Implementer’s Guide
to the CAESAR Hardware API

Ekawat Homsirikamol, William Diehl, Ahmed Ferozpuri,
Farnoud Farahmand, and Kris Gaj

Electrical and Computer Engineering Department,
George Mason University
Fairfax, Virginia 22030
{ehomsiri, wdiehl, aferozpu, ffarahma, kgaj}@gmu.edu

June 10, 2016
## Contents

1 Introduction 4

2 Top-level Block Diagram of a High-Speed Implementation 6
   2.1 PreProcessor 9
   2.2 PostProcessor 10
   2.3 CMD FIFO 11

3 The Development and Benchmarking of High-Speed and Lightweight Implementations 12

4 The AEAD Configuration 14
   4.1 High-Speed Implementations 14
   4.2 Lightweight Implementations 15

5 CipherCore Development for High-Speed Implementations 18
   5.1 Interface 18
   5.2 Handshakes 23
   5.3 Design Procedure 24
   5.4 Dummy Authenticated Ciphers 27
   5.5 AES and Keccak Permutation F 31

6 Test Vector Generation 33

7 Simulation 35

8 Generation and Publication of Results 36

A The Supporting Package Description 37
CONTENTS

B Installation of Libraries and Tools 39
B.1 Interpreter and compiler .......................... 39
B.2 OpenSSL Installation .............................. 40
B.3 Python module (aeadtvgen) ....................... 41

Bibliography 42
1 Introduction

The CAESAR Hardware API [1] is intended to meet the requirements of all algorithms submitted to the CAESAR competition, as well as many earlier developed authenticated ciphers, such as AES-GCM, AES-CCM, etc. The major parts of its specification [1] include the minimum compliance criteria, interface, communication protocol, and timing characteristics supported by the core. All of these parts have been defined with the goals of guaranteeing (a) compatibility among implementations of the same algorithm developed by different designers, and (b) fair benchmarking of authenticated ciphers in hardware.

Our proposed API is suitable for both high-speed and lightweight implementations of authenticated ciphers. The only difference at the API level is the width of Public Data Input (PDI) and Data Output (DO) ports, which is defined as follows:

Lightweight implementations: \( w = 8, 16, 32 \)
High-speed implementations: \( 32 \leq w \leq 256 \).

From the Implementer’s point of view, this difference is important, as small values of \( w \) (used in lightweight implementations) imply that any preprocessing (such as padding) and any postprocessing (such as zeroization of unused bytes) are significantly easier to implement compared to the case of large values of \( w \) (used in high-speed implementations).

As a result, we leave the internal structure of any lightweight implementation entirely to the designers of such implementations. The only support we provide to the designers of lightweight implementations is in the areas of test vector generation (Chapter 6), simulation (Chapter 7), as well as result generation and publication (Chapter 8).

On the other hand, for the designers of high-speed implementations, we provide the following support:

- universal top-level block diagram (see Fig. 2.1)
universal VHDL code for the PreProcessing unit

universal VHDL code for the PostProcessing unit

hardware API for the heart of the design, called CipherCore

implementer’s guide to designing any specific CipherCore

VHDL code for the three dummy CipherCores following the CipherCore API

Below we describe all these supporting materials one by one. It should be stressed that the high-speed implementations of authenticated ciphers compliant with the CAESAR hardware API can be also developed without using any resources described in this document, by just following directly the specification of the CAESAR API [1].
2 Top-level Block Diagram of a High-Speed Implementation

The proposed top-level block diagram of a high-speed, non-pipelined implementation of a single-pass authenticated cipher compliant with the CAESAR hardware API is shown in Fig. 2.1. The corresponding block diagram for a two-pass authenticated cipher is shown in Fig. 2.2. The only difference are ports used for communication with an external Two-Pass FIFO, used to store an output from the first pass of an implemented algorithm.

In each case, the top-level unit is divided into four lower-level units, called PreProcessor, PostProcessor, Command (CMD) FIFO, and CipherCore. The universal VHDL codes of the first three units are designed to be suitable for all authenticated ciphers to be implemented as a part of the CAESAR benchmarking project. These codes are provided as a part of the supporting package [2]. Due to the availability of this package as well as the well-defined hardware API of the CipherCore itself (described in Chapter 5), the implementers of any specific authenticated cipher do not need to be concerned with the internal details of the PreProcessor, PostProcessor, and CMD FIFO.

Because of the availability of the open source code for the PreProcessor, PostProcessor, and CMD FIFO, the designers of high-speed implementations of authenticated ciphers can focus exclusively on the development of the CipherCore unit, which can be further separated into its own datapath and controller, if desired.

Below is a high-level description of major functions of these units.
CHAPTER 2. TOP-LEVEL BLOCK DIAGRAM OF A HIGH-SPEED IMPLEMENTATION

Figure 2.1: Top-level block diagram of a high-speed architecture of a single-pass authenticated cipher core, AEAD
CHAPTER 2. TOP-LEVEL BLOCK DIAGRAM OF A HIGH-SPEED IMPLEMENTATION

Figure 2.2: Top-level block diagram of a high-speed architecture of a two-pass authenticated cipher core, AEAD_TP
CHAPTER 2. TOP-LEVEL BLOCK DIAGRAM OF A HIGH-SPEED IMPLEMENTATION

Figure 2.3: The PreProcessor Design. SIPO = Serial-In Parallel-Out unit. pdi_* and bdi_* stand for all PreProcessor ports, shown in Fig. 2.1 with the names starting from the respective strings.

2.1 PreProcessor

The PreProcessor is responsible for the execution of the following tasks common for majority of CAESAR candidates:

- parsing segment headers
- loading and activating keys
- Serial-In-Parallel-Out loading of input blocks
- padding input blocks, and
- keeping track of the number of data bytes left to process.

