT INFO] 0=Encryption/Decryption, 1=Hash
BDO Control			
bdo_valid	out	1	BDO data is valid
bdo_ready	in	1	PostProcessor is ready to receive data.
bdo_valid_bytes	in	ccw/8	[BLOCK INFO] Encoding of the byte locations that are valid.
end_of_block	out	1	[BLOCK INFO] The current BDO block is the last block of its type.
bdo_type	out	4	[BLOCK INFO] Type of BDO data. See Table 5.2.
TAG Verification			
msg_auth	out	1	1=Authentication success, 0=Authentication failure
msg_auth_valid	out	1	Authentication output is valid
msg_auth_ready	in	1	PostProcessor is ready to accept authentication result

Table 5.4: Header FIFO Port Descriptions.

Name	Direction	Size	Description
PreProcesor & FIFO			
`din`	in	w	Header info
`din_valid`	in	1	data is valid
`din_ready`	out	1	FIFO ready to receive data
PostProcesor & FIFO			
`dout`	out	w	Header info
`dout_valid`	out	1	data is valid
`dout_ready`	in	1	PostProcessor ready to receive data

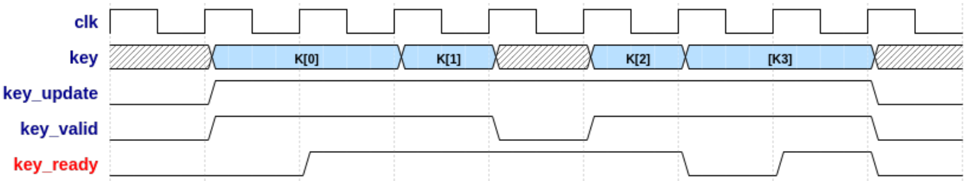

Figure 5.2: Handshake example of loading a key, for ccsw=32

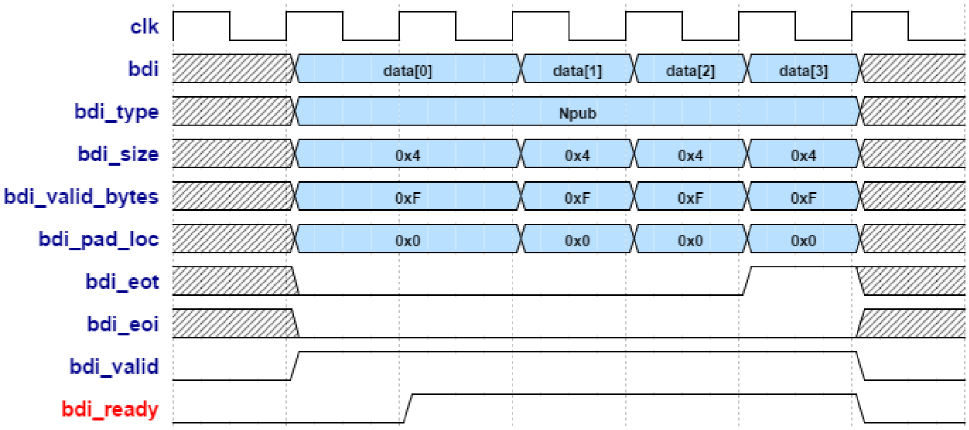

Figure 5.3: Handshake example of loading Npub, for ccw=32

An example of loading a 128-bit Npub is shown in Fig. 5.3.

Figures 5.4 and 5.5 illustrate examples of loading 120-bit AD and 104-bit message respectively.

The same applies for hash messages with the exception of the empty hash message ϵ. Figure 5.6 shows the handshaking for an empty hash message.

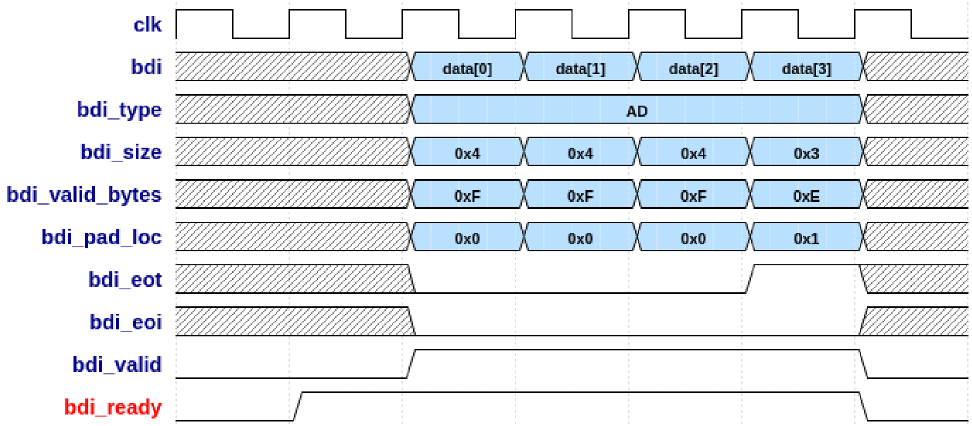

Figure 5.4: Handshake example of loading AD, for ccw=32, with data[3] containing the last 3 bytes of AD

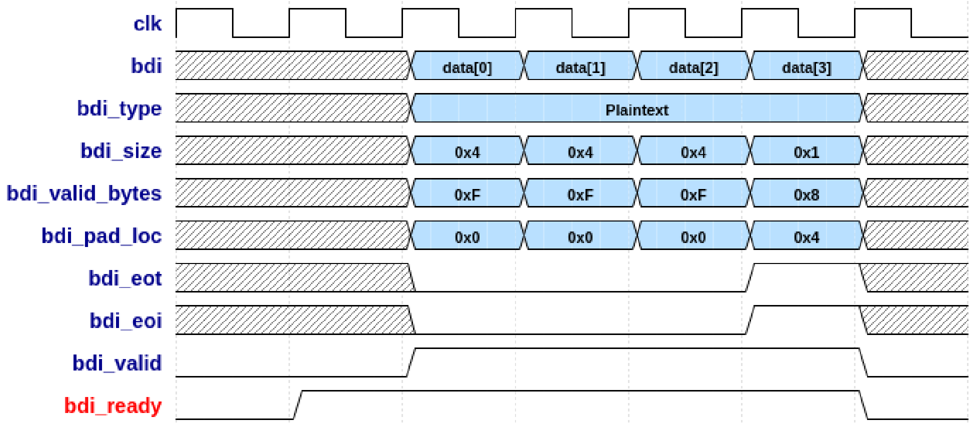

Figure 5.5: Handshake example of loading a message, for ccw=32, with data[3] containing the last 1 byte for encryption mode

Finally, an example of a handshake for authentication is shown in Fig. 5.7. For every decryption operation, the PostProcessor will set the `msg_auth_ready` signal to indicate its readyness to accept verification result. The result should be provided by CryptoCore via `msg_auth` and indicated that it's valid by `msg_auth_valid`.

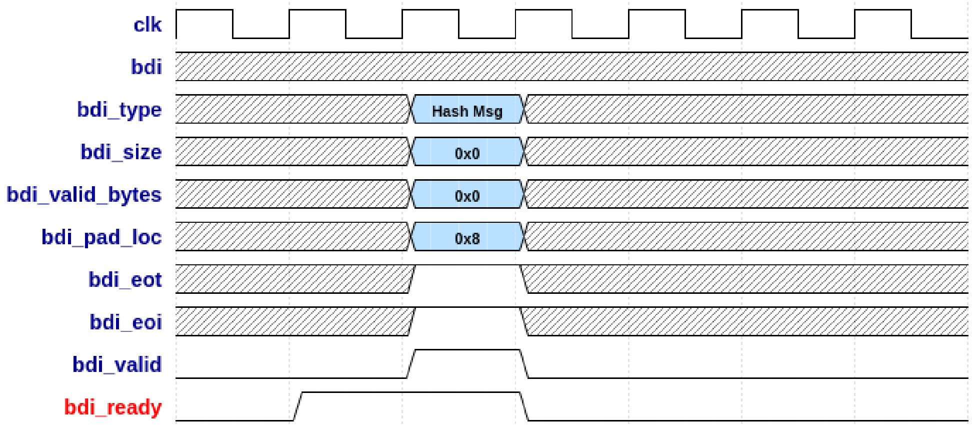

Figure 5.6: Handshake example of an empty message for ccw=32

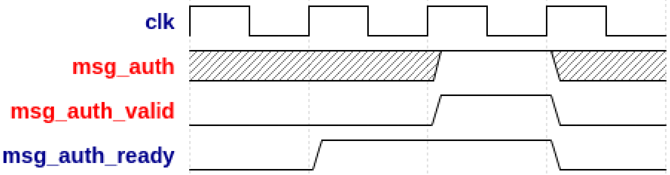

Figure 5.7: Handshake example for message authentication

5.4 Design Procedure

It is recommended that you start the development of the CryptoCore, specific to a given authenticated cipher, by using the code provided in the Development Package, in the folder

```
$root/hardware/LWC_rtl
```

