GMU Hardware API for Authenticated Ciphers*

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

Electrical and Computer Engineering Department
George Mason University
Fairfax, Virginia 22030
email: {ehomsiri, wdiehl, aferozpu, ffarahma, masharif2, kgaj}@gmu.edu

Abstract. In this paper, we propose a universal hardware API for authenticated ciphers, which can be used in any future implementations of authenticated ciphers submitted to the CAESAR competition. A common interface and communication protocol would help in reducing any potential biases, and would make the comparison in hardware more reliable and fair. By design, our proposed API is equally suitable for hardware implementations of authenticated ciphers developed manually (at the register-transfer level), and those obtained using high-level synthesis tools. Our implementation of the proposed interface and communication protocol includes universal, open-source pre-processing and post-processing units, common for all CAESAR candidates. Apart from the full documentation, examples, and the source code of the pre-processing and post-processing units, we are making available in public domain a) a universal testbench to verify the functionality of any CAESAR candidate implemented using the GMU hardware API, b) a Python script used to automatically generate test vectors for this testbench, c) VHDL wrappers used to determine the maximum clock frequency and the resource utilization of all implementations, and d) RTL VHDL source codes of high-speed implementations of AES and the Keccak Permutation F. We hope that the existence of these resources will substantially reduce the time necessary to develop hardware implementations of all CAESAR candidates for the purpose of evaluation, comparison, and future deployment in real products.

1 Motivation

The CAESAR competition [1], launched in 2014, aims at identifying a portfolio of future authenticated ciphers with security, performance, and flexibility exceeding that of the current standards, such as AES-GCM [2] and AES-CCM [3].

Although, security is commonly accepted to be the most important criterion in all cryptographic contests, it is rarely by itself sufficient to determine a winner. This is because multiple candidates generally offer adequate security, and a trade-off between security and performance must be investigated.

The focus of this paper is to facilitate the comparison of modern authenticated ciphers in terms of their performance and cost in hardware, and in particular in FPGAs, All Programmable Systems on Chip, and ASICs. As a starting point for such a comparison we propose defining hardware API, composed of the specification of an interface of the authenticated cipher core, and the communication protocol describing the exact format of all inputs and outputs, as well as the timing dependencies among all data and control signals passing through the specified interface.

Similarly to the case of previous contests, software implementations of the CAESAR candidates are being compared using a uniform API, clearly defined in the call for submissions [1]. So far, no similar hardware API has been proposed, not to mention accepted by the cryptographic community.

As a result any attempt at the comparison of existing hardware implementations is highly dependent on specific assumptions about the hardware API, made independently by various hardware designers. These assumptions can have potentially a very high influence on all major performance measures of the developed implementations.

Additionally, hardware API is typically much more difficult to modify than software API, making any last minute standardization efforts and code adjustments highly inefficient and questionable.

* This work is supported by NSF Grant #1314540

Therefore, there is a clear need for a proposal regarding a uniform hardware API, which could be further modified and improved using a feedback from the cryptographic community, and eventually endorsed by the CAESAR Committee, and adopted by majority of future hardware developers. The goal of our paper is to address this issue by providing the exact specification of the proposed interface, as well as multiple supporting materials, such as open-source codes of pre-processing and post-processing units, a universal testbench, and uniform ways of generating optimized results.

2 Proposed Features

The proposed features of our hardware API are as follows:

- inputs of arbitrary size in bytes (but a multiple of a byte only)
- size of the entire message/ciphertext does not need to be known before the encryption/decryption starts (unless required by the algorithm itself)
- wide range of data port widths, $8 \leq w \leq 256$
- independent data and key inputs
- simple high-level communication protocol
- support for the burst mode
- possible overlap among processing the current input block, reading the next input block, and storing the previous output block
- storing decrypted messages internally, until the result of authentication is known
- support for encryption and decryption within the same core
- ability to communicate with very simple, passive devices, such as FIFOs
- ease of extension to support existing communication interfaces and protocols, such as AMBA-AXI4 – a de-facto standard for the System-on-Chip (SoC) buses [4], and PCI Express – high-bandwidth serial communication between PCs and hardware accelerator boards [5].

3 Previous Work

Several general-purpose interfaces for SoCs have been recently proposed, including but not limited to,

- AXI4, AXI4-Lite, AXI4-Stream (Advanced eXtensible Interface) from ARM [4]
- PLB (Processor Local Bus) and OPB (On-chip Peripheral Bus) from IBM [6]
- Avalon from Altera [7]
- FSL (Fast Simplex Link) from Xilinx Inc. [8], and
- Wishbone (used by opencores.org) from Silicore Corp. [9]

These interfaces define the meaning and role of all data and control signals of the communication buses, and the timing dependencies among them, but they do not describe the format of either data inputs or data outputs passing the boundaries of the cryptographic core.

During the SHA-3 contest [10], the first full hardware APIs, dedicated to hash functions, were proposed by:

- GMU [11], [12]
- Virginia Tech [13], and
- University College Cork [14].

Our current proposal is partially based on the experiences gained during the SHA-3 contest by designing and using the GMU interface and communication protocol for hash function cores.

The majority of interfaces used so far in the CAESAR competition have been quite minimalistic and candidate specific (e.g., [15]).

The only major exception was the adoption of the AXI4-Stream interface by the ETH student, Cyril Arnould, in his Master's Thesis defended in March 2015 [16]. However, the limitation of this solution was the use of non-uniform, algorithm-specific control ports, which make the corresponding cores mutually incompatible. Additionally, Arnaud's proposal does not contain any description of the exact formats of inputs and outputs of the cipher.

4 Specification

4.1 Interface

The general idea of our proposed interface for an authenticated cipher core (denoted by AEAD) is shown in Fig. 1. The interface is composed of three major data buses for

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

as well as the corresponding handshaking control signals, named *valid* and *ready*. The *valid* signal indicates that the data is ready at the source, and the *ready* signal indicates that the destination is ready to receive them.

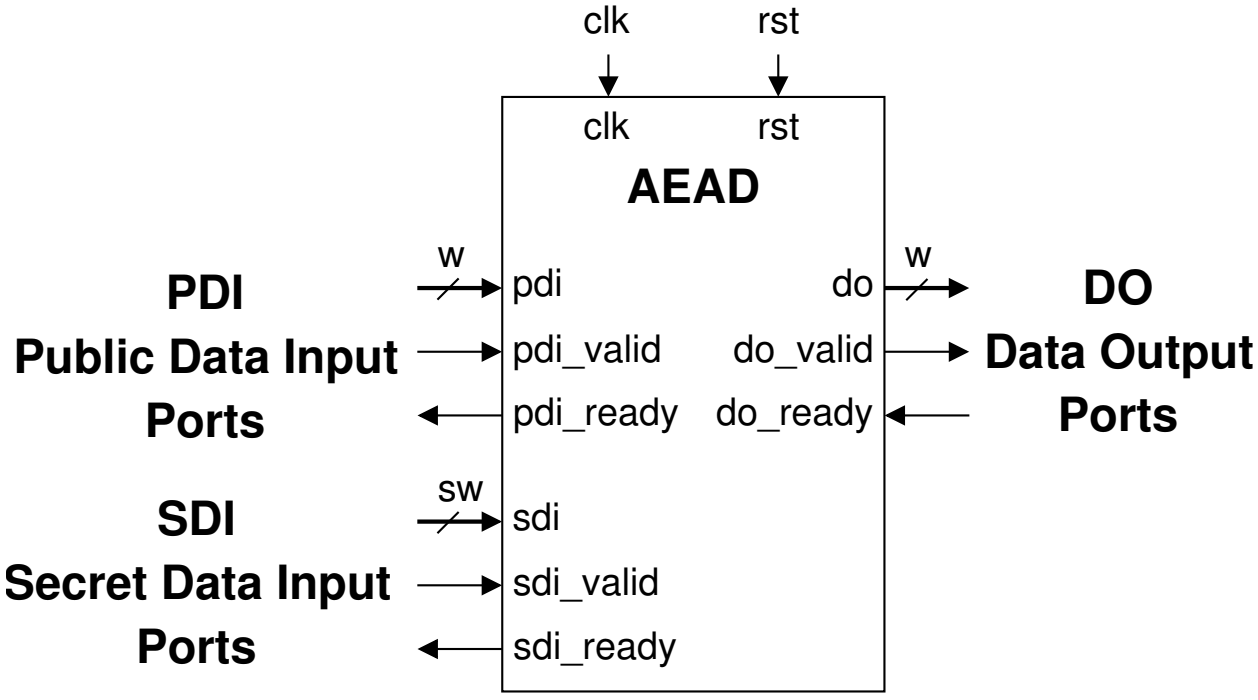


Fig. 1: AEAD Interface

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

The handshaking signals are a subset of major signals used in the AXI4-Stream interface. As a result AEAD can communicate directly with the AXI4-Stream Master through the Public Data Input, and with the AXI4-Stream Slave through the Data Output, as shown in Fig. 2. At the same time, AEAD is also capable of communicating with much simpler external circuits, such as FIFOs, as shown in Fig. 3.