An overview of the PreProcessor design is shown Fig. 2.3. This unit can be configured to operate in two modes, registered and non-registered. The choice between these modes is made based on the width of public data input, PDI, (denoted as \( w \) in Fig. 2.1) and the size of an input block (denoted as \( DBLK\_SIZE \) in Fig. 2.1).

In a typical scenario, where the size of an input block is larger than the width of PDI, \( w \), the PreProcessor operates in the registered mode. If the width of PDI is the same as the size of an input block, the non-registered mode should be used. The non-registered mode ensures a high-throughput operation for algorithms that require a new block of data every clock cycle. It must be noted that operating the design in non-registered mode may affect the overall maximum clock frequency of the design due to additional critical path associated with the padding logic (if used).
CHAPTER 2. TOP-LEVEL BLOCK DIAGRAM OF A HIGH-SPEED IMPLEMENTATION

Figure 2.4: The PostProcessor Design. PISO = Parallel-In Serial-Out unit. msg_*, bdo_*, and do_* stand for all PostProcessor ports, shown in Fig. 2.1 with the names starting from the respective strings.

2.2 PostProcessor

The PostProcessor is responsible for the following tasks:

- clearing any portions of output blocks not belonging to the ciphertext or plaintext
- Parallel-In-Serial-Out conversion of output blocks into words
- formatting output words into segments
- generating the status block with the result of authentication.

An overview of the PostProcessor design is shown Fig. 2.4. This unit can be configured to operate in either registered or non-registered mode. The choice is made based on the dependence between the size of an output block (equal to the size of an input block) and the width of the data out, DO port (equal to width of public data input, PDI). Namely, when an output block size is larger than the width of DO, the registered mode is preferable. Otherwise, the non-registered mode should be used. Similarly to the PreProcessor design, when the unit operates in the non-registered mode, the maximum clock frequency maybe be affected.

The PreProcessor and PostProcessor units are highly configurable using generics of AEAD. These generics can be used, for example, to determine:

- the widths of the pdi, sdi, and do ports
- the size of the associated data block, message/ciphertext block, key, and tag
CHAPTER 2. TOP-LEVEL BLOCK DIAGRAM OF A HIGH-SPEED IMPLEMENTATION

- padding for the associated data and the message.

They have been designed to assure:

- Ease of use

- No influence on the maximum clock frequency of AEAD (up to 300 MHz in Virtex 7)

- Limited area overhead.

2.3 CMD FIFO

The Command (CMD) FIFO is a small 4x24 First-Word-Fall-Through (FWFT) FIFO that temporarily stores all significant bits of instructions and segment headers that need to be passed to the output. This module allows the Pre-Processor to operate with the maximum efficiency. This FIFO’s width is selected based on the fact that the instructions defined in [1], Fig. 7, contain only 4 significant bits, and segment headers, defined in [1], Fig. 8, contain only 24 significant bits.
3 The Development and Benchmarking of High-Speed and Lightweight Implementations

The development and benchmarking of a **high-speed** implementation of a selected authenticated cipher can be performed using the following major steps, described in the subsequent chapters of this guide:

1. Configure the provided AEAD entity declaration for high-speed implementations (Chapter 4)
2. Develop CipherCore (Chapter 5)
3. Generate test vectors (Chapter 6)
4. Verify the AEAD design (including the CipherCore design) using functional simulation (Chapter 7)
5. Generate optimized results for AEAD using FPGA tools (Chapter 8).

The development and benchmarking of a **lightweight** implementation of a selected authenticated cipher can be performed using the following major steps, described in the subsequent chapters of this guide:

1. Configure the provided AEAD entity declaration for lightweight implementations (Chapter 4)
2. Develop the entire AEAD core from scratch, based on the CAESAR Hardware API specification
3. Generate test vectors (Chapter 6)
4. Verify the AEAD design using functional simulation (Chapter 7).

5. Generate optimized results for AEAD using FPGA tools (Chapter 8).

As can be seen from the above description, only the first two steps are different. All remaining steps are universal and apply to both high-speed and lightweight implementations.
4 The AEAD Configuration

4.1 High-Speed Implementations

The entity declaration of AEAD for high-speed implementations is available as a part of the supporting package in the file

$ROOT/hardware/AEAD/src_rtl_hs/AEAD.vhd

This entity declaration contains multiple generics defined in Table 4.1. Additional generics, used to determine the desired padding scheme are defined in Tables 4.2 and 4.3. The names of all generics, listed in the aforementioned tables, are supplemented in the VHDL code with the prefix G_.

The following restrictions must be considered when configuring the AEAD entity for high-speed implementations:

4.1.1 I/O Port Widths

Consistently with the specification of the CAESAR hardware API, the allowed values of the port widths for high-speed implementations are as follows:

\[ 32 \leq w \leq 256, \]
\[ 32 \leq sw \leq 64. \]

These widths are described in the AEAD entity declaration using generics W and SW.

4.1.2 Block sizes

Values of generics ABLK_SIZE and DBLK_SIZE, describing the sizes of input blocks for associated data and message/ciphertext, respectively, must be multiples of the generic W. Similarly, the generic KEY_SIZE must be
a multiple of the generic SW. Additionally, ABLK_SIZE is assumed to be smaller than or equal to DBLK_SIZE.

4.1.3 The Preprocessor and PostProcessor Maximum Input/Output Rates

The maximum rate at which the PreProcessor can provide a block of data and the PostProcessor can accept a block of data is dependent on the size of the message/ciphertext block (DBLK_SIZE) and the I/O port width (W). In the registered mode of operation, a new block of input data can be provided by the PreProcessor and accepted by the PostProcessor every \( \frac{DBLK\_SIZE}{W} + 1 \) clock cycles. In the non-registered mode, a new block of input data can be provided by the PreProcessor and accepted by the PostProcessor every clock cycle.