In particular, the appropriate connections among the CryptoCore, the PreProcessor, the PostProcessor, and the HeaderFIFO modules are already specified in this code. A designer only needs to develop the CryptoCore Datapath and the CryptoCore Controller. The development of the CryptoCore is left to individual designers and can be performed using their own preferred design methodology. Typically, when using a traditional RTL (Register Transfer Level) methodology, the CryptoCore Datapath is first modeled using a block diagram, and then translated to a hardware description language (VHDL or Verilog HDL). The CryptoCore Controller is then described using an algorithmic state machine (ASM) chart or a state diagram, further translated to HDL. An ASM chart of the CryptoCore Controller typically contains the following states/steps:

1. Idle
2. Load (Process) Key
3. Load (Process) Npub
4. Wait AD
5. Load (Process) AD
6. Load (Process) Data
7. Output Data
8. Process Tag
9. Output/Verify Tag
10. Init Hash
11. Empty Hash
12. Load (Process) Hash Message
13. Output Hash Value

Depending on the implemented cipher some of the wait states might be omitted and some of the processing states might be extended to multiple states. An example ASM chart for the CryptoCore Controller is shown Fig. 5.8. As description in its entirety is too complex; this ASM is only intended to give a brief overview. For a more detailed view, a well commented dummy core is provided.

Idle After a new instruction or after reset, the Controller should wait for the first block of data in the *Idle* state. The CryptoCore should monitor the `bdi_valid` and `key_valid` for the first input.

Key Update If `key_valid` is high, `key_update` indicates whether the current key requires an update. If it does, the controller changes the state to *Load_ Key*. The `key_ready` signal should be activated in this state if the CryptoCore is ready to receive. The deassertion of `key_update` indicates that the complete key has been transmitted. Alternatively, if a counter is already in use by design (e.g. an address counter), it can be used to keep track of the received words. After a new key is loaded, the CryptoCore returns to idle.

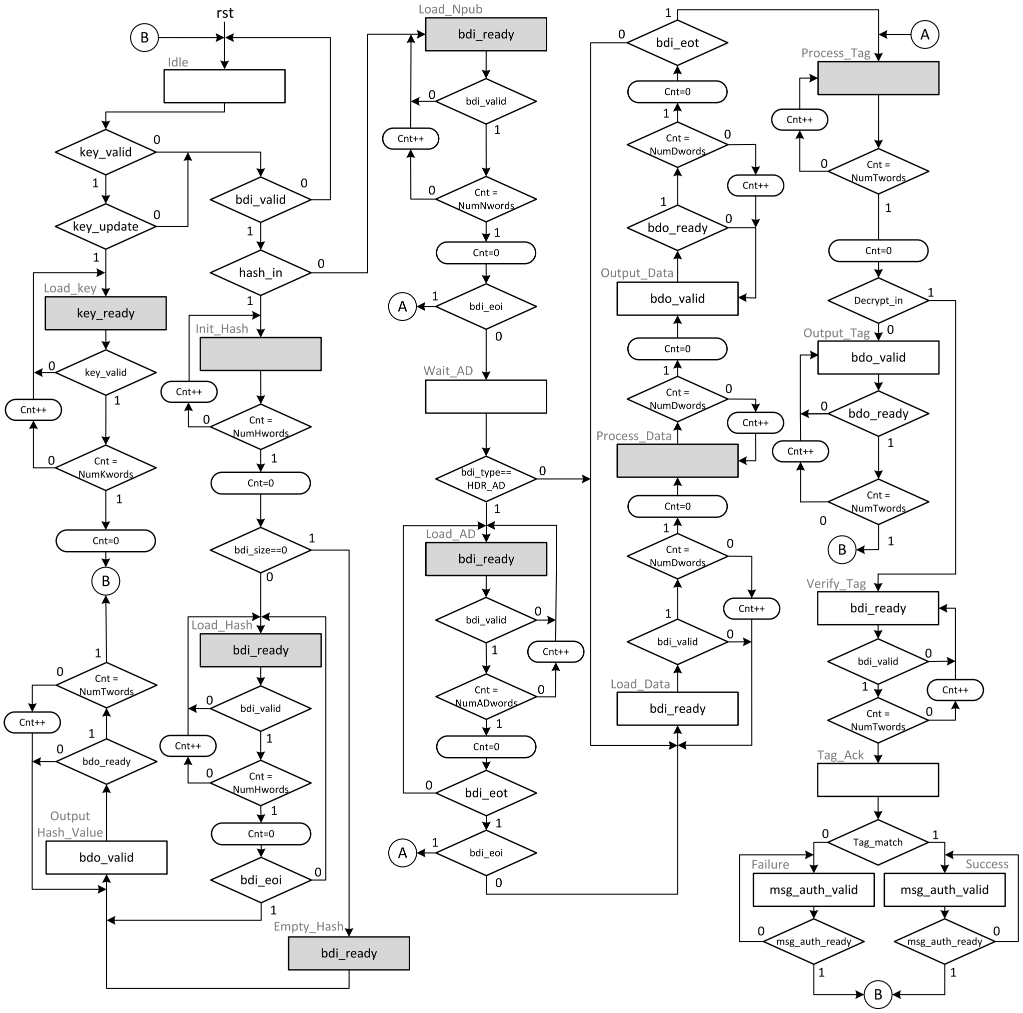

Figure 5.8: A typical Algorithmic State Machine (ASM) chart of the CryptoCore Controller. Each shaded state in this diagram may need to be replaced by a sequence of states in the actual implementation of a complex authenticated cipher.

AEAD or Hash If `bdi_valid` is high, the controller checks if a hash value generation or an authenticated encryption/decryption takes place, by inspecting the signal `hash_in`. An authenticated encryption/decryption starts with loading the Npub in the *Load_ Npub* state. The calculation of a hash value starts with the initialization in the *Init_ Hash* state.

Npub The `bdi_ready` signal should be activated in this state if the CryptoCore is ready to receive. Again, either a counter or the signal `bdi_eot` can used to determine if all words of Npub have been received.

AD After processing the Npub, the controller moves to *Wait_ AD* to decide whether there are Associated Data at all, and if so further to *Load_ AD* to load and process the Associated Data.

PT/CT In the *Load_ Data* state, the circuit waits until the input data is valid (`bdi_valid`=1), loads the data and then processes it in *Load_ Data*. Finally the corresponding plaintext or ciphertext is output.

Tag generation In the *Process_ Tag* state, the tag is calculated. Next, depending on the `decrypt_in` signal either the tag is output in the state *Output_ Tag*, or the calculated tag is compared against the received tag in *Verify_ Tag* state.

Hash The calculation of a hash value is similar: Depending on the cipher, the internal state is initialized. If the hash value of the empty string ϵ (`bdi_valid`=1 and `bdi_size`=0) is calculated, a single acknowledgment (`bdi_ready`=1 in the state *Empty_ Hash*) is needed. For an non empty input, the input data is loaded and processed in the state *Load_ hash*. Finally, the hash value is output in the state *Output_ hash_ value*. This state can be combined with the state *Output_ Tag* if both outputs share the same size.

Shortcuts and Extensions Depending on the algorithm, additional processing may be required for the last block of data. This block can be determined using the end-of-type input (`bdi_eot`). This signal is also used to move to the processing of the next data type. The `bdi_eoi` indicates, that no further input is expected. In this case Ⓐ the controller can progress to the *Process_ Tag* state directly.

5.5 Dummy Authenticated Cipher

An example design of the ligtweight CryptoCore, corresponding to a dummy authenticated cipher, dummy_lw, is provided as a part of our distribution.