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

An additional advantage of using FIFOs at all data ports is their potential role as suitable boundaries between the two clock domains, used for communication and computations, accordingly. This role is facilitated by the use of separate read and write clocks, shown in Fig. 3 as rd_clk and wr_clk, accordingly. All FIFOs mentioned in our description are assumed to operate in the standard mode (as opposed to the First-Word Fall-Through mode).

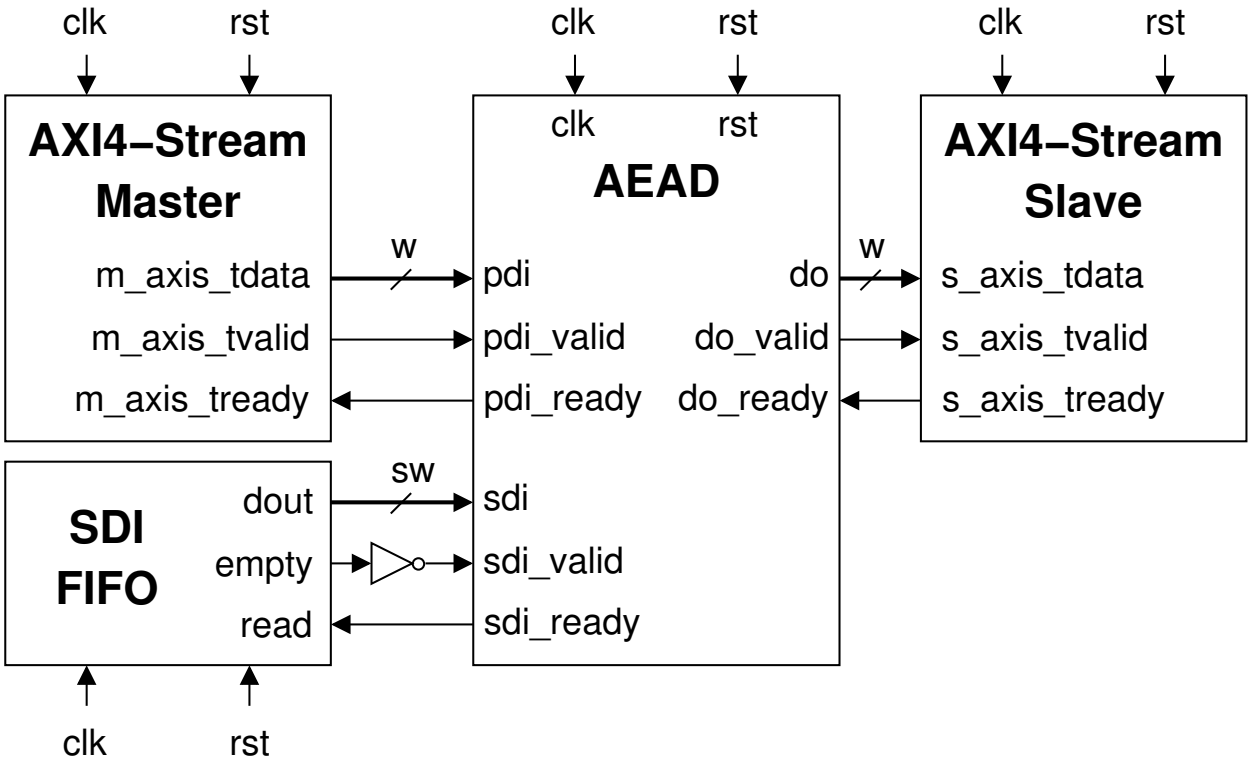


Fig. 2: Typical external circuits: AXI4 IPs

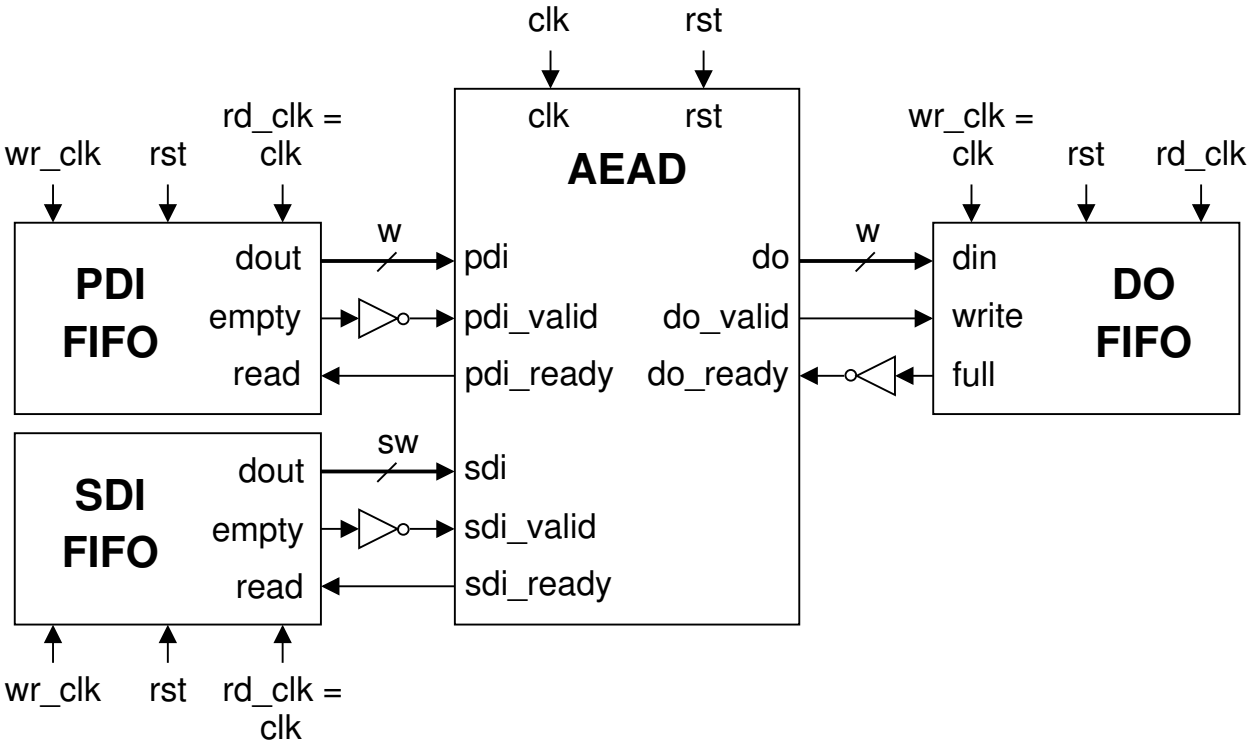


Fig. 3: Typical external circuits: FIFOs

4.2 Communication Protocol

All typical inputs and outputs of an authenticated cipher are shown in Fig. 4. Npub denotes Public Message Number, such as Nonce or Initialization Vector. Nsec denotes Secret Message Number, which was recently introduced in some authenticated ciphers. Both Npub and Nsec are typically assumed to be unique for each message encrypted using a given key. In our hardware API, the Invalid output is assumed to be a single word composed of all ones.

All inputs to encryption, other than a key, are optional, and can be omitted. If a given input is omitted, it is assumed to be an empty string.

The proposed format of the Secret Data Input is shown in Fig. 5. The entire input starts with an instruction, which in case of SDI is limited to Load Key. The instruction is followed by one or two segments. Each segment starts with a separate header, describing its type and size. In case of SDI, the only allowed segment types are: Key and Nsec (Secret Message Number).

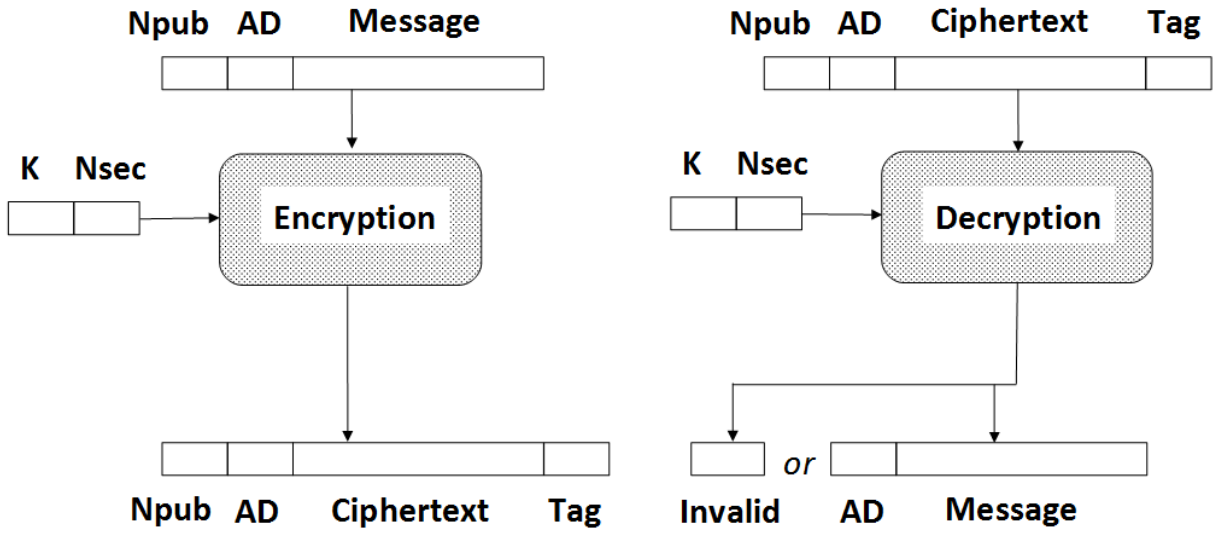


Fig. 4: Input and Output of an Authenticated Cipher. Notation: Npub - Public Message Number, Nsec - Secret Message Number, AD - Associated Data