4.2 Lightweight Implementations

The entity declaration of AEAD for lightweight implementations is available as a part of the supporting package in the file

\$ROOT/hardware/AEAD/src_rtl_lw/AEAD.vhd

This entity declaration contains only values of generics G_W and G_SW, used to determine the I/O port widths, \( w \) and \( sw \), respectively. Consistently with the specification of the CAESAR hardware API, the allowed values of these port widths are as follows:

\[
\begin{align*}
w &= 8, 16, 32, \\
sw &= 8, 16, 32.
\end{align*}
\]
Table 4.1: AEAD Generics

<table>
<thead>
<tr>
<th>Generic</th>
<th>Type</th>
<th>Default Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>I/O Widths in Bits</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W</td>
<td>Integer</td>
<td>32</td>
<td>Public data input and data output width</td>
</tr>
<tr>
<td>SW</td>
<td>Integer</td>
<td>32</td>
<td>Secret data input width</td>
</tr>
<tr>
<td><strong>Reset Behavior</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASYNC_RSTN</td>
<td>Boolean</td>
<td>False</td>
<td>Reset behavior. True=Asynchronous active low, False= Synchronous active high.</td>
</tr>
<tr>
<td><strong>Special Features</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENABLE_PAD</td>
<td>Boolean</td>
<td>False</td>
<td>Enable padding (See additional settings in Tables 4.2 and 4.3)</td>
</tr>
<tr>
<td>CIPH_EXP</td>
<td>Boolean</td>
<td>False</td>
<td>Ciphertext expansion mode. This option should be used when the ciphertext size is not the same as the plaintext size, i.e., the ciphertext is expanded. It should also be used when Cipher=Ciphertext</td>
</tr>
<tr>
<td>REVERSE_CIPH</td>
<td>Boolean</td>
<td>False</td>
<td>Reverse ciphertext mode. Used, for example, by PRIMATEs-APE, currently not supported.</td>
</tr>
<tr>
<td>MERGE_TAG</td>
<td>Boolean</td>
<td>False</td>
<td>No tag segment. This parameter should be set to True when the CipherCore does not separate Tag from Ciphertext, i.e., Cipher=Ciphertext</td>
</tr>
<tr>
<td><strong>Block Size Parameters in Bits</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABLK_SIZE</td>
<td>integer</td>
<td>128</td>
<td>Associated data block size. This value should be smaller than or equal to DBLK_SIZE.</td>
</tr>
<tr>
<td>DBLK_SIZE</td>
<td>integer</td>
<td>128</td>
<td>Data (message/ciphertext) block size</td>
</tr>
<tr>
<td>KEY_SIZE</td>
<td>integer</td>
<td>128</td>
<td>Key size</td>
</tr>
<tr>
<td>TAG_SIZE</td>
<td>integer</td>
<td>128</td>
<td>Tag size. Note: This value is not used when MERGE_TAG is True.</td>
</tr>
<tr>
<td><strong>Padding Parameters</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PAD_STYLE</td>
<td>integer</td>
<td>1</td>
<td>Padding style. See Table 4.2</td>
</tr>
<tr>
<td>PAD_AD</td>
<td>integer</td>
<td>1</td>
<td>Padding behavior for associated data. See Table 4.3</td>
</tr>
<tr>
<td>PAD_D</td>
<td>integer</td>
<td>1</td>
<td>Padding behavior for message. See Table 4.3</td>
</tr>
</tbody>
</table>
Table 4.2: Extended description of PAD_STYLE.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No padding</td>
</tr>
<tr>
<td>1</td>
<td>10(^x) padding rule</td>
</tr>
<tr>
<td>2</td>
<td>ICEPOLE padding rule</td>
</tr>
</tbody>
</table>

Table 4.3: Parameters of PAD_AD and PAD_D. A = Pad enable. B = Extra block is added when AD/D is empty. C = Extra block is added when AD/D is a non-zero multiple of a block size.

<table>
<thead>
<tr>
<th>Value</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A \</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
</tr>
<tr>
<td>3</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td>x</td>
</tr>
</tbody>
</table>
5 CipherCore Development for High-Speed Implementations

5.1 Interface

The interface of CipherCore is shown in Figure 5.1. Ports marked using dashed lines 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 key_valid and key_ready handshake signals.

Figure 5.1: CipherCore
CHAPTER 5. CIPHERCORE DEVELOPMENT FOR HIGH-SPEED IMPLEMENTATIONS

*key_update* is used to notify the CipherCore that it should update the internal key prior to processing the next message.

Similarly to the *key* port, the *bdi* port is controlled using the *bdi_valid* and *bdi_ready* handshake signals. The *decrypt* signal informs the core whether the current operation is encryption or decryption. The *bdi_type* input indicates the type of input data, with the encoding shown in Table 5.1.

It must be noted that all ports of the BDI communication group and *bdi* are synchronized with the *bdi_valid* input. Their values should be read only when the *bdi_valid* signal is high.

Table 5.1: *bdi_type* Encoding. – represents don’t care.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>00–</td>
<td>Associated Data</td>
</tr>
<tr>
<td>01–</td>
<td>Message/Ciphertext/Ciphertext</td>
</tr>
<tr>
<td>100</td>
<td>Tag</td>
</tr>
<tr>
<td>101</td>
<td>Length</td>
</tr>
<tr>
<td>110</td>
<td>Public message number</td>
</tr>
<tr>
<td>111</td>
<td>Secret message number</td>
</tr>
</tbody>
</table>

The same scenario also applies to the block data output port (*bdo*) and its associated control signals, which are synchronized with the value of the *bdo_valid* output. *bdo_size* is not used unless the CIPH_EXP generic of AEAD is set to True. When this is the case, each active value of *bdo_valid* must be accompanied by providing the size of an output block, in bytes, using the *bdo_size* port.

The message authentication ports (*msg_auth_* *) are only used during the authenticated decryption operation, when the core must provide output signals indicating whether the authentication is done and the result is (or is not) valid. Note that *msg_auth_valid* signal is synchronized with *msg_auth_done* signal.

