A Universal Hardware API for Authenticated Ciphers

Ekawat Homsirikamol, William Diehl, Ahmed Ferozpuri,
Farnoud Farahmand, Malik Umar Sharif, and Kris Gaj
Volgenau School of Engineering
George Mason University
Fairfax, Virginia 22030
Email: {ehomsiri, wdiehl, aferozpu, ffarahma, msharif2, kgaj}@gmu.edu

***Abstract*—In this paper, we propose a universal hardware Application Programming Interface (API) for authenticated ciphers. In particular, our API is intended to meet the requirements of all algorithms submitted to the CAESAR competition. Two major parts of the API, the interface and the communication protocol, were developed with the goal of reducing any potential biases in benchmarking of authenticated ciphers in hardware. Our high-speed implementation of the proposed hardware API includes universal, open-source pre-processing and post-processing units, common for all CAESAR candidates and the current standards, such as AES-GCM and AES-CCM. Apart from the full documentation, examples, and the source code of the pre-processing and post-processing units, we have made available in public domain a) a universal testbench to verify the functionality of any CAESAR candidate implemented using our 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, which may be used as building blocks in implementations of related ciphers. 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.**

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

Therefore, there is a clear need for a proposal regarding a uniform hardware API, which could be further modified and improved using feedback from the cryptographic community, and eventually endorsed by the CAESAR Committee, and adopted by majority of future hardware developers. Our goal 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.

II. 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

This work is supported by NSF Grant #1314540.

978-1-4673-9406-2/15/$31.00

- 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].

III. 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 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 these APIs.

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.

IV. Specification

A. 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.

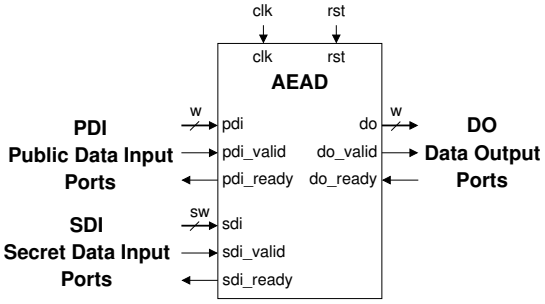

Fig. 1: AEAD Interface

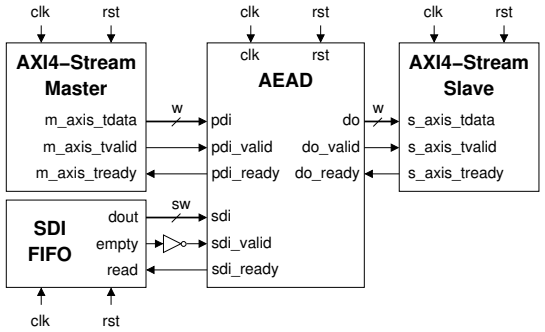

Fig. 2: Typical external circuits: AXI4 IPs

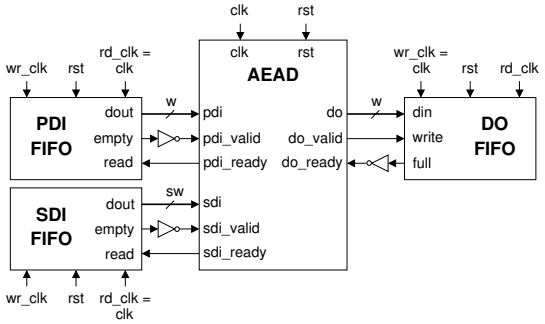

Fig. 3: Typical external circuits: FIFOs

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) 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).

B. 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.

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 (LDKEY) and Load Round Key (LDRKEY). The instruction is followed by a single segment. A segment starts with a separate header, describing its type and size. In case of `SDI`, the only allowed segment types are: `Key` and `Round Key`. Note that instruction Load Key and its accompanying data segment (`Round Key`) should only be used when key scheduling is done in software.

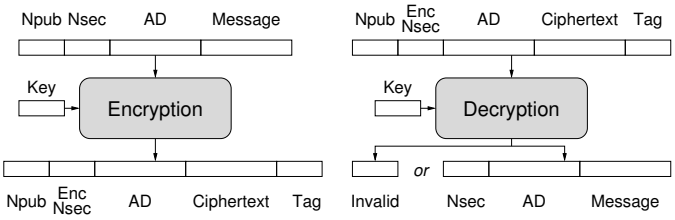

Fig. 4: Input and Output of an Authenticated Cipher. Notation: Npub - Public Message Number, Nsec - Secret Message Number, AD - Associated Data

(a) Key loading (b) Round key loading

Fig. 5: Format of Secret Data Input for a) Loading main key, b) Loading a sequence of round keys