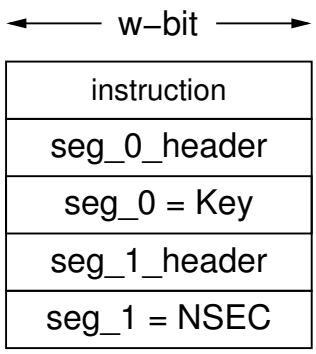
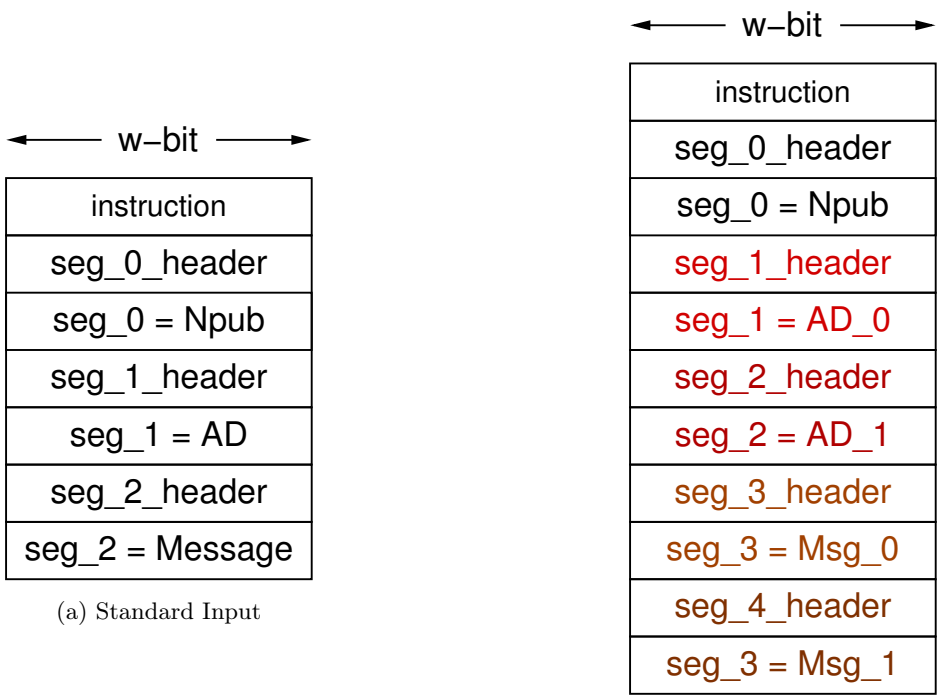


Fig. 5: Format of Secret Data Input



(a) Standard Input

(b) Extended Input

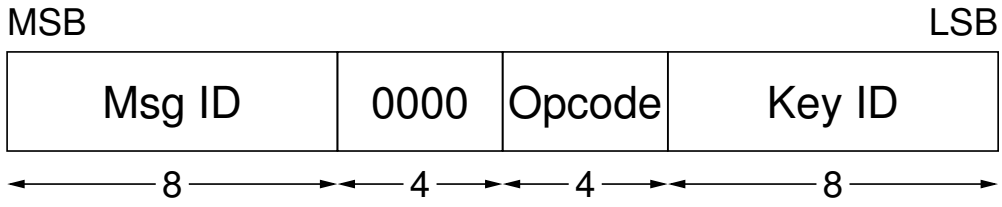
Fig. 6: Format of Public Data Input

The proposed format of the Public Data Input is shown in Fig. 6. The allowed instruction types are: Activate Key, Authenticated Encryption, and Authenticated Decryption. The Activate Key instruction, typically directly precedes the Authenticated Encryption or Authenticated Decryption instruction. PDI is divided into segments. Segment types allowed during authenticated encryption include: Public Message Number (Npub), Associated Data (AD), and Message. Segment types allowed during authenticated decryption include: Public Message Number (Npub), Associated Data (AD), Ciphertext, and Tag. Any segment type can be omitted, if it is not required by a given cipher. Public Message Number can only use one segment, as its size is typically quite small (in the range of 16 bytes). The Associated Data and Message can be (but do not have to be) divided into multiple segments (as shown in Fig. 6b).

The primary reasons for dividing AD and Message into multiple segments is that the full message size may be unknown when authenticated encryption starts, and/or the maximum single segment size (determined by the parameters of the implementation) is smaller than the message size (e.g., 2^{16} bytes in case of GMU supporting codes).

The instruction format is shown in Fig. 7. The Opcode field determines which operation should be executed next. The Msg ID field should be set to a unique message identifier, between 0 and 255. Similarly, the Key ID field should be set to a unique key identifier, between 0 and 255.

The segment header format is shown in Fig. 8. Seg Len is a size of a segment expressed in bytes. The field Info contains information about the Segment Type, as well as single-bit flags denoting the last segment of a particular type (EOT), and the last segment of the entire input (EOI), accordingly. In case of decryption, both the tag segment and the last segment before the tag must be marked as the last segment of the entire input (EOT=1 and EOI=1).

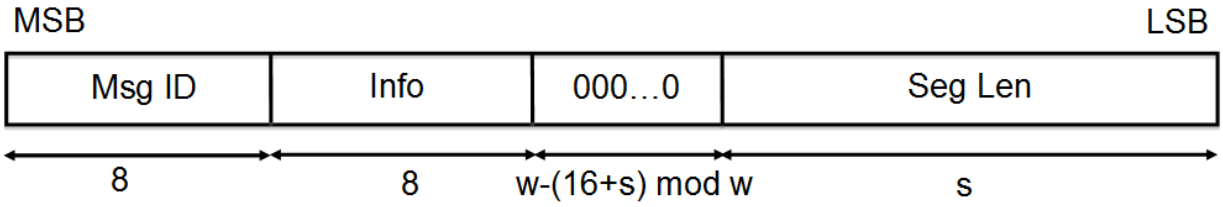


Divided into $\lceil 24/w \rceil$ words, starting from MSB

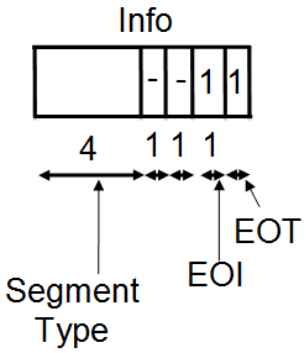
Opcode:

0010 – Authenticated Encryption
0011 – Authenticated Decryption
Others – Reserved
0100 – Load Key
0101 – Activate Key

Fig. 7: Instruction Format



Divided into $\lceil (16+s)/w \rceil$ words, starting from MSB.



Segment Type:

0000 – Reserved
0001 – Npub
0010 – AD
0011 – Message
0100 – Ciphertext
0101 – Tag
0110 – Key
1000 – Nsec

EOI = 1 if the last segment of input
0 otherwise

EOT = 1 if the last segment of its type (AD, Message, Ciphertext),
0 otherwise

Fig. 8: Segment Header Format

5 Supporting Codes for High-Speed Implementations

5.1 High-Level Block Diagram

The high-level block diagram of our proposed high-speed implementation of an authenticated cipher is shown in Fig. 9. AEAD consists of AEAD Core and the memory region. The memory region is separated from the AEAD Core for the ease of benchmarking.

The AEAD Core consists of the following three primary units: PreProcessor, PostProcessor, and CipherCore. Supporting codes for PreProcessor, PostProcessor, and the memory region are provided as a part of the GMU HW API distribution.

Bypass FIFO is a standard FIFO used for holding public input data that should be transferred to the output module unchanged, e.g., segment headers and associated data. This data is held in the Bypass FIFO for a short period of time until the PostProcessor is ready to receive it.

AUX FIFO is an auxiliary FIFO, operating in the standard mode, used to store a decrypted message until this message is either fully authenticated or found invalid.