Port descriptions are provided in Table 5.2. Ports related to *bdi* control are categorized according to the following criteria:

**COMM** A handshake signal.

**INPUT INFO** An auxiliary signal that remains valid until a given input is fully processed. Deactivation is typically done at the end of input.
Table 5.2: CipherCore Port Descriptions. \( \text{LBS}_{\text{BYTES}} = \log_2(\text{DBLK}_{\text{SIZE}}/8) \)

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Input &amp; Output</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>key</td>
<td>in</td>
<td>KEY_SIZE</td>
<td>Key data</td>
</tr>
<tr>
<td>bdi</td>
<td>in</td>
<td>DBLK_SIZE</td>
<td>Block data input</td>
</tr>
<tr>
<td>bdo</td>
<td>out</td>
<td>DBLK_SIZE</td>
<td>Block data output</td>
</tr>
<tr>
<td>Key Control</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>key_valid</td>
<td>in</td>
<td>1</td>
<td>Key data is valid</td>
</tr>
<tr>
<td>key_ready</td>
<td>out</td>
<td>1</td>
<td>CipherCore is ready to receive a new key</td>
</tr>
<tr>
<td>key_update</td>
<td>in</td>
<td>1</td>
<td>Key must be updated prior to processing a new input</td>
</tr>
<tr>
<td>BDI Control</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>decrypt</td>
<td>in</td>
<td>1</td>
<td>[INPUT INFO] 0=Encryption, 1=Decryption</td>
</tr>
<tr>
<td>bdi_valid</td>
<td>in</td>
<td>1</td>
<td>[COMM] BDI data is valid</td>
</tr>
<tr>
<td>bdi_ready</td>
<td>out</td>
<td>1</td>
<td>[COMM] CipherCore is ready to receive data</td>
</tr>
<tr>
<td>bdi_type</td>
<td>in</td>
<td>3</td>
<td>[BLOCK INFO] Type of BDI data. See Table 5.1.</td>
</tr>
<tr>
<td>bdi_eot</td>
<td>in</td>
<td>1</td>
<td>[BLOCK INFO] The current BDI block is the last block of its type. Note: Only applies when the type is either AD, Message, or Ciphertext.</td>
</tr>
<tr>
<td>bdi_eoi</td>
<td>in</td>
<td>1</td>
<td>[BLOCK INFO] 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.</td>
</tr>
<tr>
<td>bdi_partial</td>
<td>in</td>
<td>1</td>
<td>[SEGMENT INFO] The current block is either a partial block of AD or Message, or the result of encryption of a partial message block. Note: This optional signal is used only in the implementations of the ciphertext expansion algorithms. We are aware of its necessity only for the implementation of the Round 2 AES-COPA.</td>
</tr>
<tr>
<td>bdi_pad_loc</td>
<td>in</td>
<td>DBLK_SIZE/8</td>
<td>[BLOCK INFO] Encoding of the byte location where padding begins. See Table 5.3.</td>
</tr>
<tr>
<td>bdi_valid_bytes</td>
<td>in</td>
<td>DBLK_SIZE/8</td>
<td>[BLOCK INFO] Encoding of the byte locations that are valid. See Table 5.3.</td>
</tr>
<tr>
<td>bdi_size</td>
<td>in</td>
<td>LBS.GetBytes+1</td>
<td>[BLOCK INFO] Number of valid bytes in bdi.</td>
</tr>
<tr>
<td>BDO Control</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bdo_valid</td>
<td>out</td>
<td>1</td>
<td>BDO data is valid</td>
</tr>
<tr>
<td>bdo_ready</td>
<td>in</td>
<td>1</td>
<td>PostProcessor is ready to receive data</td>
</tr>
<tr>
<td>bdo_size</td>
<td>out</td>
<td>LBS.GetBytes+1</td>
<td>Number of valid bytes in bdo. This port must be used when CIPH_EXP is active.</td>
</tr>
<tr>
<td>Tag Verification</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>msg_auth_valid</td>
<td>out</td>
<td>1</td>
<td>1=Authentication success, 0=Authentication failure</td>
</tr>
<tr>
<td>msg_auth_done</td>
<td>out</td>
<td>1</td>
<td>Authentication done</td>
</tr>
</tbody>
</table>
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 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 5.3, 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. Assuming the 10\* padding, this block consists of a single 1 followed by 31 zeros.

Table 5.3: Values of the special control signals `bdi_valid_bytes`, `bdi_pad_loc`, and `bdi_size` for the `bdi` bus with the width \( w = 32 \). Byte Validity represents the byte locations in `bdi` that were the part of input (e.g., AD or message) before padding.

<table>
<thead>
<tr>
<th>Byte/Bit Position</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Case A</strong></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><strong>Case B</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td><strong>Case C</strong></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td><strong>Case D</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

For the CipherCore that supports Two-Pass algorithms, additional ports have been added to accommodate the communication with the external FIFO, as shown in Figure 5.2.
The additional port descriptions required for a CipherCore that supports Two-Pass algorithms are provided in Table 5.4. It must be noted that all the ports listed in Table 5.2 are also present in the interface of the Two-Pass core.

Table 5.4: Additional Port Descriptions for a Two-Pass CipherCore.