This example is aimed at presenting the behavior of the Pre- and Post-processors for a typical CryptoCore. The dummy authenticated cipher is specified using the following equations:

$$AD = AD_1, AD_2, ..., AD_{n-1}, AD_n \tag{5.1}$$

$$PT = PT_1, PT_2, ..., PT_{m-1}, PT_m \tag{5.2}$$

$$CT = CT_1, CT_2, ..., CT_{m-1}, CT_m \tag{5.3}$$

$$CT_i = PT_i \oplus i \oplus Key \oplus Npub \tag{5.4a}$$
$$PT_i = CT_i \oplus i \oplus Key \oplus Npub \tag{5.4b}$$

for $i = 1..m - 1$.

$$CT_m = Trunc(PT_m \oplus i \oplus Key \oplus Npub, PT_m) \tag{5.5a}$$
$$PT_m = Trunc(CT_m \oplus i \oplus Key \oplus Npub, CT_m) \tag{5.5b}$$

$$Tag = Key \oplus Npub \oplus Len \oplus \bigoplus_{i=1}^{n-1} AD_i \oplus Pad(AD_n) \oplus \bigoplus_{i=1}^{m-1} PT_i \oplus Pad(PT_m) \tag{5.6}$$

where,

- PT_i and CT_i are the plaintext and ciphertext blocks, respectively,
- AD_i are the associated data blocks,
- $AD_{block\_size} = PT_{block\_size} = CT_{block\_size} = 128$ bits
- $Pad(\cdot)$ represents a 10^* padding operation applied to the last AD and/or the last plaintext block,
- $Pad(AD_n) = AD_n$ **if** $len(AD_n) = block\_size$ **else** $AD_n||10^*$
- $Pad(PT_m) = PT_m$ **if** $len(PT_m) = block\_size$ **else** $PT_m||10^*$

- $Trunc(X, Y)$ truncates X to the size of Y,
- i is the 128-bit block number,
- Key is a 128-bit key,
- $Npub$ is the 96 bit Public message number (nonce),
- Len = 64-bit associated data length (in bits) || 64-bit plaintext length (in bits).

For an XOR operation with inputs of different sizes, the smaller operands are appended with zeros to have the same length as the longest operand. The result has the length of the longest operand.

The design of the controller used in our dummy cores is based on the ASM chart discussed in the previous section.

The code of the Cipher Core is developed to work correctly with ccw=ccsw=8, 16, and 32.

5.6 Dummy Hash

An example design of the lightweight hash function, corresponding to a dummy hash implementation, dummy_lw, is provided as a part of our distribution.

$$HASH\_VALUE = \bigoplus_{i=1}^{m-1} HASH\_MSG_i \oplus Pad(HASH\_MSG_m) \quad (5.7)$$

The following parameters are used:

- $HASH\_MSG_{block\_size} = 256$ bits
- $Pad(HASH\_MSG_n) = HASH\_MSG_n$ **if** $len(HASH\_MSG_n) = block\_size$ **else** $HASH\_MSG_n||10^*$
- The empty string ϵ has $HASH\_VALUE = 0$.

The code of the CryptoCore is developed to work correctly with ccw=ccsw=8, 16, and 32.

6 Verification

6.1 Test vector generation (*cryptotvgen*)

The Python script called *cryptotvgen* and accompanying examples provide a framework to generate test vectors for any authenticated cipher based on the user's specified parameters. The script is located in the folder

```
$root/software/cryptotvgen/cryptotvgen
```

and the examples of calling it with parameters specific to multiple authenticated ciphers in the folder

```
$root/software/cryptotvgen/examples
```

The framework relies on the reference implementations of authenticated ciphers and hash function (including, but not limited to NIST LWC candidates), which can be placed in the following folders.

```
$root/software/dummy_lwc_ref/crypto_aead
$root/software/dummy_lwc_ref/crypto_hash
```

6.1.1 Setup

In order to run *cryptotvgen*, you need to have the following installed in your system:

- gcc
- Python v3.6+

The below instructions describe how to install and configure these packages from scratch.

Linux

The following instructions assume the use of Ubuntu v18.04 or above for Linux. The latest version of MSYS2 is assumed for Windows.

```
### Install required tools
sudo apt install gcc python3 python3-pip;
pip3 install request;

### For MSYS2 user, python3-cffi package may not be available
### so the following instruction can be referred as a workaround.
pacboy -S libcrypt-devel
pacboy -S libffi-devel
CFLAGS=-I/usr/lib/libffi-3.2.1/include pip install cffi

### Install wheel
python3 -m pip install -e $root/software/cryptotvgen/.

### Test that the program has been installed
### by calling help
cryptotvgen -h

### Uninstalling cryptotvgen
python3 -m pip uninstall cryptotvgen
```

6.1.2 Compiling shared libraries

```
### The following instruction provides a step-by-step guide into preparing a shared
    library for use with cryptotvgen using prepare_src utility. The instruction
    assumes that all build environment is setup correctly.

# Downloads SUPERCOP and make LWC candidates.
# Downloaded files and built shared libraries are located at ~/.cryptotvgen
cryptotvgen --prepare_libs

# If SUPERCOP is already downloaded. candidate_dir can be the location of SUPERCOP
    or any other
# directory that contains crypto_aead or crypto_hash format.
cryptotvgen  --prepare_libs <algorithm_name> --candidates_dir=/path

# Example
cryptotvgen --prepare_libs dummy --candidates_dir=$root/software/dummy_lwc_ref
```

6.1.3 Adding a new library

A new software library, corresponding to a new authenticated cipher, can be added to our framework as long as it follows SUPERCOP software API. The user simply needs to place the code using the same structure as SUPERCOP

(<algorithm_class>/<algorithm_name>/<implementation_name>). Then, follow instructions provided in Section 6.1.2.

6.1.4 Generating test vectors

It is recommended that the user understands the arguments of *cryptotvgen*, in order to properly create test vectors for the design under verification. The arguments to be used are the function of

- algorithm
- parameters of the algorithm (e.g., key size, block size)
- phase of verification.

As a result, basic knowledge of the target design, including the parameters of the algorithm and implementation, are required. While it is possible to generate test vectors using pure shell command syntax, this process is likely to be error prone due to the large number of available options. Instead, we recommend that the user create a Python script that utilizes *cryptotvgen* as a third party library in Python and then calls it using cryptotvgen(args).

Various examples of such Python scripts can be found in

`$root/software/cryptotvgen/examples`

An example of generating a set of test vectors for dummy_lw is shown below:

```
### Generate test vectors for dummy_lw
cd $root/software/cryptotvgen/examples

# Create test vectors for dummylw
python3 dummy_lw.py
```

The user is encouraged to use the files

`$root/software/cryptotvgen/examples/dummy_lwc_*.py`

as templates and a starting point to create the customized script for the targeted design.

The provided template contains a list of possible options for the majority of use cases. It must be noted, however, that the user must take into account the specific characteristics of the algorithm and design when generating these test vectors. Providing as much coverage as possible ensures that the design can withstand a real-world usage.

In particular, a typical process of verifying the functionality of an authenticated cipher module includes the following phases, devoted to the verification of:

1. Single AD and Message/Ciphertext Block

2. Random Inputs with Custom Selected Sizes

3. Empty Message, Empty AD, Basic Message/ID Sizes

4. Randomly Generated Test Vectors with Varying AD, Message, and Ciphertext Lengths.

Test vectors for these phases can be generated using the *cryptotvgen* options:

1. --*gen_single*

2. --*gen_custom*

3. --*gen_hash*

4. --*gen_test_routine*

5. --*gen_test_combined*

6. --*gen_random*

7. --*gen_benchmark*

respectively, as illustrated in `gimli24v1.py`.

The choice of one of these phases can be accomplished simply by uncommenting the respective line of the script, e.g.,

```
## PHASE 3:
    args = basic_args + gen_test_routine
```

Please note that only for the --gen_single option, the knowledge of the key, Npub, Nsec, AD, and Data sizes is required to generate test vectors. For all other cases, these sizes are inferred from the values of basic arguments (basic_args), such as --io, --key_size, --block_size, etc., which need to be specified only once.

After the analysis using these most commonly used sets of option, the designer has the flexibility of generating his own verification strategy, based on the detailed knowledge and understanding of options of *cryptotvgen*. This additional verification may be necessary to cover the full functionality offered by the specific algorithm, especially in case of encrypting and decrypting multiple inputs of various sizes and internal compositions.

6.2 Hardware Simulation

Once test vectors are generated, copy them into your simulation folder or update generic properties in LWC_TB.vhd to their paths appropriately.

Table 6.1: LWC_TB.vhd Generics

Value	**Description**
G_MAX_FAILURES	Max test vector failures before halting simulation
G_TEST_MODE	See the Test modes tables
G_TEST_IPSTALL	Controls when PDI stalls in TEST_MODE 1 & 2
G_TEST_ISSTALL	Controls when SDI stalls in TEST_MODE 1 & 2
G_TEST_OSSTALL	Controls when DO stalls in TEST_MODE 1 & 2
G_LOG2_FIFODEPTH	Controls the FIFO Depth of for FPDI, FSDI, and FDO
G_PERIOD	Clock period during simulation
G_FNAME_PDI	Path to the PDI test vectors
G_FNAME_SDI	Path to the SDI test vectors
G_FNAME_DO	Path to the DO test vectors
G_FNAME_LOG	Output log file destination path
G_FNAME_TIMING	Log file when in TEST_MODE 4
G_FNAME_TIMING_CSV	CSV log file when in TEST_MODE 4
G_FNAME_FAILED_TVS	Log of test vectors that failed
G_FNAME_RESULT	Contains status of simulation run

Simulation is performed until the end-of-file is reached or the G_MAX_FAILURES threshold is hit by mismatches between expected and actual output. A clock signal is deactivated when either of these two conditions is met.