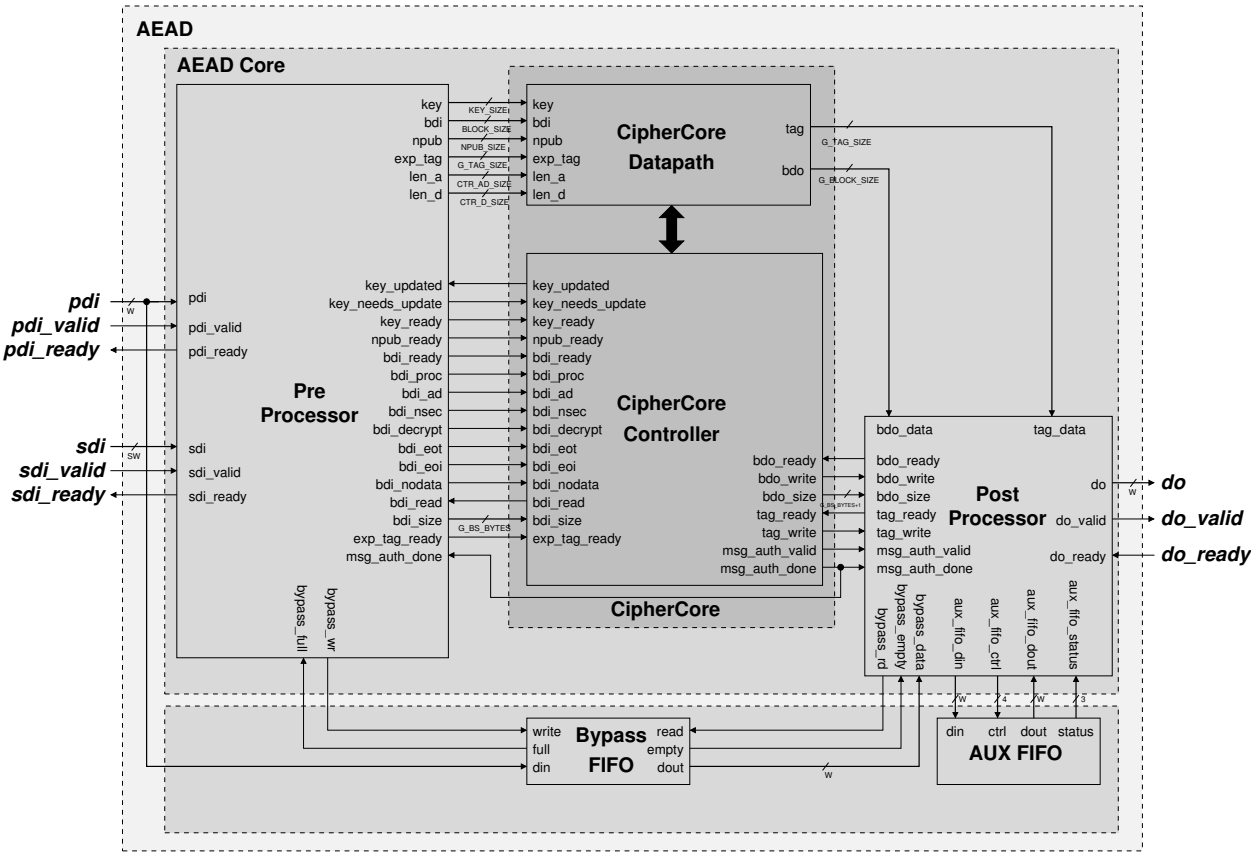


Fig. 9: High-level block diagram of a high-speed implementation

5.2 PreProcessor and PostProcessor

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.

The PostProcessor is responsible for the following tasks:

– clearing any portions of output blocks not belonging to ciphertext or plaintext
– Parallel-In-Serial-Out conversion of output blocks into words
– formatting output words into segments
– storing decrypted messages in AUX FIFO, until the result of authentication is known, and
– generating an error word if authentication fails.

Our goal is to assure the following features of the supporting codes:

– Ease of use
– No influence on the maximum clock frequency of AEAD (up to 300 MHz in Virtex 7)
– Limited area overhead
– Clear separation between the core unit and internal FIFOs, including the
  • Bypass FIFO for passing headers and associated data directly to PostProcessor, and
  • AUX FIFO for temporarily storing unauthenticated messages after decryption.

The PreProcessor and PostProcessor cores are highly configurable using generics described in Appendix A. These generics can be used for example to determine:

– the widths of the pdi, sdi, and do ports,
– the size of the message/ciphertext block, key, nonce, and tag,
– padding for the associated data and the message, and
– types and order of segments expected by a particular cipher.

The detailed specification of all ports of the PreProcessor or PostProcessor units is provided in Appendix B and Appendix C.

The way of loading and activating a new key by the PreProcessor is described below:

For the first message and the subsequent key change, a new key must be loaded into the PreProcessor via the SDI port first. This can be done by providing the Load Key instruction. A typical key loading sequence of words is shown below:

```
# 001 : Instruction(Opcode=Load key)
INS = 0104010000000000
# 001 : SgtHdr (Size=  16) (EOI=1)(EOT=1)(SgtType=Key)
HDR = 0163000000000010
DAT = D7B1CB5221D16D92
DAT = BB910D157C6F1C04
```

In this example, the first word specifies the Load Key instruction. The second word specifies that the subsequent data segment is of the key type, with the size of 16 bytes (128 bits). This segment is also the end-of-type and the end-of-input segment. The next two words consist of the data representing the key.

Before the new key becomes active, it must be activated via the PDI port first. This mechanism facilitates the synchronization between the two input ports. It also allows loading a new key without interfering with the key that is being used. A typical key activation process is shown below:

```
# 001 : Instruction  (Opcode=Activate key)
INS = 0105010000000000
```

This word must be applied before any other instruction word.

5.3 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. Permutation have been developed. 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. All resources are available at the GMU ATHENa website

https://cryptography.gmu.edu/athena

5.4 Using Supporting Codes

A typical hardware development process based on the use of our supporting codes requires a designer to modify the default values of generics in the AEAD_Core to match the needs of a targeted algorithm, and then develop the CipherCore based on user's preferences (see Section 6).

The primary benefit of using our supporting codes is that the designers can focus on developing the CipherCore specific to a given algorithm, without worrying about the functionality common for multiple authenticated ciphers. Additionally, the interface of the CipherCore has full-block widths for all major data buses, which should substantially simplify the development effort.

6 The Development of CipherCore

The development of the CipherCore is left to individual designers, and can be performed using their own preferred design methodology. The detailed meaning of all ports is given in Appendix B and Appendix C. Answers to the following frequently asked questions might be helpful in developing an implementation fully compatible with the GMU Hardware API:

– **How to load a new key and activate it?**

In order to properly handle keys, the cipher core should monitor the key_needs_update and key_ready inputs, and provide key_updated output at the appropriate time. The circuit should operate as follows:

After reset, key_needs_update and key_ready are low. At this point, a new key can be loaded into the input processor at any time.

After the new key is loaded using the SDI port, key_ready goes high. After the instruction ACTIVATE_KEY is received at the PDI port, the key_needs_update goes high. Note: The above two events can occur in an arbitrary order.

After key_ready and key_needs_update are both high, and the cipher core is either in the period between reset and the first input, or in the period between two consecutive inputs, the cipher core should read the new key. After the key is read, key_updated signal should be set to high. The key_updated signal should be deactivated at the end of processing of the current input.

If a user wants to use the same key for the subsequent input data, ACTIVATE_KEY instruction can be omitted from the PDI input port. In this case, the processing of new data will start as soon as an instruction describing the way of processing a new input is decoded (which is indicated by bdi_proc set to high).

In summary, the cipher core should monitor the key_needs_update port prior to processing any new input. If key_needs_update is high, the cipher core should wait for key_ready=1, and then read the new key, and acknowledge its receipt using the key_updated output. If key_needs_update is low and the first instruction describing the way of processing a new input is decoded (bdi_proc=1), then the cipher core should move directly to processing a new input using a previous key. If none of these two events is detected, the cipher core should remain in the same state. The described behavior is shown in

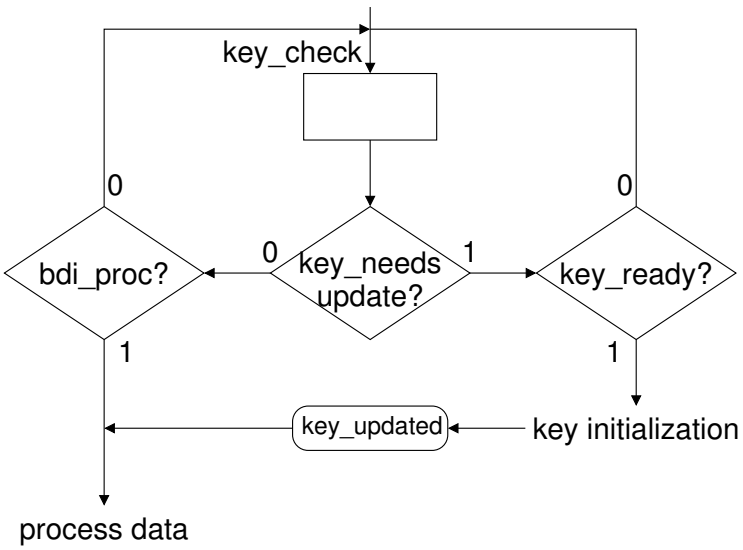


Fig. 10: A part of the Algorithmic State Machine (ASM) chart describing a way in which the CipherCore Controller may handle key loading and key activation

Fig. 10. The key initialization and process data are two separate states that operate depending on the requirements of a specific cipher.