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdi_data</td>
<td>in</td>
<td>FW</td>
<td>Input data from the two-pass FIFO</td>
</tr>
<tr>
<td>fdo_data</td>
<td>out</td>
<td>FW</td>
<td>Output data to the two-pass FIFO</td>
</tr>
<tr>
<td>fdi_valid</td>
<td>in</td>
<td>1</td>
<td>fdi data is valid</td>
</tr>
<tr>
<td>fdi_ready</td>
<td>out</td>
<td>1</td>
<td>CipherCore is ready to receive a new two-pass data</td>
</tr>
<tr>
<td>fdo_valid</td>
<td>out</td>
<td>1</td>
<td>CipherCore is ready to send a new two-pass data</td>
</tr>
<tr>
<td>fdo_ready</td>
<td>in</td>
<td>1</td>
<td>two-pass FIFO is ready to receive a new data</td>
</tr>
</tbody>
</table>
CHAPTER 5. CIPHERCORE DEVELOPMENT FOR HIGH-SPEED IMPLEMENTATIONS

5.2 Handshakes

This section presents examples of handshakes. All ports in the figures of this section are represented by the blue and the red color, for input and output ports, respectively. Fig. 5.3 provides an example of a handshake used for loading a block of data using the (bdi) port. Data and its auxiliary signals are synchronized with the bdi_valid signal. Similarly for key, data is synchronized with the key_valid signal, as shown in Figure 5.4.

Fig. 5.5 provides an example of a handshake used to write output to the PostProcessor. Fig. 5.5a presents an example for the standard mode of operation of an authenticated cipher. Figure 5.5b presents an example for the case of an algorithm operating in the ciphertext expansion mode. An additional output port (bdo_size) is now required to update the PostProcessor about the size of the current message block after decryption. This information is used by the PostProcessor to update the header with correct value of the last segment size.

Finally, an example of a handshake for authentication is shown in Fig. 5.6. For every decryption operation, PostProcessor should issue the msg_auth_done signal to indicate the completion of the authentication check. At the same time, msg_auth_valid is captured by the PostProcessor to determine the result of authentication. These two signals should only be activated once for every decryption. Subsequent values of the msg_auth_done signal during the same decryption operation are ignored.
5.3 Design Procedure

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

$\text{ROOT}/\text{hardware}/\text{AEAD}/\text{src}_\text{rtl}_\text{hs}$

In particular, the appropriate connections among the CipherCore, the Pre-Processor, the PostProcessor, and the CMD FIFO modules are already specified in this code. A designer needs to modify generics in the AEAD module, and then develop the CipherCore Datapath and the CipherCore Controller.

The development of the CipherCore 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
CipherCore Datapath is first modeled using a block diagram, and then translated to a hardware description language (VHDL or Verilog HDL). The CipherCore Controller is then described using an algorithmic state machine (ASM) chart or a state diagram, further translated to HDL.

An ASM chart of the CipherCore Controller typically contains the following states:

1. Idle
2. Activate Key
3. Load Npub
4. Load Data
5. Process AD
6. Process AD Last
7. Process Data
8. Process Data Last
9. Generate/verify Tag (GenVer Tag).

An example ASM chart for the CipherCore Controller is shown in Fig. 5.7. After a new instruction or after reset, the control should wait for the first block of data in the Idle state. The CipherCore should monitor the bdi_valid for the first block of data, which is typically Npub. When this signal is active, the circuit should check whether the current key requires an update by inspecting the key_update signal. If it does, the controller changes its state to Activate Key. In this state either a new key is stored internally within the CipherCore or the corresponding round keys are precomputed. Once this task is completed, key_ready should be activated to acknowledge the key activation.

Once a new key is activated or no new key is required (key_update=0), the circuit is ready to process the first block of data (Npub) in the Load Npub state. At the same time, that the Npub block is loaded into the CipherCore, the circuit needs to acknowledge its receipt by setting the bdi_ready output to high. The controller then moves to the next processing state, Load Data. In the case that Npub is the last block of data (AD size = Message/Ciphertext size = 0), which can be determined using the bdi_eoi input, the controller state can change directly to Generate/verify tag.

In the Load Data state, the circuit waits until the next input block is valid (bdi_valid=1), and then processes data based on the incoming input
Figure 5.7: A typical Algorithmic State Machine (ASM) chart of the CipherCore 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. * _r are status registers storing values of the respective inputs read during the last bdi handshake.
type (bdi\_type). 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). At the same time, the end-of-input signal (bdi\_eoi) may be stored in a register within the CipherCore to keep track of the last input state. This status register is useful to determine when no additional data block is expected after processing of the last AD block, so that the controller can progress to the last state (Generate/verify tag) directly.

In the last state, Generate/verify tag, during the authenticated encryption operation, the core should generate a new tag and pass it to the PostProcessor via the bdo bus. During the authenticated decryption operation, msg\_auth\_done should be activated, and the msg\_auth\_valid signal should be used to provide the result of authentication.

### 5.4 Dummy Authenticated Ciphers

Five example designs of the CipherCore and AEAD, corresponding to five Dummy Authenticated Ciphers, are provided as a part of our distribution. The first three Dummy Authenticated Ciphers is specified using the following equations:

\[
AD = AD_1, AD_2, ..., AD_{n-1}, AD_n \quad (5.1)
\]

\[
PT = PT_1, PT_2, ..., PT_{m-1}, PT_m \quad (5.2)
\]

\[
CT = CT_1, CT_2, ..., CT_{m-1}, CT_m \quad (5.3)
\]

\[
CT_i = PT_i \oplus i \oplus Key \oplus Npub \quad (5.4)
\]

for \(i = 1..m - 1\).

\[
CT_m = \text{Trunc}(PT_m \oplus i \oplus Key \oplus Npub, PT_m) \quad (5.5)
\]

when CIPH\_EXP=False.

\[
CT_m = \text{Pad}(PT_m) \oplus m \oplus Key \oplus Npub \quad (5.6)
\]
CHAPTER 5. CIPHERCORE DEVELOPMENT FOR HIGH-SPEED IMPLEMENTATIONS

when \( \text{CIPH\_EXP} = \text{True} \).

\[
Tag = \text{Key} \oplus \text{Npub} \oplus \text{Len} \oplus \bigoplus_{i=1}^{n-1} \text{AD}_i \oplus \text{Pad}(\text{AD}_n) \oplus \bigoplus_{i=1}^{m-1} \text{PT}_i \oplus \text{Pad}(\text{PT}_m)
\]

(5.7)

where,

- \( PT_i \) and \( CT_i \) are the plaintext (message) and ciphertext blocks, respectively,
- \( AD_i \) = associated data block,
- \( \text{Pad}(\cdot) \) represents a padding operation applied to the last AD and/or the last plaintext block,
- \( \text{Trunc}(X, Y) \) truncates X to the size of Y,
- \( i \) = 128-bit block number,
- \( \text{Key} \) = 128-bit key,
- \( \text{Npub} \) = Public message number,
- \( \text{Len} \) = 64-bit associated data length (in bytes) || 64-bit plaintext length (in bytes).