Finally, in the practical experimental testing of any module, there is no guarantee that the input source will be ready with the new input whenever the module attempts to read it. Similarly, the destination circuit may not be always ready to receive the new output. These conditions must be comprehensively verified using simulation, before the experimental testing is attempted.

In our testbench, these conditions can be accomplished using the features of stalling input and stalling output. The rate at which the data is stalled can be configured using TEST_IPSTALL (public input stall), TEST_IPSTALL (secret input stall) and TEST_OSTALL (output stall), expressed in clock cycles.

TEST_MODE 4 was added in release v1.1.0 to support Measurement Mode. This mode is intended to aid designers with the verification of formulas for execution time. In this mode results are logged into a text file and csv file. The path of these files can be set by G_FNAME_TIMING and G_FNAME_TIMING_CSV

These settings will only become active if TEST_MODE is set to the value shown in Table 6.2.

Table 6.2: Test modes

Value	Description
0	No stall
1	Input & Output stall test
2	Input only stall test
3	Output only stall test
4	Measurement Mode

Finally, it must be stressed that the aforementioned verification is paramount to ensuring that the design can withstand a real-world usage, where the intermittent data transmission is very common. At the very least, the user should ensure that the design under verification is successfully validated when *TEST_MODE* is set to 1.

6.3 Hardware Testing

6.3.1 UART based Framework

An universal UART wrapper can be found at [7]. It contains a python script to parse the generated `pdi.txt`, `sdi.txt`, and `do.txt`, and send them to a UART. A VHDL module handles the UART communication and provides the `pdi`, `sdi`, and `do` ports. Figure 6.1 shows an example block diagram. This framework focuses on functional verification.

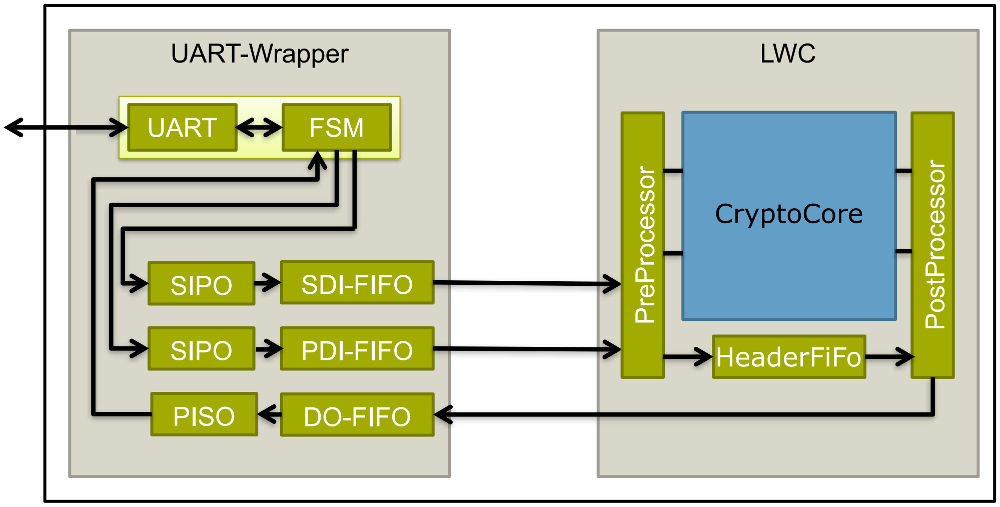

Figure 6.1: Example block diagram for functional verification.

6.3.2 Pynq based Frameworks

The framework from [8] and its extended version from [9] comprise an open source, simple plug and play framework which enables testing of implementations of cryptographic algorithms on a physical System on Chip (SoC) hardware, namely the PYNQ-Z1 board. It is compatible with the CAESAR Hardware API and also with the LWC API. In addition to functional verification, the framework measures the run time, power and energy consumption, and allows for verification of the maximum clock frequency on real hardware.

The Processing System (PS) of Zynq SoC runs cryptotvgen to generate test vectors. They are then send to the Programmable Logic (PL) and the results back read back, with the use of the Xilinx Direct Memory Access (DMA) to AXI4-Stream (AXIS) controllers. Run time of the core itself and including the overhead required to send data to and from the LWC core through DMA is measured through two hardware timers. It uses the XXBX Power Shim [10] and the Xilinx XADC of the SoC to measure power consumption. It supports on-chip power measurements and determining the maximum clock frequency using experimental testing.

This framework has been used successfully to locate errors in the HDL code of CAESAR candidates [8,9], preventing the corresponding implementations from running properly on the board. Even though the generation of primary timing and resource utilization results does not require experimental testing, the detected errors and the follow-up changes in the code may have influence on the final results. Additionally, experimental measurements of power consumption and maximum clock frequency can be used to verify the accuracy of the respective FPGA tools, and verifying the validity of assumptions used by these tools.

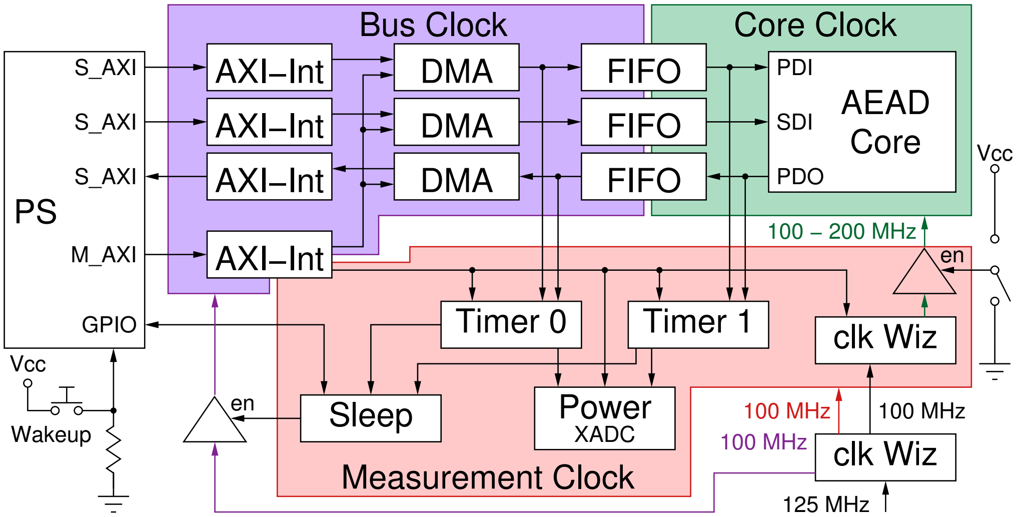

Figure 6.2: Block diagram for power and performance evaluation.

6.3.3 Side-Channel Analysis Framework

The Flexible, Opensource workBench fOr Side-channel analysis (FOBOS) is designed to be an inexpensive side channel analysis setup that includes a complete software package with programs for data acquisition and data analysis. In order to evaluate side-channel leakage of hardware platforms, FOBOS uses off-the shelf FPGA boards as control and device under test (DUT). Starting with version 2, to be released in Fall 2019, it supports the LWC API. Figure 6.3 shows the block diagram of FOBOS 2. The control board is a Basys3, which communicates with the PC via USB serial, sends test vectors to the DUT, provides the clock for the DUT and a trigger for the oscilloscope. FOBOS provides a wrapper for the "Function Core" to enable users to simply plug in their LWC core as shown in Fig. 3.1.