– **How do I know if there is no associated data or data?**

Empty associated data can be derived during loading the first data block via bdi_ad signal. If the first data block is not of type AD (bdi_ad=0), there is an empty associated data.
Empty data can be derived while processing the last block of associated data. If the last block of AD (bdi_ad=1 and bdi_eot=1) is also the last block of input (bdi_eoi=1), there is an empty data/encrypted data.

– **How to specify a decryption operation?**

The instruction word loaded via the PDI port contains an operational code that should be monitored in the PreProcessor. An example of an Authenticated Decryption instruction is shown below:

```
# 002 : Instruction (Opcode=AEAD decrypt)
INS = 0203020000000000
```

When either a decryption (*b'0001'*) or authenticated decryption (*b'0011'*) operation is specified, the bdi_decrypt signal goes high. The cipher core needs to monitor the bdi_decrypt in order to determine whether the current input should be processed using a decryption or encryption operation.

– **How to communicate with the PostProcessor?**

The PostProcessor is capable of understanding outputs from the cipher core. The cipher core only needs to provide the encrypted data/decrypted data and the tag to the bdo_data and tag ports, respectively, whenever these values are available. The PostProcessor uses the segment header information transmitted via Bypass FIFO to understand the order and organizes outputs accordingly. Typically, the PostProcessor receives data and organizes the output from Bypass FIFO and the cipher core's datapath into the following format.

For encryption operation:

```
Source            | Word    : Data type
==========================================================
bypass FIFO       | Word 1  : Instruction header (Decryption)
bypass FIFO       | Word 2  : Segment header
bypass FIFO       | Word 3  : AD
bypass FIFO       |
bypass FIFO       | Word A  : Segment header
Datapath          | Word A+1: Ciphertext
Datapath          |
Output Processor  | Word B  : Segment header (Tag)
Datapath          | Word B+1: Tag_0
Datapath          | Word B+2: Tag_1
```

For decryption operation:

```
Source            | Word    : Data type
==========================================================
bypass FIFO       | Word 1  : Segment header
bypass FIFO       | Word 2  : AD
bypass FIFO       |
bypass FIFO       | Word A  : Segment header
Datapath          | Word A+1: Decrypted message
Datapath          |
```

Note: During decryption, the cipher core needs to calculate and pass to the PostProcessor all blocks of decrypted message followed by all blocks of tag. The Bypass FIFO handles all headers (including the instruction header) and all blocks of AD. The instruction header, public message number, and tag segments are omitted in the final output sent by the PostProcessor to its output DO.

– **Why does my decrypted data come out slower than an encrypted data?**

During an authenticated decryption process, the PostProcessor stalls output until the msg_auth_done is activated. All output blocks generated by the PostProcessor are placed within a buffer until the tag is verified. Once the tag is verified, the stored data are released. As a result, the outputs of the decrypted data are delayed. If the result of verification is incorrect, the PostProcessor sends just a single error code, consisting of all ones, to the output DO port.

– **How can I keep information signals valid until the end of the block processing?**

Delaying the acknowledge signal (bdi_read) from the cipher core until a block is processed can achieve the desired behavior. However, this approach can delay the loading of next data block as the PreProcessor is stalled until it receives the acknowledge signal. As a result, creating a status register to store the required information within the cipher core itself is better. Afterwards, user can examine these registers at leisure.

– **What should I do when an output port is not being used?**

When an output port is not being used, one can simply ignore the output port or assign the value open in the port map instantiation. For instance, bdi_size ⇒ open, would tell the synthesis tool to ignore this output port.

7 Universal Testbench and Test Vector Generation

Our supporting codes include the

– universal testbench for any authenticated cipher core that follows the GMU Hardware API
– AETVgen: **A**uthenticated **E**ncryption **T**est **V**ector **gen**eration script
– C codes of the CAESAR candidates from the SUPERCOP distribution.

The testbench is located in the folder: `$root/src_tb`,
the test vector generation script in: `$root/software/AETVgen`,
and the C codes of CAESAR candidates in `$root/software/CAESAR`.

AETVgen generates a comprehensive set of test vectors for a specific CAESAR candidate, based on the reference C code of that candidate, and additional parameters, provided by the user.

7.1 Compiler and interpreter prerequisites

Windows

- **MinGW with MSYS**
  Download and install the latest version from http://www.mingw.org. MSYS should be included in the installation package.
  Note: MSYS is the console for MinGW in Windows
- **Python v3.4+**
  Download and install the latest Python distribution package from https://www.python.org.
  Note: The GMU code has been tested with v3.4

Linux

- **Python v3.4+**

7.2 Python package prerequisites

AETVGen requires two Python packages:

- PyCrypto
- cffi

In Windows, the installation of these two packages can be done by calling the *easy_install* script, typically located in `C:/Python/Scripts`.

```
C:\Python\Scripts> easy_install PyCrypto
C:\Python\Scripts> easy_install cffi
```

In Linux, the installation procedure of these packages is dependent on the package manager used by the user. As a result, we do not cover this issue in detail.

7.3 Quick User Guide

This section provides a step-by-step quick user guide.

1. Create shared libraries (*.dll in Windows and *.so in Linux)
   (a) In console, navigate to the CAESAR folder (`$root/software/CAESAR`).
   Note: For Windows, perform this step using *msys* console
   (b) type
   ```
   make
   ```
2. Generate the script using a pre-defined settings
   (a) Modify the main method in `$root/software/AETVgen/gen.py` using your favorite editor to call a pre-defined method.
   (b) In console, type
   ```
   gen.py
   ```
3. Three test vector files (pdi.txt, sdi.txt and do.txt) should be generated in *AETVgen* folder.

Pre-defined Methods have the following format:

```
$Method($NumberTestVector, $TestMode, $Verbose, $Decrypt)
```

where,

- $\$Method$ is the name of the pre-defined method. Typically the name of the algorithm is used, i.e. AES_GCM.
- $\$NumberTestVector$ is the number of test vectors to be generated by the script.
- $\$TestMode$ is the method in which the AETVgen will generate the test vectors. Currently, the following modes are supported:
  - *False*: Generate randomized test vector based on the given parameters.
  - *0*: Generate test vectors with 0x5555.. for key, 0xA0A0... for AD, 0xFFFF... for data.
  - *1*: Similar to 0 except input data is randomized

  For $\$TestMode = 0$ and 1, the test vectors will produce a pre-defined routine following the description provided below:

```
Msg  1 = AEAD encrypt [AD Size= 1,          Msg Size=0]
Msg  2 = AEAD decrypt [AD Size= 1,          Msg Size=0]
Msg  3 = AEAD encrypt [AD Size= 0,          Msg Size=1]
Msg  4 = AEAD decrypt [AD Size= 0,          Msg Size=1]
Msg  5 = AEAD encrypt [AD Size= 1,          Msg Size=1]
Msg  6 = AEAD decrypt [AD Size= 1,          Msg Size=1]
Msg  7 = AEAD encrypt [AD Size= blockSize,  Msg Size=blockSize]
Msg  8 = AEAD decrypt [AD Size= blockSize,  Msg Size=blockSize]
Msg  9 = AEAD encrypt [AD Size= blockSize-1,Msg Size=blockSize-1]
Msg 10 = AEAD decrypt [AD Size= blockSize-1,Msg Size=blockSize-1]
Msg 11 = AEAD encrypt [AD Size= blockSize+1,Msg Size=blockSize+1]
Msg 12 = AEAD decrypt [AD Size= blockSize+1,Msg Size=blockSize+1]
Msg 13 = AEAD encrypt [AD Size= blockSize*2,Msg Size=blockSize*2]
Msg 14 = AEAD decrypt [AD Size= blockSize*2,Msg Size=blockSize*2]
Msg 15 = AEAD encrypt
    [AD  Size= X where 0<X<blockSize*2 and X /= Y,
     Msg Size= Y where 0<Y<blockSize*2]
Msg 16 = AEAD decrypt
    [AD  Size= X where 0<X<blockSize*2 and X /= Y,
     Msg Size= Y where 0<Y<blockSize*2]
Msg 17 = AEAD encrypt [AD Size= blockSize*3,Msg Size=blockSize*3]
Msg 18 = AEAD decrypt [AD Size= blockSize*3,Msg Size=blockSize*3]
Msg 19 = AEAD encrypt [AD Size= blockSize*4,Msg Size=blockSize*4]
Msg 20 = AEAD decrypt [AD Size= blockSize*4,Msg Size=blockSize*4]
...
```

- $\$Verbose$ prints output from the modified CAESAR program that is encapsulated by the `#ifdef DBG ... #endif` macro. Accepted values are either $True$ or $False$.
- $\$Decrypt$ performs decryption after encryption. By default, AETVgen only generates test vectors for the encryption operation. This flag should be used in conjunction with the $\$Verbose$ operation to view the output of decryption operation. Accepted values are either $True$ or $False$.

7.4 Debugging

Oftentimes, it maybe necessary to view the intermediate state of the encryption or decryption operation. It is up to the user to add the necessary debugging information to the C source code. This can be done by printing values of the relevant variables into the screen. It is recommended to surround a print statement with the `#ifdef` preprocessor directive, so that when $\$Verbose$ is set to False, this information will not be printed out, e.g.,

```
#ifdef DBG
  printf("%02X", state);
#endif
```

Note: The user will need to recompile the shared library again in order for the changes in the source codes to take effect.