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. All examples are based on a 128-bit data block, unless specified otherwise. The differences between each Dummy Authenticated Cipher are primarily based on the definition of padding and values of parameters described below. Please note that a typical padding behavior is either appending all zeros (0* or one followed by zeros (10*).

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

The features of all five dummy cores are summarized in Table 5.5.
CHAPTER 5. CIPHERCORE DEVELOPMENT FOR HIGH-SPEED IMPLEMENTATIONS

Table 5.5: Summary of features/parameters of five dummy authenticated ciphers and their high-speed implementations

<table>
<thead>
<tr>
<th>CIPHER</th>
<th>Npub Size</th>
<th>AD Block size</th>
<th>Pad?</th>
<th>PT Block size</th>
<th>Pad?</th>
<th>Tag Off-line?</th>
<th>Pre-Processor Data buffer?</th>
<th>Key buffer?</th>
</tr>
</thead>
<tbody>
<tr>
<td>dummy1</td>
<td>False</td>
<td>96</td>
<td>128</td>
<td>True</td>
<td>128</td>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>dummy2</td>
<td>False</td>
<td>128</td>
<td>96</td>
<td>False</td>
<td>128</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>dummy3</td>
<td>True</td>
<td>128</td>
<td>128</td>
<td>True</td>
<td>128</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>dummy4</td>
<td>False</td>
<td>128</td>
<td>32</td>
<td>True</td>
<td>32</td>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>dummy5</td>
<td>False</td>
<td>128</td>
<td>32</td>
<td>True</td>
<td>32</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

5.4.1 dummy1

This example is aimed at presenting the behavior of the Pre- and Post-processors for a typical CipherCore. The following parameters are used:

- \( AD_{\text{block size}} = PT_{\text{block size}} = 128 \) bits
- \( Npub_{\text{size}} = 96 \) bits
- \( \text{Pad}(AD_n) = AD_n \text{ if len}(AD_n) = \text{block size} \text{ else } AD_n \| 10^* \)
- \( \text{Pad}(PT_m) = PT_m \text{ if len}(PT_m) = \text{block size} \text{ else } PT_m \| 10^* \)
- \( \text{CIPH\_EXP} = \text{False} \)

5.4.2 dummy2

This example aims at presenting the behavior of the PreProcessor when \( AD_{\text{block size}} \neq PT_{\text{block size}} \), and zero padding is applied to AD. The following parameters are used:

- \( AD_{\text{block size}} = 96 \) bits
- \( PT_{\text{block size}} = Npub_{\text{size}} = 128 \) bits
- \( \text{Pad}(AD_n) = AD_n \text{ if len}(AD_n) = \text{block size} \text{ else } AD_n \| 0^* \)
- \( \text{Pad}(PT_m) = PT_m \text{ if len}(PT_m) = \text{block size} \text{ else } PT_m \| 10^* \)
- \( \text{CIPH\_EXP} = \text{False} \)
5.4.3 dummy3

This example aims at presenting an example implementation for algorithms that have ciphertext expansion. The following parameters are used:

- \( AD_{\text{block\_size}} = PT_{\text{block\_size}} = Npub_{\text{size}} = 128 \) bits
- \( \text{Pad}(AD_n) = AD_n \text{ if } \text{len}(AD_n) = \text{block\_size} \text{ else } AD_n || 10^* \)
- \( \text{Pad}(PT_m) = PT_m \text{ if } \text{len}(PT_m) = \text{block\_size} \text{ else } PT_m || 10^* \)
- \( \text{CIPH\_EXP} = \text{True} \)

Additionally, the Len segment is removed from the tag generation for this dummy core, so the new equation for \( Tag \) is

\[
Tag = Key \oplus Npub \oplus \bigoplus_{i=1}^{n-1} AD_i \oplus \text{Pad}(AD_n) \oplus \bigoplus_{i=1}^{m-1} PT_i \oplus \text{Pad}(PT_m) \quad (5.8)
\]

5.4.4 dummy4

This example aims at presenting the behavior of the Pre- and Post-processor for the following cases:

- External public bus size is equal to the internal data bus size, i.e., \( W = \text{DBLK\_SIZE} \). This allow the PreProcessor to operate in the non-registered mode for the bdi input.
- Tag size is larger than the data bus size, i.e., \( \text{TAG\_SIZE} > \text{DBLK\_SIZE} \).
- \( Npub \) size is larger than the data bus size.