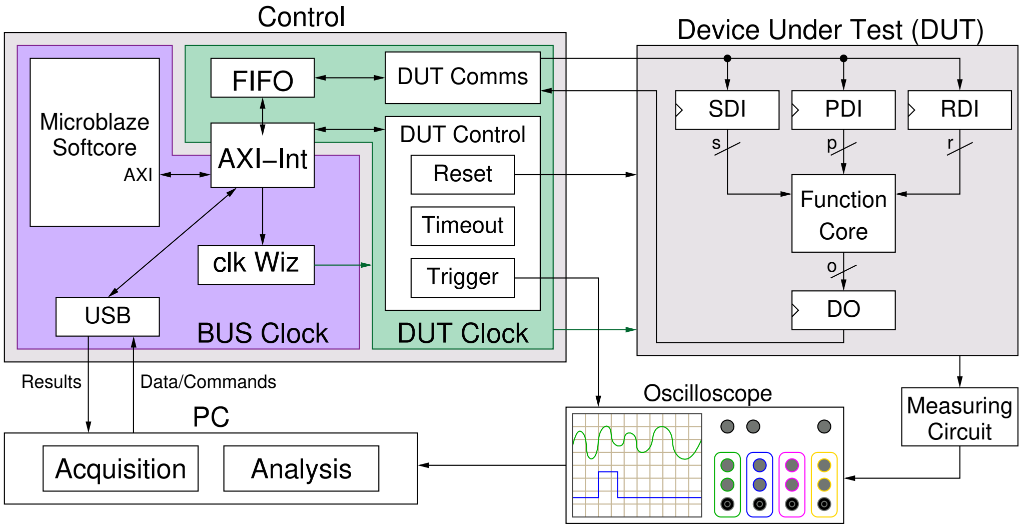

Figure 6.3: FOBOS 2 Block diagram.

Figure 6.4 shows a typical FOBOS 2 setup consisting of a Basys3 board as control, a CW305 Artix FPGA Target Board as DUT and a Picoscope for collecting the measurements.

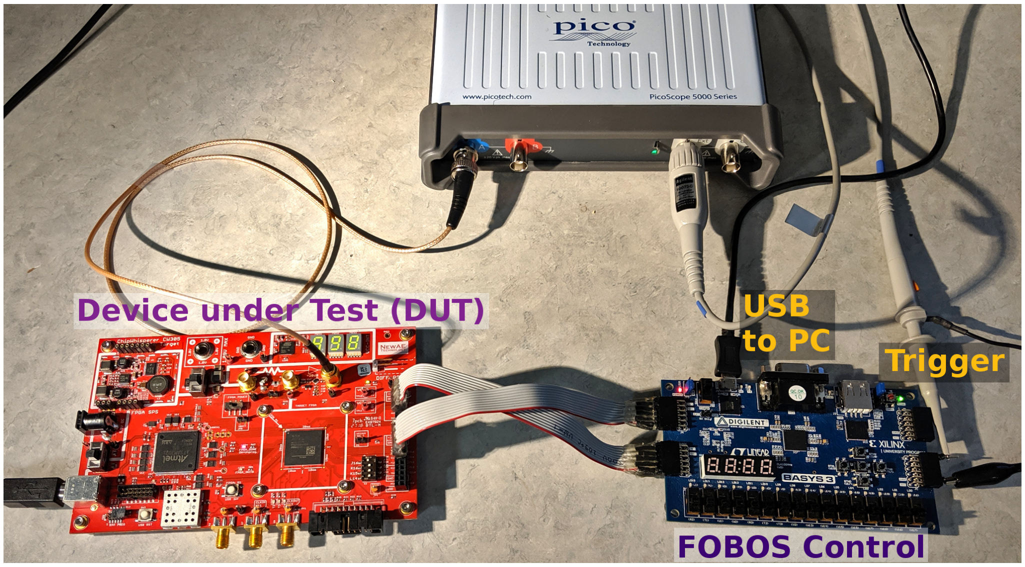

Figure 6.4: Typical FOBOS 2 setup.

7 Generation and Publication of Results

Generation of results is possible for the LWC core and the CryptoCore. We recommend generating results primarily for the LWC cores. Benchmarking and reporting results for FPGAs should be performed using the most-recent low-cost families of FPGA devices from at least two major vendors, Intel and Xilinx. For Intel, such families include: Cyclone V and Cyclone 10 FPGAs and Cyclone V SoC FPGAs; for Xilinx: Artix-7 and Spartan-7 FPGAs, and Zynq-7000 All Programmable SoCs. The most recent versions of tools from the respective vendors should be used. Only final results obtained after placing and routing should be reported. In terms of optimization of tool options, for Xilinx FPGAs and SoCs, we recommend generating results using Minerva [11]. In case of ASICs, state-of-the-art libraries of standard cells should be used. Comprehensive results, generated after the respective submission deadlines for the HDL code, are expected to be made publicly available in the ATHENa Database of Results for Authenticated Ciphers [12] or an equivalent or extended database of results, focused on LWC candidates.

8 Differences Compared to the CAESAR Hardware API Development Package

Major differences between the proposed Development Package for Hardware Implementations Compliant with the Hardware API for Lightweight Cryptography and the Development Package for Hardware Implementations Compliant with the CAESAR Hardware API, defined in [5], are as follows:

8.1 Functionality

8.1.1 API

In terms of the Minimum Compliance Criteria: a) One additional configuration, encryption/decryption/hashing, has been added on top of the previously supported configuration: encryption/decryption. b) On top of the maximum sizes of AD/plaintext/ciphertext already supported in the CAESAR Hardware API, two additional maximum sizes, $2^{16} - 1$ and $2^{50} - 1$, have been added.
In terms of the Interface: An additional optional output, `do_last`, has been added to the Data Output ports.
In terms of the Communication Protocol: a) In the Instruction/Status, an additional opcode value, representing hash function, has been added. b) In the Segment Header word, two additional Segment Type values, representing Hash Message and Hash Value, have been added.

8.1.2 Support for Hashing

Hashing is fully supported. The PreProcessor has a new output signal `hash` to indicate, that the CryptoCore should execute a hash instruction. Correspondingly, there is a new type encoding "0111" for `bdi_type` to indicate, that the `bdi` contains data to be hashed. An empty hash is indicated by `bdi_valid` set to "1" and `bdi_size` set to zero. The PreProcessor expects an acknowledgment read. The CryptoCore must set `bdi_ready` to "1" for one cycle. The cryptotvgen also supports the generation of hash test vectors.

8.1.3 Deprecated Features

The following features are not supported:

- Tag comparison in PostProcessor.

8.1.4 Added Features

- Features added in version 1.1.0
  - Enhanced compatibility with VHDL Standards IEEE 1076-1993, IEEE 1076-2002, and IEEE 1076-2008.
  - Improved cryptotvgen for easier install and use. See README.md.
  - Fixed incorrect EOI flag when a hash message is empty
  - Timing test mode in LWB_TB.vhd enabled by setting G_TEST_MODE=4. This test mode reports cycles required for given message sizes outputting this data into a log and csv file specified by G_FNAME_TIMING and G_FNAME_TIMING_CSV respectively.
- Prior feature differences compared to the CAESAR Hardware API:
  - Support different (w, ccw) and (sw, ccsw) combinations. The following new combinations are supported: (32, 32), (32, 16), and (32, 8). They can be used independently for w and sw.
  - The PostProcessor sets unused bytes in `bdo` to zero.
  - Multiple input and output segments for Ciphertext, Plaintext, and Hash Message are supported for lightweight implementations.

8.2 Internal Structure

The VHDL code of the PreProcessor and Postprocessor had a major code review to improve functionality, readability and code coverage. The top-level module AEAD was renamed to LWC. The module CipherCore was renamed to CryptoCore.

8.2.1 Configuration

The configuration was reordered: The CryptoCore (including the widths of the interface to the PreProcessor and PostProcessor) is configured in `design_pkg.vhd`. The `NIST_LWAPI_pkg.vhd` contains all constants and functions for the PreProcessor and PostProcessor. Additionally the widths of `pdi`, `sdi` and `do` are configured here.

The generics G_W and G_SW in LWC are replaced by the constants W and SW. The configuration parameters PW and SW are replaced by CCW and CCSW.

8.3 Implementer's Guide

The Implementer's Guide was rewritten to reflect the changes. Additionally, some minor issues were fixed or clarified.