7.5 Addition of a new library

The script currently supports a limited set of CAESAR libraries. In order to add an additional library to the script, one needs to perform modification in C and Python. It must be noted that the instruction in this section assumes that the new library follows the CAESAR software API.

C-related modification

– Modification of the header files and macros in `encrypt.c` file, located in the reference implementation (ref) folder of the targeted algorithm

1. Headers

```
// Old
#include "../../crypto_aead.h"
// New
#include "../../crypto_aead.h"
#include "../../dll.h"
```

2. Insert the pre-defined macros, `EXPORT`, in front of the primary function calls, crypto_aead_encrypt() and
crypto_aead_decrypt()

```
EXPORT int crypto_aead_encrypt(
    ...
)

EXPORT int crypto_aead_decrypt(
    ...
)
```

– Modification of the global Makefile located inside the `$root/CAESAR` folder. This can be done by inserting your new algorithm in the list of primitives at the top of the file as shown below:

```
PRIMITIVES = \
             $new_library \
```

Note: Do not forget to recompile the code according to the above instruction. You may also need to perform "Make clean" first.

Python-related modification

1. In the init() method of AETVgen.py, add a new ID to the list of supported CAESAR algorithms. The ID of the new algorithm should be unique and does not necessarily need to be the same as the name of the library.

```
    self.supportedCaesarAlg = [$new_algorithm_id, ... ]
```

2. Add appropriate parameters and the library that the new algorithm should be targeted at, e.g.:

```
  elif algorithm.lower() == "$new_algorithm_id":
    expTagSize          = 16
    expKeySize          = 16
    expNPUBSize         = 12
    lib                 = "$new_library"
    self.blockSize      = 16
```

3. In the *gen.py*, create a new method with pre-defined settings for ease of reuse:

```
def $new_algorithm_id(size, testMode, verbose, decrypt):
  args = {'stdout': False, 'sizePIO': 4, 'sizeSIO': 4,
          'sizeTag': 16, 'sizeKey': 16, 'sizeNPUB': 12,
          'verbose': verbose, 'testMode': testMode,
          'decrypt': decrypt}
  aetv = AETVgen.AETVgen('$new_algorithm_id', **args)
  aetv.genTestVectors(size)
```

4. Add a new method to *main*

```
if __name__ == '__main__':
  testMode = 1
  verbose  = False
  decrypt  = False
  $new_algorithm_id(20, testMode, verbose, decrypt)
```

Note: Replace $new_algorithm with the name of your algorithm, e.g. "AES_GCM".

8 Generation and Publication of Results

Generation of results is possible for AEAD, AEAD Core, and CipherCore (see Fig. 9). We strongly recommend generating results primarily for AEAD Core. This recommendation is based on the fact that

- CipherCore has an incomplete functionality and a full-block-width interface,
- Using AEAD may cause difficulty with setting BRAM usage to 0 (as often desired in order to easily calculate throughput to area ratio).

The results for Xilinx FPGAs can be generated using any of the development and benchmarking flows shown in Figs. 11, Figs. 12, and Figs. 13.

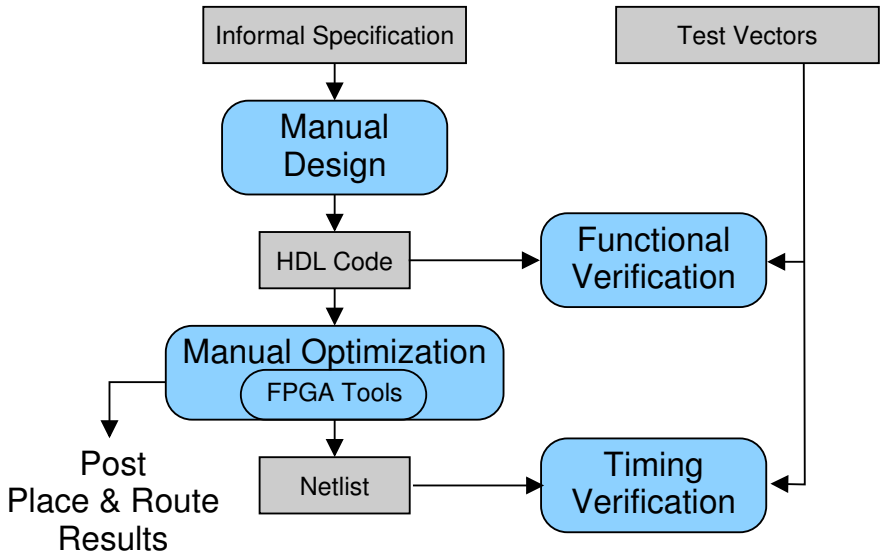


Fig. 11: Traditional Development and Benchmarking Flow

8.1 Wrappers

For Virtex 7 and Zynq, we recommend generating results using Xilinx Vivado [17], operating in the Out-of-Context (OOC) mode [18]. In this mode, no pin limit applies.

For Virtex 6 and below, we recommend using a simple wrapper, with 5 ports: clk, rst, sin, sout, piso_mux_sel, provided as a part of supporting files in the folder
`$roots/src_rtl/wrappers`

8.2 Optimization Strategies

For Virtex 7 and Zynq, we recommend the use of 25 default optimization strategies available in Xilinx Vivado. For Virtex 6 and below, we recommend using Xilinx ISE and ATHENa [19], [20]. For Altera FPGAs, we suggest using Altera Quartus II and ATHENa.

8.3 Overheads

The first preliminary results regarding an overhead introduced by extending CipherCore with AEAD Core are summarized in Fig. 14. As seen in this figure, the overhead does not exceed 18% even for the smallest investigated cipher cores, and reaches values in the range of 2-3% for the biggest cores.