For this example, the same padding rules as those used in dummy1 are applied, together with the following values of parameters:

- \( AD_{\text{block\_size}} = PT_{\text{block\_size}} = 32 \) bits
- \( Npub_{\text{size}} = 128 \) bits
- \( Key = 128 \) bits
- \( Tag = 64 \) bits.
Additionally, the ciphertext and the tag are described as followed:

\[ C_{Ti} = PT_i \oplus i \oplus KN \quad (5.9) \]

for \( i = 1..m - 1 \).

\[ C_{Tm} = Trunc(PT_m \oplus m \oplus KN, PT_m) \quad (5.10) \]

\[ Tag_{63..32} = KN \oplus \bigoplus_{i=1}^{n-1} AD_i \oplus Pad(AD_n) \oplus \bigoplus_{i=1}^{m-1} PT_i \oplus Pad(PT_m) \quad (5.11) \]

\[ Tag_{31..0} = \bigoplus_{i=1}^{n-1} AD_i \oplus Pad(AD_n) \oplus \bigoplus_{i=1}^{m-1} PT_i \oplus Pad(PT_m) \quad (5.12) \]

where,

\[ KN = Key_{127..96} \oplus Key_{95..64} \oplus Key_{63..32} \oplus Key_{31..0} \]

\[ \oplus Npub_{127..96} \oplus Npub_{95..64} \oplus Npub_{63..32} \oplus Npub_{31..0} \]

5.4.5 *dummy5*

This example uses the same algorithm as *dummy4* except that the hardware implementation relies on a different PreProcessor settings. In particular, the key bus size (KEY _SIZE) is set to the same width as sdi bus size (SW). As a result, the PreProcessor operates in a non-registered mode for the key as well as the bdi input. This mode reduces the AEAD overall resource utilization as the key is not buffered inside the PreProcessor.

5.5 AES and Keccak Permutation F

Additional support is provided for designers of cipher cores of CAESAR candidates based on AES and Keccak. Fully verified VHDL codes, block diagrams, and ASM charts of AES and Keccak Permutation F have been developed and made available at [?]. Our AES core implements a basic iterative architecture of a block cipher, with the SubBytes operation realized
using memory. Either distributed memory (implemented using multipurpose LUTs) or block memory is inferred depending on the specific options of FPGA tools.
6 Test Vector Generation

Test vectors for the targeted algorithm can be generated using our test vector generator (*aedtvgen*) available in the *software* folder of our development package. The program relies on a reference software implementation located at `$ROOT/software/CAESAR/$algorithm/ref` that uses the CAESAR software API to create a shared library used by our program, where `$algorithm` is algorithm’s name/implementation. A limited set of reference C implementations of Round 2 CAESAR candidates is provided as part of our development package.

In the case that the targeted algorithm is not available as part of our package, the user can add a new algorithm at the location noted above, and perform a slight modification to the source code. In particular, user must include an additional header file (`dll.h`) located in the `$REPO/software/CAESAR` folder as well as prepend an *EXPORT* syntax to CAESAR software API, e.g.:

```c
#include "../dlib.dll"
EXPORT int crypto_aead_encrypt(...) { ... }
EXPORT int crypto_aead_decrypt(...) { ... }
```

Note: For the installation procedure of the recommended software, please refer to Appendix B.

A standard procedure for creating the testbench can be executed as follows:

1. Create shared CAESAR libraries (*.*dll in Windows and *.*so in Linux)
   a) In console, navigate to the CAESAR folder (`$root/software/CAESAR`).
      Note: For Windows, perform this step using *msys* console
   b) Modify *Makefile* to include only targeted primitive(s).
c) (Situational) An algorithm may require OpenSSL library in order to compile. If it does, one needs to provide an appropriate compilation flag inside the following clause:

```c
ifeq ($(OS),Windows_NT)
...
else
...
endif
```

Note: The flags are available but uncommented by default.

d) type

```
make
```

2. Generate the script using aeadtvgen python program. The user can directly use the program from a command line or create a script similar to examples shown in $root/software/aeadtvgen/examples folder. Full description of the program can be found by typing

```
python -m aeadtvgen -h
```

3. Copy the three generated test vectors (pdi.txt, sdi.txt and do.txt) to simulation folder.
7 Simulation

Once test vectors are generated, copy them into your simulation folder and ensure that the PWIDTH and SWIDTH generics of the testbench are set to W and SW, respectively.

Simulation is performed until end-of-file is reached or a mismatched between expected output and output data occurs. A clock signal is deactivated when either of the conditions apply. In the case that user wants to ignore the simulation mismatch, one can set the STOP_AT_FAULT generic to False and the testbench will ignore the verification error.

In the case that the target implementation is ASIC, one can simulate the design by setting ASYNC_RSTN to True.

Finally, four test modes, summarized in Table 7.1, are provided to simulate the conditions when available of source or target communication modules are intermittent. The rate at which the data is not available can be configured using TEST_ISTALL and TEST_OSTALL for input and output (in cycles), respectively.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Always available</td>
</tr>
<tr>
<td>1</td>
<td>Input &amp; Output intermittent test</td>
</tr>
<tr>
<td>2</td>
<td>Input intermittent test</td>
</tr>
<tr>
<td>3</td>
<td>Output intermittent test</td>
</tr>
</tbody>
</table>

Table 7.1: Test modes
8 Generation and Publication of
Results

Generation of results is possible for AEAD and CipherCore. We strongly recommend generating results primarily for AEAD. This recommendation is based on the fact that CipherCore has an incomplete functionality and a full-block-width interface.

In case AEAD, for Virtex 7 and Zynq, we recommend generating results using Xilinx Vivado [3], operating in the Out-of-Context (OOC) mode [4]. In this mode, no pin limit applies. For Virtex 6 and below, since Xilinx ISE must be used, and the OOC mode is not supported by this tool, we recommend using a simple wrapper, with five ports: clk, rst, sin, sout, piso_mux_sel, provided as a part of supporting files [2].

In case of CipherCore, because of a large number of port bits and limited effectiveness of the OOC mode, we recommend using the aforementioned five-port wrapper for all FPGA families.

In terms of optimization of tool options, for Virtex 7 and Zynq, we recommend the use of 25 default optimization strategies available in Xilinx Vivado. The corresponding scripts, used to run Xilinx Vivado in batch mode, are included in our supporting codes [2], and their use is explained in detail in Appendix E. For Virtex 6 and below, we recommend using Xilinx ISE and ATHENa [5]. For Altera FPGAs, we suggest using Altera Quartus II and ATHENa.