Appendix A: cryptotvgen help

```
cryptotvgen -h
usage: cryptotvgen [--candidates_dir <PATH/TO/CANDIDATES/SOURCE/DIRECTORY>]
                   [--lib_path <PATH/TO/LIBRARY/DIRECTORY>]
                   [--aead <ALGORITHM_VARIANT_NAME>]
                   [--hash <ALGORITHM_VARIANT_NAME>] [--gen_random N]
                   [--prepare_libs [<variant_prefix> [<variant_prefix> ...]]]
                   [--supercop_version SUPERCOP_VERSION] [--gen_benchmark]
                   [--gen_custom_mode MODE] [--gen_custom Array]
                   [--gen_hash BEGIN END MODE]
                   [--gen_test_combined BEGIN END MODE]
                   [--gen_test_routine BEGIN END MODE]
                   [--gen_single MODE KEY NPUB NSEC AD PT] [-h] [--verify_lib]
                   [-V] [-v] [--io PUBLIC_PORTS_WIDTH SECRET_PORT_WIDTH]
                   [--key_size BITS] [--npub_size BITS] [--nsec_size BITS]
                   [--tag_size BITS] [--message_digest_size BITS]
                   [--block_size BITS] [--block_size_ad BITS]
                   [--block_size_msg_digest BLOCK_SIZE_MSG_DIGEST]
                   [--ciph_exp] [--ciph_exp_noext] [--add_partial]
                   [--msg_format SEGMENT_TYPE [SEGMENT_TYPE ...]] [--offline]
                   [--min_ad BYTES] [--max_ad BYTES] [--min_d BYTES]
                   [--max_d BYTES] [--max_block_per_sgmt COUNT]
                   [--max_io_per_line COUNT] [--pdi_file FILENAME]
                   [--sdi_file FILENAME] [--do_file FILENAME]
                   [--dest PATH_TO_DEST] [--human_readable] [--cc_hls]
                   [--cc_pad_enable] [--cc_pad_ad PAD_AD_MODE]
                   [--cc_pad_d PAD_D_MODE] [--cc_pad_style PAD_STYLE]

Test vectors generator for NIST Lightweight Cryptography candidates.

:::::Path specifiers::::
  Not required if using ‘--prepare_libs‘ in automatic mode (see below and README)

  --candidates_dir <PATH/TO/CANDIDATES/SOURCE/DIRECTORY>
                        Relative or absolute path to the top _directory_ where the ‘
    crypto_aead ‘crypto_hash‘ folders candidates directory,
                        Source directory structure in this folder must follow
    SUPERCOP directory structure.
                        (default: None, which will use $HOME/.cryptotvgen)
  --lib_path <PATH/TO/LIBRARY/DIRECTORY>
                        Relative or absolute path to the top _directory_ where ‘
    crypto_aead‘ and ‘crypto_hash‘ folders with the dynamic shared libraries (*.so
    or *.dll) reside.
                        e.g. ‘../software/dummy_lwc_ref/lib‘
                        (default: None, which means if candidates_dir option is
    specified will use ‘candidates_dir‘/lib
                                and if neither candidates_dir nor lib_path are
    specified will use  $HOME/.cryptotvgen/lib)

:::::At least one of these parameters are required::::
  Library name specifier::

  --aead <ALGORITHM_VARIANT_NAME>
                        Name of a the variant of an AEAD algorithm for which to
    generate test-vectors, e.g. gimli24v1
                        Note: The library should have been be generated previously
    by running in ‘--prepare_libs‘. (default: None)
  --hash <ALGORITHM_VARIANT_NAME>
                        Name of a the variant of a hash algorithm for which to
    generate test-vectors, e.g. asconxofv12
                        Note: The library should have been be generated previously
    by running in ‘--prepare_libs‘. (default: None)

:::::Test Generation Parameters::::
  Test vectors generation modes (use at least one from the list
  below)::
  Common notation and convetions:
  AD - Associated Data
  DATA - Plaintext/Message or Ciphertext
  PT - Plaintext/Message
  CT - Ciphertext
  HASH - Message to be hashed
  HASH_TAG - Message Digest
  (*)_LEN - Length of data (*) type, i.e. AD_LEN.
  Operation - 0: encryption, 1: decryption
  H* - a string composed of multiple repetitions of the hexadecimal
       digit H (the number of repetitions is determined by the size
       of a given argument)
       All lengths are expressed in bytes.

  For Boolean arguments, 0 can be used instead of False,
  and 1 can be used instead of True.

  --gen_random N        Randomly generates N test vectors with
                        varying AD_LEN, PT_LEN, and operation (For use only with
    AEAD) (default: 0)
  --prepare_libs [<variant_prefix> [<variant_prefix> ...]]
                        Build dynamically shared libraries required for testvector
    generation.
                        If one or more <variant_prefix> arguments are given, only
    build variants
                        whose name starts with either of these prefixes, otherwise
    will build
                        all libraries.
                        e.g. ‘--prepare_libs ascon‘ will only build all AEAD and
    hash variants
```

```
                      of "ascon*"

                      Automatic mode: If no ‘--candidates_dir‘ option is present
  it will
                                      download and extract reference
  implementations from SUPERCOP.
                      Subfolder mode: If ‘--candidates_dir‘ is specified, only
  build
                                      libraries found in sources directories of ‘
  candidates_dir‘
                                      (uses SUPERCOP directory structure)
                      (default: None)            See also ‘--supercop_version‘
--supercop_version SUPERCOP_VERSION
                      ’SUPERCOP version to download and use.
                      Either use specific version with ‘YYYYMMDD‘ format or use ‘
  latest‘
                      to automatically determine the latest available version from
   the SUPERCOP website. (default: latest)
--gen_benchmark       This mode generates several the following sets of test
  vectors
                      1) generic_aead_sizes_new_key: encryption and decryption of
  the following sizes
                          using a new key every time. Also
  generic_aead_sizes_reuse_key.
                          Format: (ad,PT/CT)
                          (5*--block_size_ad//8,0), (4*--block_size_ad//8,0),
  (1536,0), (64,0), (16,0)
                          (0,5*--block_size//8), (0,4*--block_size//8), (0,1536),
  (64,0), (0,16)
                          (5*--block_size_ad//8,5*--block_size), (4*--
  block_size_ad//8,4*--block_size),
                          (1536,1536), (64,64), (16,16)
                      2) basic_hash_sizes: (0, 16, 64, 1536, 4*--
  block_size_msg_digest//8,
                                           5*--block_size_msg_digest//8)
                      3) kats_for_verification: for i in range 0 to (2*--
  block_size_ad//8)-1
                                                    for x in range 0 to 2*--
  block_size//8)-1
                                                        tests += (i,x)
                                          Encryption only
                      4) blanket_hash_test: 0 to (4*--block_size_msg_digest//8) -1
                      5) pow_*: Several sets of test vectors that are only one
  message for
                                for each combination of possible values for basic
  sizes

                      Additional arguments to provide --aead, --block_size, and --
  block_size_ad.

                      Optional arguments --hash and --block_size_msg_digest allow
  for the generation
                      of the hash test vectors
                       (default: False)
  --gen_custom_mode MODE
                        The mode of test vector generation used by the --gen_custom
    option.

                        Meaning of MODE values:
                            0 = All random data
                            1 = Fixed test values.
                                Key=0xFF*, Npub=0x55*, Nsec=0xDD*,
                                AD=0xA0*, PT=0xC0*, HASH=0xFF*
                            2 = Same as option 1, except an input is now a running
                                value (each subsequent byte is a previous byte
                                incremented by 1).
                         (default: 0)
  --gen_custom Array    Randomly generate multiple test vectors, with each test
    vector
                        specified using the following fields:
                           NEW_KEY (Boolean), DECRYPT (Boolean), AD_LEN, PT_LEN or
                           HASH_LEN, HASH (Boolean)
                           ":" is used as a separator between two consecutive test
                           vectors.

                        Example:
                        --gen_custom True,False,0,20,False:0,0,0,24,True

                        Generates 2 test vectors. The first vector will
                        create a new key and perform an encryption with a dataset
    that
                        has AD_LEN and PT_LEN of 0 and 20 bytes, respectively.  The
                        second vector performs a HASH on a message with HASH_LEN of
    24
                        bytes. (default: None)
  --gen_hash BEGIN END MODE
                        This mode generates 20 test vectors for HASH only.
                        The test vectors are specified using the following array:
                           [NEW_KEY (boolean),  # Ignored due to hash operation
                            DECRYPT (boolean),  # Ignored due to hash operation
                            AD_LEN,             # Ignored due to hash operation
                            PT_LEN,
                            HASH   (boolean)]:
                        The following parameters are used:
                            [False ,    False,       0,         0        ,   True],
                            [False ,    False,       0,         1        ,   True],
                            [False,     False,       0,         2        ,   True],
                            [False ,    False,       0,         3        ,   True],
                            [False,     False,       0,         4        ,   True],
                            [False ,    False,       0,         5        ,   True],
                            [False,     False,       0,         6        ,   True],
                            [False ,    False,       0,         7        ,   True],
                            [False,     False,       0,         bsd-2    ,   True],
                            [False ,    False,       0,         bsd-1    ,   True],
                            [False,     False,       0,         bsd      ,   True],
                            [False ,    False,       0,         bsd+1    ,   True],
                            [False,     False,       0,         bsd+2    ,   True],
                            [False ,    False,       0,         bsd*2    ,   True],
                            [False,     False,       0,         bsd*2+1  ,   True],
                            [False ,    False,       0,         bsd*3    ,   True],
                            [False,     False,       0,         bsd*3+1  ,   True],
                            [False ,    False,       0,         bsd*4    ,   True],
                            [False,     False,       0,         bsd*4+1  ,   True],
                            [False ,    False,       0,         bsd*5    ,   True],
                            [False,     False,       0,         bsd*5+1  ,   True]]

                        where,
                            bsa is the associated data block size (block_size_ad = 0
      for
                            hash), and
                            bsd is the data block size (block_size = # of bytes of
                            message to hash).

                        Note that sdi.txt will have a header, but no generated keys.
                        Also, key_id = 0 for all hash test vectors.

                        BEGIN (min=1,max=22) determines the starting test number.
                        END (min=1,max=22) determines the ending test number.
                        MODE determines the test vector generation mode, where
                            0 = All random data
                            1 = Fixed test values.
                                HASH=0xF0*
                            2 = Same as option 1, except each input is now a running
                                value (each subsequent byte is a previous byte
                                incremented by 1).

                        Example:

                        --gen_hash 1 20 0

                        Generates tests 1 to 20 with MODE=0.

                        --gen_hash 5 5 1

                        Generates test 5 with MODE=1. (default: None)
  --gen_test_combined BEGIN END MODE

                        This mode generates 33 test vectors for the common sizes of
    AD and
                        PT that the hardware designer should, at a minimum, verify.
    It also
                        combines AEAD and hash test vectors into one set of test
                        vectors, which are interleaved as encrypt, decrypt, and hash
    .
                        The test vectors are specified using the following array:
                           [NEW_KEY (boolean),
                            DECRYPT (boolean),
                            AD_LEN,
                            PT_LEN,
                            HASH    (boolean)]:
                        The following parameters are used:
                            [True ,     False,      0,         0        ,   False],
                            [False,     True,       0,         0        ,   False],
                            [False,     True,       0,         0        ,   True],
                            [True ,     False,      1,         0        ,   False],
                            [False,     True,       1,         0        ,   False],
                            [False,     True,       0,         1        ,   True],
                            [True ,     False,      0,         1        ,   False],
                            [False,     True,       0,         1        ,   False],
                            [False,     True,       0,         2        ,   True],
                            [True ,     False,      1,         1        ,   False],
                            [False,     True,       1,         1        ,   False],
                            [False,     True,       0,         3        ,   True],
                            [True ,     False,      2,         2        ,   False],
                            [False,     True,       2,         2        ,   False],
                            [False,     True,       0,         4        ,   True],
                            [True ,     False,      bsa-1,     bsd-1    ,   False],
                            [False,     True,       bsa-1,     bsd-1    ,   False],
                            [False,     True,       0,         bsd-1    ,   True],
                            [True ,     False,      bsa,       bsd      ,   False],
                            [False,     True,       bsa,       bsd      ,   False],
                            [False,     True,       0,         bsd+1    ,   True],
                            [True ,     False,      bsa+1,     bsd+1    ,   False],
                            [False,     True,       bsa+1,     bsd+1    ,   False],
                            [False,     True,       0,         bsd+2    ,   True],
                            [True ,     False,      bsa*2,     bsd*2    ,   False],