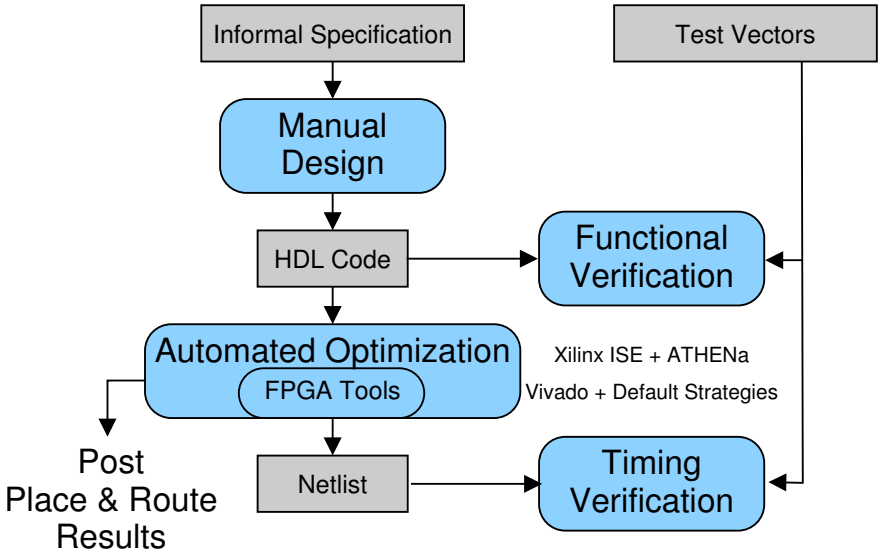


Fig. 12: Extended Traditional Development and Benchmarking Flow

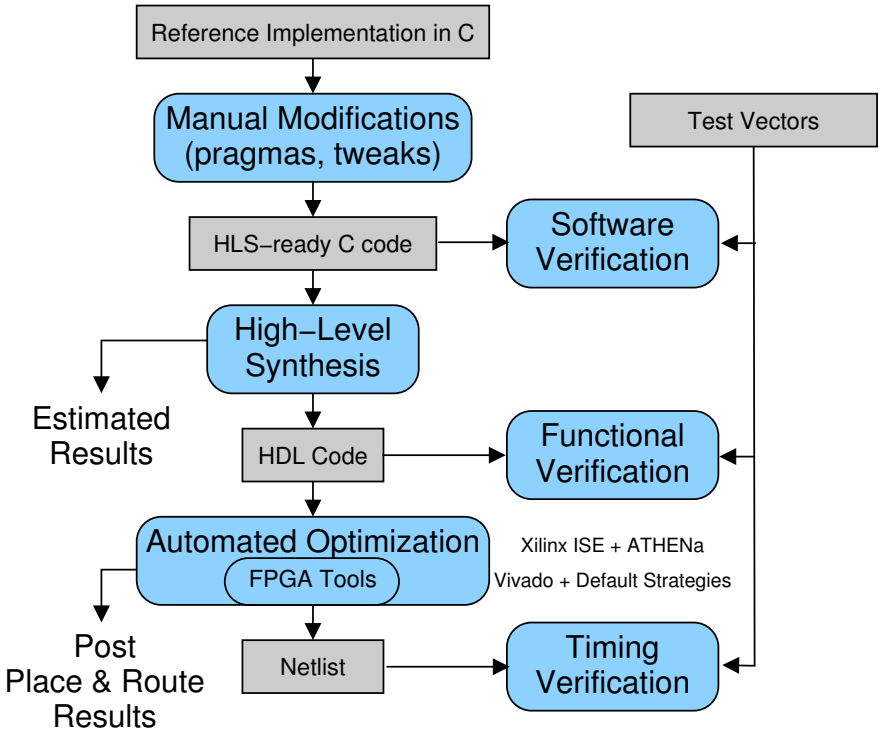


Fig. 13: Alternative HLS-Based Development and Benchmarking Flow

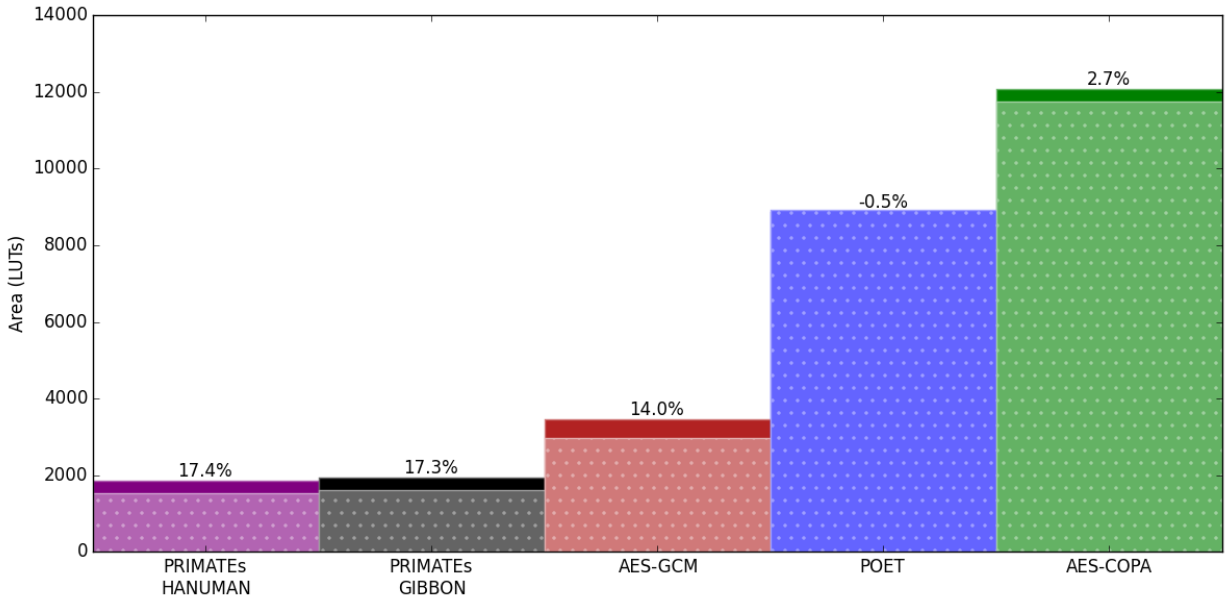


Fig. 14: AEAD Core vs. CipherCore Area Overhead of Virtex 6 FPGA

8.4 Database of Results

The ATHENa database of results for authenticated ciphers is available at http://cryptography.gmu.edu/athena under Results Database.

After receiving an account in the database, the designers can enter results by themselves. Additionally, the ATHENa Option Optimization Tool supports automatic generation of results suitable for uploading to the database.

9 Unsupported Features and Future Work

The currently unsupported features of the GMU Hardware API include:

- processing of Secret Message Number, Nsec
- full use of Message ID
- full use of Key ID.

The possible future extensions of the API and supporting codes include:

- detection and reporting of input formatting errors
- support for two-pass algorithms
- accepting inputs with padding done in software
- accepting inputs with key scheduling done in software
- support for multiple streams of data.

10 Conclusions

In this paper, we have described our proposal for a complete Hardware API for authenticated ciphers, including the interface and communication protocol. The design with the GMU Hardware API is facilitated by

- Detailed specification

- Universal testbench and Automated Test Vector Generation
- PreProcessor and PostProcessor Units for high-speed implementations
- Universal wrappers for generating results
- Source codes of AES and Keccak Permutation F
- Ease of recording and comparing results using ATHENa database.

The GMU proposal is open for discussion and possible improvements through better specification as well as better implementation of supporting codes.

References

1. CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness. (2014, Mar.) Cryptographic competitions. [Online]. Available: http://competitions.cr.yp.to/index.html
2. J. Salowey, A. Choudhury, and D. McGrew, "AES Galois Counter Mode (GCM) cipher suites for TLS," RFC 5288 (Proposed Standard), Aug 2008. [Online]. Available: https://tools.ietf.org/html/rfc5288
3. D. McGrew and D. Bailey, "AES-CCM cipher suites for TLS," RFC 6655 (Proposed Standard), July 2012. [Online]. Available: https://tools.ietf.org/html/rfc6655
4. ARM. AMBA Specifications. [Online]. Available: http://www.arm.com/products/system-ip/amba-specifications.php
5. PCI-SIG. Specifications. [Online]. Available: https://pcisig.com/specifications
6. IBM. Processor Local Bus (128-bit). [Online]. Available: https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/3BBB27E5BCC165BA87256A2B0064FFB4
7. Altera. (2015, March) Avalon Interface Specifications. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/mnl_avalon_spec.pdf
8. Xilinx. (2012, December) Logicore IP Fast Simplex Link (FSL) V20 Bus (v2.11f). [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/fsl_v20.pdf
9. OpenCores. (2010) Wishbone B4: WISHBONE System-on-Chip (SoC)Interconnection Architecture for Portable IP Cores. [Online]. Available: http://cdn.opencores.org/downloads/wbspec_b4.pdf
10. National Institute of Standards and Technology. (2014, Mar.) Third (Final) Round Candidates. [Online]. Available: http://csrc.nist.gov/groups/ST/hash/sha-3/
11. K. Gaj, E. Homsirikamol, and M. Rogawski, "Fair and Comprehensive Methodology for Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates Using FPGAs," in *Cryptographic Hardware and Embedded Systems, CHES 2010*, ser. Lecture Notes in Computer Science, S. Mangard and F.-X. Standaert, Eds., vol. 6225. Springer Berlin Heidelberg, 2010, pp. 264–278.
12. E. Homsirikamol, M. Rogawski, and K. Gaj, "Comparing hardware performance of fourteen round two SHA-3 candidates using FPGAs," Cryptology ePrint Archive, Report 2010/445, 2010.
13. Z. Chen, S. Morozov, and P. Schaumont, "A hardware interface for hashing algorithms," Cryptology ePrint Archive, Report 2008/529, 2008.
14. B. Baldwin, A. Byrne, L. Lu, M. Hamilton, N. Hanley, M. O'Neill, and W. P. Marnane, "A hardware wrapper for the SHA-3 hash algorithms," Cryptology ePrint Archive, Report 2010/124, 2010.
15. A. Moradi, "A Hardware Implementation of POET," Germany, Jan 2015. [Online]. Available: https://groups.google.com/forum/#!msg/crypto-competitions/j0goqKCqFMI/SYG5-61mEcwJ
16. C. Arnould, "Towards Developing ASIC and FPGA Architectures of High-Throughput CAESAR Candidates," Master's thesis, ETH Zurich, March 2015.
17. Xilinx. Vivado Design Suite. [Online]. Available: http://www.xilinx.com/products/design-tools/vivado.html
18. ——, *Vivado Design Suite User Guide: Hierarchical Design*, April 2015. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_1/ug905-vivado-hierarchical-design.pdf
19. K. Gaj, J.-P. Kaps, V. Amirineni, M. Rogawski, E. Homsirikamol, and B. Y. Brewster, "ATHENa – automated tool for hardware evaluation: Toward fair and comprehensive benchmarking of cryptographic hardware using FPGAs," in *20th International Conference on Field Programmable Logic and Applications - FPL 2010*. IEEE, 2010, pp. 414–421.
20. Cryptographic Engineering Research Group (CERG) at GMU. ATHENa: Automated Tool for Hardware EvaluatioN. [Online]. Available: https://cryptography.gmu.edu/athena/

Appendix A: Generics used by the PreProcessor and/or the PostProcessor

Table A1: Generics used by the PreProcessor and/or the PostProcessor

Pre-Processor	Post-Processor	Name	Default Value	Brief Definition
x	x	W	32	Public data input and Data output width (bits)
x		SW	32	Secret data input width (bits)
x		NPUB_SIZE	64	Npub size (bits)
x		ABLK_SIZE	64	Block size of associated data (bits)
x	x	DBLK_SIZE	128	Block size of message and ciphertext (bits)
x		KEY_SIZE	128	Key size (bits)
x	x	TAG_SIZE	128	Tag size (bits)
x	x	BS_BYTES	4	The number of bits required to hold the size of an incomplete block, expressed in bytes = $log_2\lceil max(ABLK\_SIZE, DBLK\_SIZE)/8\rceil$
x		PAD	0	Enable 10* padding to a multiple of a block size.
x		PAD_STYLE	1	[0] = No actual padding, the unit will produce bdi_pad_loc, [1] = Pad10*, [2] = ICEPOLE's specific mode, [3] = Keyak's specific mode
x		PAD_AD	1	(Active when PAD=1) [0] Disable padding for AD segment [1] Enable padding for AD segment
x	x	PAD_D	1	(Active when PAD=1) [0] Disable padding for data segment [1] Enable padding for data segment [2] Enable padding for data segment and add an additional block if message size % block size = 0
x		CTR_AD_SIZE	64	The width of the len_a port representing the length of associated data
x		CTR_D_SIZE	64	The width of the len_d port representing the length of data (the length of message for encryption, and the length of ciphertext for decryption
x		PLAINTEXT_MODE	0	Plaintext input handling mode. See Table A2 for more details.
x	x	CIPHERTEXT_MODE	0	Ciphertext output handling mode. See Table A3 for more details.
x	x	REVERSE_DBLK	0	[0] Ciphertext block arrives as normal [1] Ciphertext block arrives in a reversed order (last block first).