Our database of results for authenticated ciphers is available at [6]. After receiving an account in the database, the designers can enter results by themselves.
A The Supporting Package Description

The contents of our development package is shown in Table A.1.
## Table A.1: Directory structure of the development package

<table>
<thead>
<tr>
<th>Folder</th>
<th>Files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>scripts</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VivadoBatch</td>
<td>Directory that contains a set of scripts for result generation using Vivado</td>
</tr>
<tr>
<td></td>
<td>ModelSim</td>
<td>modelsim script to run a reference design</td>
</tr>
<tr>
<td><strong>software</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AETVgen</td>
<td>Test vector generator program</td>
</tr>
<tr>
<td></td>
<td>gen.py</td>
<td>Example usage</td>
</tr>
<tr>
<td></td>
<td>CAESAR/ {algorithm}/ref</td>
<td>Directory that contains an implementation of a specific algorithm of CAESAR candidate from SUPERCOP distribution</td>
</tr>
<tr>
<td><strong>hardware</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AEAD/src_rtl ls</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AEAD.vhd</td>
<td>Entity only file of AEAD</td>
</tr>
<tr>
<td></td>
<td>AEAD_Arch.vhd</td>
<td>Architecture only file of AEAD</td>
</tr>
<tr>
<td></td>
<td>CipherCore.vhd</td>
<td>CipherCore template</td>
</tr>
<tr>
<td></td>
<td>AEAD_TP.vhd</td>
<td>Entity only file of Two-Pass AEAD</td>
</tr>
<tr>
<td></td>
<td>AEAD_TP_Arch.vhd</td>
<td>Architecture only file of Two-Pass AEAD</td>
</tr>
<tr>
<td></td>
<td>CipherCore_TP.vhd</td>
<td>Two-Pass CipherCore template</td>
</tr>
<tr>
<td></td>
<td>lwft_fifo.vhd</td>
<td>First-Word-Fall-Through FIFO</td>
</tr>
<tr>
<td></td>
<td>PostProcessor.vhd</td>
<td>PostProcessor file</td>
</tr>
<tr>
<td></td>
<td>PreProcessor.vhd</td>
<td>PreProcessor file</td>
</tr>
<tr>
<td></td>
<td>AEAD/src_rtl lw</td>
<td>Top-level template file for lightweight hardware design.</td>
</tr>
<tr>
<td></td>
<td>AEAD TB.vhd</td>
<td>Universal testbench file</td>
</tr>
<tr>
<td></td>
<td>AEAD_TP TBWrapper.vhd</td>
<td>Wrapper file to use with the test of Two-Pass AEAD</td>
</tr>
<tr>
<td></td>
<td>std_logic_1164_additions.vhd</td>
<td>Additional simulation package</td>
</tr>
<tr>
<td></td>
<td>AEAD Wrapper.vhd</td>
<td>Wrapper file for implementation of AEAD</td>
</tr>
<tr>
<td></td>
<td>CipherCore_Wrapper.vhd</td>
<td>Wrapper file for implementation of CipherCore</td>
</tr>
<tr>
<td></td>
<td>AEAD_TP Wrapper.vhd</td>
<td>Wrapper file for implementation of Two-Pass AEAD</td>
</tr>
<tr>
<td></td>
<td>CipherCore_TP Wrapper.vhd</td>
<td>Wrapper file for implementation of Two-Pass CipherCore</td>
</tr>
<tr>
<td><strong>dummy</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>KAT</td>
<td>Known-Answer-Test folder for dummy design</td>
</tr>
<tr>
<td></td>
<td>src_rtl</td>
<td>Reference dummy related code</td>
</tr>
<tr>
<td></td>
<td>scripts</td>
<td>ModelSim script folder to perform a quick simulation</td>
</tr>
</tbody>
</table>
B Installation of Libraries and Tools

B.1 Interpreter and compiler

B.1.1 Windows

- **MinGW with MSYS as a compiler**
  Download and install the latest version from [http://www.mingw.org](http://www.mingw.org).
  MSYS should be included in the installation package.
  Note: MSYS is the console for MinGW in Windows.

Below is an example in how to compile the program using MinGW with Msys console (MingW shell).

```
$ cd /c/Downloads/GMU_API_v20/software/CAESAR
$ make
```

- **Python v3.5+**
  Download and install the latest Python distribution package from [https://www.python.org](https://www.python.org).
  Note: Please make sure that all installations are done as an administrator and the path to python is correctly set in the environmental variable.

B.1.2 Linux

- **Python v3.5+**
B.2 OpenSSL Installation

1. Download and uncompress the latest version of OpenSSL

2. Navigate to download folder and uncompress files
   a) Open terminal (Msys console for Windows)
   b) Navigate to the download folder
      • Windows
        cd /c/Users/$USER/Downloads/openssl-1.0.2e
      • Linux Open terminal
        cd /home/$USER/Downloads
   c) Uncompress downloaded file, e.g.
      tar -zxvf openssl-1.0.2e.tar.gz
   d) Change working directory
      cd openssl-1.0.2e
   e) Configure OpenSSL
      • Windows
        ./Configure mingw --prefix=/usr/local shared

Note: Possible error
"gcc command not found" error
This is caused by a problem during installation where your MingW’s /bin folder is not included as a part of environmental variable. You can either try to re-install or add the variable manually. To do this manually
   i. Access environmental variable on Windows system, right-click @ My Computer (or This PC on some system) -> Properties -> Advanced System Settings -> Advanced Tab -> Environmental Variables
   ii. Prepend C:/MingW/bin; to PATH variable by editing PATH variable in either User variable for $USER or System variable.
iii. OK -> Apply

- Linux

```
./Configure --prefix=/usr/local shared
```

f) Compile and install

```
make && make install
```

### B.3 Python module (aeadtvgen)

The distribution package for `aeadtvgen` can be found as a wheel (*.whl) package under `$root/software/aeadtvgen/dist` folder. To install, type

```
python -m pip install _PACKAGED_MODULE_.whl
```
Bibliography