                            [False,     True,       bsa*2,     bsd*2    ,   False],
                            [False,     True,       0,         bsd*2    ,   True],
                            [True ,     False,      bsa*2+1,   bsd*2+1  ,   False],
                            [False,     True,       bsa*2+1,   bsd*2+1  ,   False],
                            [False,     True,       0,         bsd*2+1  ,   True],
                            [True ,     False,      bsa*3,     bsd*3    ,   False],
                            [False,     True,       bsa*3,     bsd*3    ,   False],
                            [False,     True,       0    ,     bsd*3    ,   True]]

                        where,
                            bsa is the associated data block size (block_size_ad),
    and
                            bsd is the data block size (block_size).

                        Note: key_id = 0 for all hash test vectors.

                        BEGIN (min=1,max=33) determines the starting test number.
                        END (min=1,max=33) determines the ending test number.
                        MODE determines the test vector generation mode, where
                            0 = All random data
                            1 = Fixed test values.
                                Key=0xF*, Npub=0x5*, Nsec=0xD*,
                                Ad=0xA0*, PT=0xC0*, HASH=0xF*
                            2 = Same as option 1, except each input is now a running
                                value (each subsequent byte is a previous byte
                                incremented by 1).

                        Example:

                        --gen_test_combined 1 20 0
```

```
                      Generates tests 1 to 20 with MODE=0.

                      --gen_test_combined 5 5 1

                      Generates test 5 with MODE=1. (default: None)
--gen_test_routine BEGIN END MODE
                      This mode generates test vectors for the common sizes of AD
  and
                      PT that the hardware designer should, at a minimum, verify.
                      Only AEAD test vectors are generated, hashes are not
  generated.
                      The test vectors are specified using the following array:
                         [NEW_KEY (boolean),
                          DECRYPT (boolean),
                          AD_LEN,
                          PT_LEN,
                          HASH    (boolean)]:
                      The following parameters are used:
                          [True ,     False,      0,         0        ,   False],
                          [False,     True,       0,         0        ,   False],
                          [True ,     False,      1,         0        ,   False],
                          [False,     True,       1,         0        ,   False],
                          [True ,     False,      0,         1        ,   False],
                          [False,     True,       0,         1        ,   False],
                          [True ,     False,      1,         1        ,   False],
                          [False,     True,       1,         1        ,   False],
                          [True ,     False,      bsa,       bsd      ,   False],
                          [False,     True,       bsa,       bsd      ,   False],
                          [True ,     False,      bsa-1,     bsd-1    ,   False],
                          [False,     True,       bsa-1,     bsd-1    ,   False],
                          [True ,     False,      bsa+1,     bsd+1    ,   False],
                          [False,     True,       bsa+1,     bsd+1    ,   False],
                          [True ,     False,      bsa*2,     bsd*2    ,   False],
                          [False,     True,       bsa*2,     bsd*2    ,   False],
                          [True ,     False,      bsa*3,     bsd*3    ,   False],
                          [False,     True,       bsa*3,     bsd*3    ,   False],
                          [True ,     False,      bsa*4,     bsd*4    ,   False],
                          [False,     True,       bsa*4,     bsd*4    ,   False],
                          [True ,     False,      bsa*5,     bsd*5    ,   False],
                          [False,     True,       bsa*5,     bsd*5    ,   False]

                      where,
                          bsa is the associated data block size (block_size_ad),
  and
                          bsd is the data block size (block_size).

                      BEGIN (min=1,max=22) determines the starting test number.
                      END (min=1,max=22) determines the ending test number.
                      MODE determines the test vector generation mode, where
                          0 = All random data
                          1 = Fixed test values.
                              Key=0xF*, Npub=0x5*, Nsec=0xD*,
                              Ad=0xA0*, PT=0xC0*
```

```
                            2 = Same as option 1, except each input is now a running
                                value (each subsequent byte is a previous byte
                                incremented by 1).

                        Example:

                        --gen_test_routine 1 20 0

                        Generates tests 1 to 20 with MODE=0.

                        --gen_test_routine 5 5 1

                        Generates test 5 with MODE=1.
                         (default: None)
  --gen_single MODE KEY NPUB NSEC AD PT
                        Generate a single test vector based on the provided values
    of
                        all inputs expressed in the hexadecimal notation. (For use
    only
                        with AEAD)

                        Example:
                        --gen_single 0 5555 0123456 789ABCD 010204 08090A   #Encrypt
                        --gen_single 2 0 0 0 0 1212121                      #Hash

                        Note:
                        KEY, NPUB and NSEC must have size equal to the expected
                        value. Exception: NSEC is ignored --nsec_size is set to 0.
                        All arguments must contain an even number of hexadecimal
                        digits, e.g., 00 is valid; 0 is invalid.

                        IS_DECRYPT, KEY, NPUB, NSEC, AD parameters are ignored in
    HASH mode.
                         (default: None)

:::::Optional Parameters:::::
  Debugging options::

  -h, --help            Show this help message and exit.
  --verify_lib          This operation will verify the generated test vectors
                        via the decryption operation.

                        Note: This option provides an additional check against
    possible
                              mismatch of results between encryption and decryption
                              in the reference software.
                         (default: False)
  -V, --version         show program's version number and exit
  -v, --verbose         Verbose for script debugging purposes. (default: False)

:
  Algorithm and implementation specific options::

  --io PUBLIC_PORTS_WIDTH SECRET_PORT_WIDTH
```

```
                        Size of PDI/DO and SDI port in bits. (default: (32, 32))
  --key_size BITS       Size of key in bits (default: 128)
  --npub_size BITS      Size of public message number in bits (default: 128)
  --nsec_size BITS      Size of secret message number in bits (default: 0)
  --tag_size BITS       Size of authentication tag in bits (default: 128)
  --message_digest_size BITS
                        Size of message digest (hash_tag) in bits (default: 64)
  --block_size BITS     Algorithm's data block size (default: 128)
  --block_size_ad BITS  Algorithm's associated data block size.
                                This parameter is assumed to be equal to block_size
    if unspecified. (default: None)
  --block_size_msg_digest BLOCK_SIZE_MSG_DIGEST
                        Algorithm's hash data block size (default: None)
  --ciph_exp            Ciphertext expansion algorithm. When this option is set, the
     last
                        block will have its own segment. This is required for a
    correct
                        operation of the accompanied PostProcessor.

                        Currently, we assume that PAD_AD and PAD_D are both set to 4
                        when this mode is used.
                         (default: False)
  --ciph_exp_noext      [requires --ciph_exp]

                        Additional option for the ciphertext expansion mode. This
    option
                        indicates that the algorithm does not expand the ciphertext
                        (i.e., does not make the ciphertext size greater than the
    message
                        size) if the message size is a multiple of a block size. (
    default: False)
  --add_partial         [requires --ciph_exp]

                        For use with --ciph_exp flag. When this option is set, a
    PARTIAL
                        bit will be set to 1 in the header of a data segment
                        if the size of this segment is not a multiple of a block
    size.

                        Note: This option is required for algorithms such as
    AES_COPA
                         (default: False)

:
  Formatting options::

  --msg_format SEGMENT_TYPE [SEGMENT_TYPE ...]
                        Specify the order of segment types in the input to
    encryption and
                        decryption. Tag is always omitted in the input to encryption
    , and
                        included in the input to decryption. In the expected output
    from
```

```
                        encryption tag is always added last. In the expected output
    from
                        decryption only nsec and data are used (if specified).
                        Len is always automatically added as a first segment in the
                        input for encryption and decryption for the offline
    algorithms.
                        Len is not allowed as an input to encryption or decryption
    for
                        the online algorithms.

                        Example 1:
                        --msg_format npub tag data ad

                        The above example generates
                        for an input to encryption: npub, data (plaintext), ad
                        for an expected output from encryption: data (ciphertext),
    tag
                        for an input to decryption: tag, data (ciphertext), ad
                        for an expected output from decryption: data (plaintext)

                        Example 2:
                        --msg_format npub_ad data_tag

                        The above example generates
                        for an input to encryption:  npub_ad, data (plaintext)
                        for an expected output from encryption: data_tag (
    ciphertext_tag)
                        for an input to decryption: npub_ad, data_tag (
    ciphertext_tag)
                        for an expected output from decryption: data (plaintext)

                        Valid Segment types (case-insensitive):
                            npub    -> public message number
                            nsec    -> secret message number
                            ad      -> associated data
                            ad_npub -> associated data || npub
                            npub_ad -> npub || associated data
                            data    -> data (pt/ct)
                            data_tag -> data (pt/ct) || tag
                            tag     -> authentication tag

                        Note: no support for multiple segments of the same type,
                        separated by segments of another type e.g., header and
    trailer,
                        treated as two segments of the type AD, separated by the
    message segments

                         (default: ('npub', 'ad', 'data', 'tag'))
  --offline             Indicate that the cipher is offline, i.e., the length of AD
    and
                        DATA must be known before the encryption/decryption starts.
    If this
                        option is used, the length segment will be automatically
    added as
```

```
                        a first segment in the input to encryption and decryption.
                        Otherwise, the length segment will not be generated for
    either
                        encryption or decryption.
                         (default: False)
  --min_ad BYTES        Minimum randomly generated AD length (default: 0)
  --max_ad BYTES        Maximum randomly generated AD length (default: 1000)
  --min_d BYTES         Minimum randomly generated data length (default: 0)
  --max_d BYTES         Maximum randomly generated data length (default: 1000)
  --max_block_per_sgmt COUNT
                        Maximum data block per segment (based on --block_size)
    parameter (default: 9999)
  --max_io_per_line COUNT
                        Maximum data length in multiples of I/O width in a data line
     of test
                        file. This option helps readability when a test vector is
    large.

                        Example:
                        If a user wants to limit a vector representation of data in
    a file
                        to a block size where a block size is 64-bit and I/O = 32-
    bit,
                        the value should be set to 2 (32*2 = 64 bits).

                        --io 32 --block_size 64
                        DAT = 000102030405060708090A0B0C0D0E0F

                        --io 32 --block_size 64 --max_io_per_line 2

                        DAT = 0001020304050607
                        DAT = 08090A0B0C0D0E0F
                         (default: 9999)
  --pdi_file FILENAME   Public data input filename (default: pdi.txt)
  --sdi_file FILENAME   Secret data input filename (default: sdi.txt)
  --do_file FILENAME    Data output filename (default: do.txt)
  --dest PATH_TO_DEST   Destination folder where the files should be written to. (
    default: .)
  --human_readable      Create a human readable file (tests_vectors.txt) for each
                        test vector in the format similar to NIST test vectors
                        used in SHA-3, i.e.:

                        # Message 1
                        Key      = HEXSTR    # if AEAD
                        Npub     = HEXSTR    # if AEAD
                        Nsec_PT  = HEXSTR    # if --nsec_size > 0
                        AD       = HEXSTR    # if AEAD
                        PT       = HEXSTR    # if AEAD
                        HASH     = HEXSTR    # if hash
                        Nsec_CT  = HEXSTR    # if --nsec_size > 0
                        CT       = HEXSTR    # if AEAD
                        TAG      = HEXSTR    # if AEAD
                        HASH_TAG = HEXSTR    # if hash
                         (default: False)
```

```
:
  [Experimental] CryptoCore options::

  --cc_hls              Generates test vectors for CryptoCore in C (used by HLS)
                         (default: False)
  --cc_pad_enable       Enable padding operation (default: False)
  --cc_pad_ad PAD_AD_MODE
                        Associated data padding mode (default: 0)
  --cc_pad_d PAD_D_MODE
                        Data input padding mode (default: 0)
  --cc_pad_style PAD_STYLE
                        Padding style (default: 1)
```