Table A2: Extended description of PLAINTEXT_MODE. Note: (*) default option. (**) Npub related signals are disabled.

Generic Value	Mode	Description
0*	N_A_M	Separate Nonce, Associated Data, and Message segments.
1**	NA_M	The Associated Data segment contains Nonce concatenated with Associated Data.
2**	AN_M	The Associated Data segment contains Associated Data concatenated with Nonce.
3	N_A_M_A	Separate Nonce, Associated Data - Header, Message, and Associated Data - Trailer segments.

The operations specific to each CIPHERTEXT_MODE value are further described below:

Table A3: Extended description of CIPHERTEXT_MODE. Note: (*) default option. (**) not yet supported. (***) partially supported.

Generic Value	Mode	Description
0*	C_T	Separate Ciphertext and Tag segments.
1**	CT	The Ciphertext segment contains Ciphertext concatenated with Tag.
2***	Cexp_T	Separate Ciphertext and Tag segments. Ciphertext segment is expanded to a multiple of the block size.

(0) CT

during encryption

- len_d = $|M|$
- The tag output of the Datapath is not used
- The output processor does not wait for 1 at the tag_write output of the Datapath
- The size of C in the ciphertext segment header = $|M| + |T|$

during decryption

- The size of C in the ciphertext segment header = $|M| + |T|$
- len_d = $|M| + |T|$ ($|M|$ is calculated inside of the datapath)
- The exp_tag input of the Datapath is not used.
- The exp_tag_ready input of the Controller is not used. Datapath

(1) C_T

during encryption

- len_d= $|M|$
- The tag output of the Datapath is used
- The output processor waits for 1 at the tag_write output of the Datapath
- The size of C in the ciphertext segment header = $|M|$

during decryption

- The size of C in the ciphertext segment header = $|M|$
- len_d = $|M|$
- The exp_tag input of the Datapath is used.
- The exp_tag_ready input of the Controller is used.

(2) Cexp_T

during encryption

- len_d = $|M|$
- The tag output of the Datapath is used
- The output processor waits for 1 at the tag_write output of the Datapath
- The size of C in the ciphertext segment header = $|M|$ but the output processor expects $block\_size * \lceil |M|/block\_size \rceil$ bits of the ciphertext

during decryption

- The size of C in the ciphertext segment header = $|M|$, but the input processor reads and passes to the Datapath $block\_size * \lceil |M|/block\_size \rceil$ bits of the ciphertext
- len_d = $|M|$
- The exp_tag input of the Datapath is used.
- The exp_tag_ready input of the Controller is used.

Appendix B: PreProcessor Ports

Table B4: PreProcessor Ports

Name	Direction	Width	Definition
clk	in	1	Global clock signal
rst	in	1	Global reset signal (synchronous)
pdi	in	W	Public data input
pdi_valid	in	1	Public data input valid
pdi_ready	out	1	Public data input ready
sdi	in	SW	Secret data input
sdi_valid	in	1	Secret data input valid
sdi_ready	out	1	Secret data input ready
key	out	KEY_SIZE	Key data
bdi	out	DBLK_SIZE	Input block data
npub	out	NPUB_SIZE	Public message number (Npub). This port is inactive if PLAINTEXT_MODE = 1 or 2.
exp_tag	out	TAG_SIZE	Expected tag data. This output is valid for authenticated decryption operation.
len_a	out	CTR_AD_SIZE	*[SEGMENT INFO]* Length of authenticated data in bytes (used in some algorithms)
len_d	out	CTR_D_SIZE	*[SEGMENT INFO]* Length of data in bytes (used in some algorithms)
key_ready	out	1	Key ready signal. This signal indicates that the key is available.
key_needs_ update	out	1	Key needs an update signal. This signal indicates to the crypto core that the key should be updated (i.e., new round keys calculated). The crypto core should update the key before the next input is processed.
key_updated	in	1	Return signal from the crypto core acknowledging that the key has been updated
npub_ready	out	1	*[INPUT INFO]* Npub ready signal. This port is inactive if PLAINTEXT_MODE = 1 or 2.
bdi_ready	out	1	Block ready signal
bdi_proc	out	1	*[INPUT INFO]* Input processing. This signal indicates that the current input is being processed. This signal will remain high from the moment of decoding an instruction describing the way of processing a given input to the moment when the last block of the input has been fully processed. This signal is low after reset and in any interval between two consecutive inputs (including the time of decoding and executing any Activate Key instructions).
bdi_ad	out	1	*[SEGMENT INFO]* Input block is an authenticated data
bdi_nsec	out	1	*[SEGMENT INFO]* Input block is a secret message number
bdi_decrypt	out	1	*[INPUT INFO]* Current input should be decrypted.
bdi_pad	out	1	*[BLOCK INFO]* Current block has been padded by an external program/unit [Note: Not fully implemented]

bdi_eot	out	1	*[BLOCK INFO]* Current block is the last block of its type. There may be more data blocks belonging to different segments following this block. For instance, if the current block is IV, the subsequent block is generally either of type message or authenticated data.
bdi_eoi	out	1	*[BLOCK INFO]* Current block is the last block of the given public data input (i.e., all segments associated with a given message or ciphertext). This signifies that the following block will be the first block of the group of segments associated with another message or ciphertext..
bdi_nodata	out	1	*[BLOCK INFO]* Current block has no data (it contains only padding)
bdi_read	in	1	Return signal from the crypto core indicating that data block is being read
bdi_size	out	BS_BYTES	*[BLOCK INFO]* The size of the current block in bytes (0 for full blocks)
bdi_valid_bytes	out	DBLK_SIZE/8	*[BLOCK INFO]* Number of valid bytes of BDI.
bdi_pad_loc	out	DBLK_SIZE/8	*[BLOCK INFO]* Pad location. An active bit indicates the starting point of the padding location. Note: Must set PAD=1 (set PAD_STYLE=0 if no padding is required)
msg_auth_done	in	1	Message authentication completion signal. This signal indicates that the comparison is completed for authenticated decryption and data in exp_tag port can be overwritten.
exp_tag_ready	out	1	Expected tag (exp_tag) ready signal.
bypass_fifo_full	in	1	Bypass FIFO indicating that it is full
bypass_fifo_wr	out	1	Write signal to bypass FIFO

[INPUT INFO]. Auxiliary signal that remains valid until a given message is fully processed. Deactivation is typically done at the end of input.
[SEGMENT INFO]. Auxiliary signal that remains valid for the current segment. The value changes when a new segment is received via the PDI data bus. For length information, the values are reset for every new block of data.
[BLOCK INFO]. Auxiliary signal that is applicable only to the current block. This signal can be considered valid as long as bdi_read signal has not been received from CipherCore.

Appendix C: PostProcessor Ports

Table C5: PostProcessor Ports

Port	Direction	Width	Definition
clk	in	1	Global clock signal
rst	in	1	Global reset signal (synchronous)
do	out	W	Output data out
do_ready	in	1	Output ready
do_valid	out	1	Output write
bypass_data	in	W	Bypass FIFO data
bypass_empty	in	1	Bypass FIFO empty
bypass_rd	out	1	Bypass FIFO read
bdo_ready	out	1	Signal indicating that a new set of data block is ready to be received
bdo_write	in	1	Input data write
bdo_data	in	BLOCK_SIZE	Input data from crypto core
tag_ready	out	1	Signal indicating a new tag data is ready to be received
tag_write	in	1	Tag data write
tag_data	in	TAG_SIZE	Input tag from from crypto core
msg_auth_done	in	1	Message authentication completion signal
msg_auth_valid	in	1	Message authentication valid signal
bypass_fifo_data	in	W	Bypass FIFO data
bypass_fifo_empty	in	1	Bypass FIFO empty signal
bypass_fifo_rd	out	1	Bypass FIFO read signal
aux_fifo_din	out	W	Auxiliary FIFO input
aux_fifo_ctrl	out	4	Auxiliary FIFO control signals
aux_fifo_dout	in	W	Auxiliary FIFO output
aux_fifo_status	in	3	Auxiliary FIFO status signals