Bibliography

[1] NIST, “Lightweight Cryptography: Project Overview,” https://csrc.nist.gov/projects/lightweight-cryptography, 2019.

[2] J.-P. Kaps, W. Diehl, M. Tempelmeier, E. Homsirikamol, and K. Gaj, “Hardware API for Lightweight Cryptography,” Tech. Rep., Oct. 2019.

[3] “CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness - web page,” https://competitions.cr.yp.to/caesar.html, 2019.

[4] 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, v2.0,” GMU, Fairfax, VA, GMU Report, Dec. 2017.

[5] E. Homsirikamol, P. Yalla, and F. Farahmand, “Development Package for Hardware Implementations Compliant with the CAESAR Hardware API, v2.0,” https://cryptography.gmu.edu/athena/index.php?id=CAESAR, Dec. 2017.

[6] Cryptographic Engineering Research Group (CERG) at George Mason University, “Hardware Benchmarking of CAESAR Candidates,” https://cryptography.gmu.edu/athena/index.php?id=CAESAR, 2019.

[7] “TUMEISEC crypto implementation repository,” https://gitlab.lrz.de/tueisec/crypto-implementations/, git checkout.

[8] 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 Hard-*

ware Oriented Security and Trust, HOST 2018, Washington, DC, Apr. 2018, pp. 73–80.

[9] M. Tempelmeier, G. Sigl, and J.-P. Kaps, "Experimental Power and Performance Evaluation of CAESAR Hardware Finalists," in *2018 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2018*, Cancun, Mexico, Dec. 2018, pp. 1–6.

[10] Cryptographic Engineering Research Group (CERG) at George Mason University, "eXtended eXternal Benchmarking eXtension (XXBX)," https://cryptography.gmu.edu/xxbx/index.php, 2019.

[11] F. Farahmand, A. Ferozpuri, W. Diehl, and K. Gaj, "Minerva: Automated hardware optimization tool," in *2017 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2017*. Cancun: IEEE, Dec. 2017, pp. 1–8.

[12] Cryptographic Engineering Research Group (CERG) at George Mason University, "Authenticated Encryption FPGA Ranking," https://cryptography.gmu.edu/athenadb/fpga_auth_cipher/rankings_view, 2019.