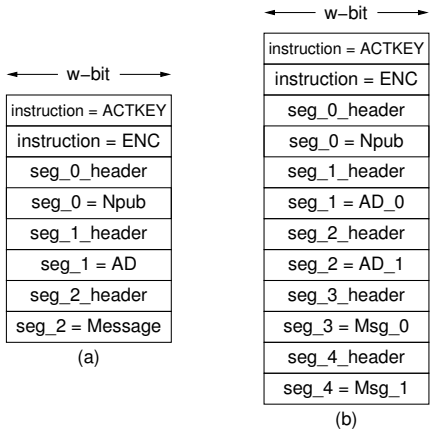

Fig. 6: Format of Public Data Input in case of a) one segment for each data type, b) multiple segments for AD and Message

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), Secret Message Number (Npub), Associated Data (AD), and Message. Segment types allowed during authenticated decryption include: Public Message Number (Npub), Encrypted Secret Message Number (EncNpub), Associated Data (AD), Ciphertext, and Tag. Any segment type can be omitted, if it is not required by a given cipher. Public, Secret and Encrypted Secret Message Number can only use one segment, as their size are 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 our supporting codes).

The instruction format is shown in Fig. 7. The Opcode field determines which operation should be executed next. The same field also serves as a Status field to indicate whether the decryption operation is successful or not. 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).

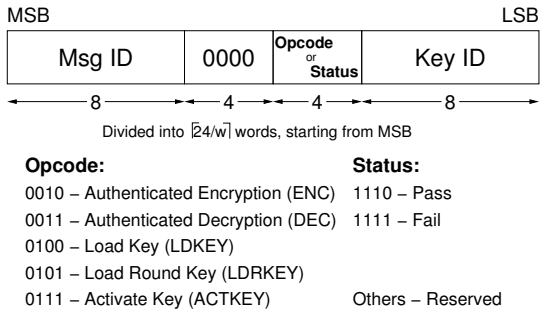

Fig. 7: Instruction Format

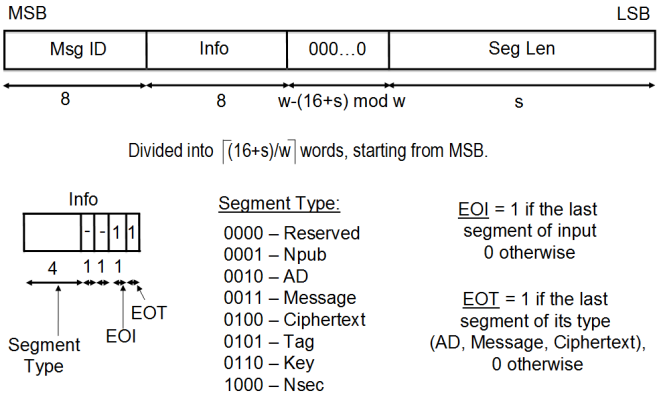

Fig. 8: Segment Header Format

V. Supporting Codes for High-Speed Implementations

A. 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 our HW API distribution [17].

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.

B. 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.

The PreProcessor and PostProcessor cores are highly configurable using generics. 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 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:

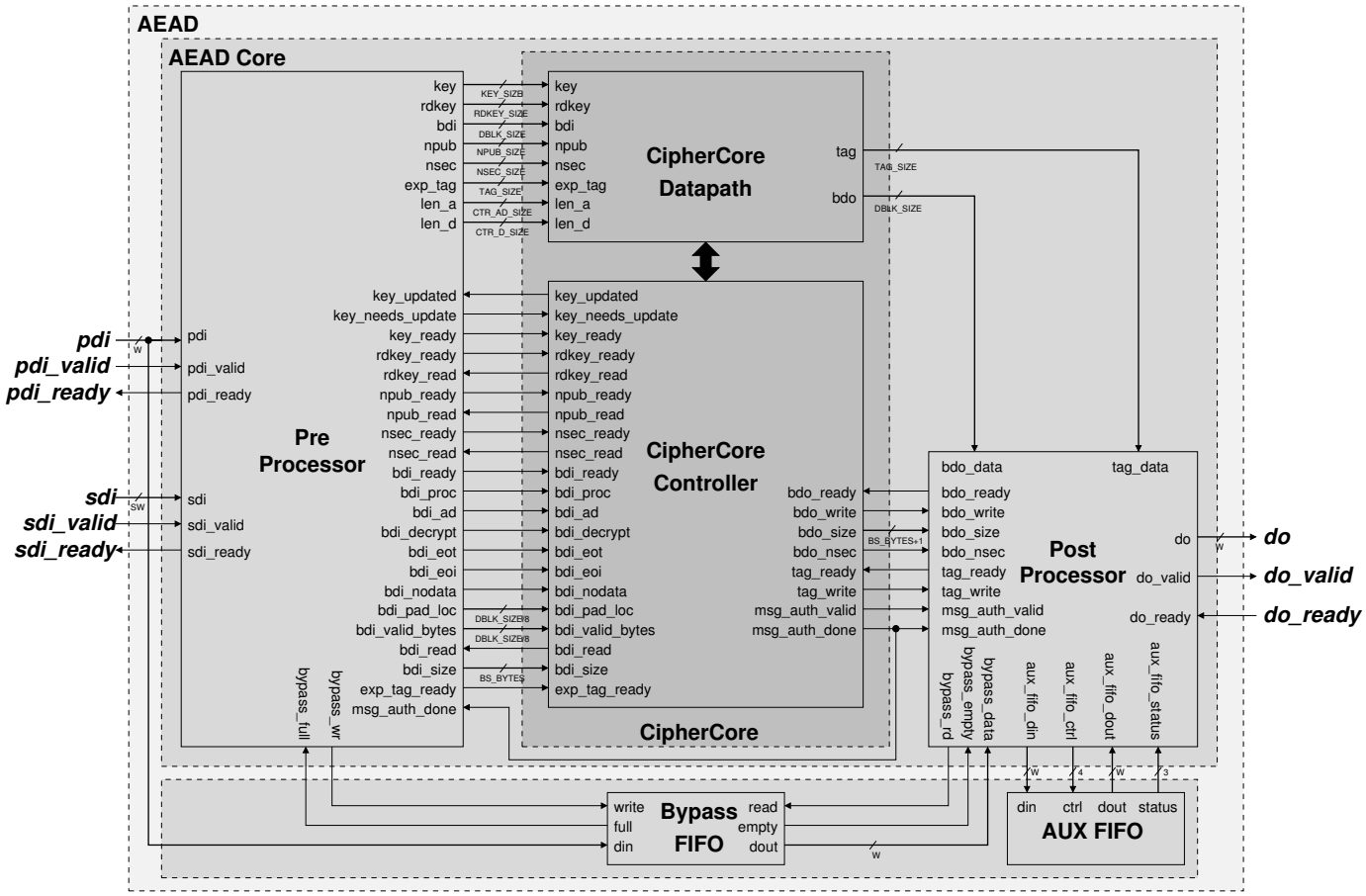

Fig. 9: High-level block diagram of a high-speed implementation

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

This word must be applied before any other instruction word.

C. 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 [17]. 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.

D. 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 preferences (see Section VI).

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.

VI. The Development of CipherCore

It is recommended to start the development of the CipherCore, specific to a given authenticated cipher, by using the provided AEAD_Core and CipherCore template files as a starting point [17]. This is because the appropriate connections among the CipherCore, the PreProcessor and the PostProcessor modules are already specified in these files. A designer needs first to modify the generics at the top of the AEAD_Core 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 algo-

rithmic state machine (ASM) chart or a state diagram, further translated to HDL.

The algorithmic state machine (ASM) of the CipherCore Controller is typically characterized by the following groups of states:

1) Load and/or activate the key
2) Process associated data
3) Process message/ciphertext
4) Generate/verify an authentication tag

In the first group of states, *Load and activate the key*, the CipherCore 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 and a new key can be loaded into the PreProcessor 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. Please note that the above two events can occur in an arbitrary order.

After `key_ready` and `key_needs_update` are both high, and the CipherCore is either in the period between reset and the first input, or in the period between two consecutive inputs, the CipherCore 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 CipherCore should monitor the `key_needs_update` port prior to processing any new input. If `key_needs_update` is high, the CipherCore 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 CipherCore should move directly to processing a new input using a previous key. If none of these two events is detected, the CipherCore should remain in the same state. The described behavior is shown in Fig. 10. The key initialization and process data are two separate states that operate depending on the requirements of a specific cipher.

In the second group of states, *Process associated data*, the core continuously waits for the next AD block until the `bdi_eot` signal becomes active. This signal indicates that the current block is the last block of associated data. The state machine needs then to process this last block, and proceed to the next group of states, responsible for encryption and decryption of data. If the first block read by the CipherCore is not of type AD (`bdi_ad`=0), then associated data is assumed to be empty. If the last block of AD (`bdi_ad`=1 and `bdi_eot`=1) is also the last block of input (`bdi_eoi`=1), then the message/ciphertext is assumed to be empty.

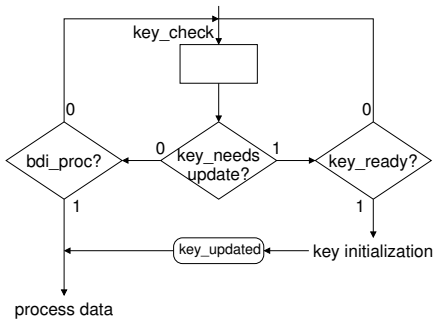

Fig. 10: A part of the Algorithmic State Machine (ASM) chart describing a way in which the CipherCore Controller may handle key loading and key activation

The third group of states, *Process message/ciphertext*, should operate in the similar way as the second group, and should similarly progress to the next group of states when the last block of ciphertext or message is processed. In this group of states, `bdi_ad` should remain *inactive* for each input block to indicate that the current block is *not* an associated data block. A corresponding output data block should be passed to the PostProcessor using the `bdo` port with an accompanied active `bdo_write` control signal.

After each block of associated data, message, or ciphertext is read by CipherCore , the `bdi_read` output must be activated for one full clock cycle. This action clears control inputs, such as `bdi_eot` and `bdi_eoi` that may need to be checked at a later time. At the same time, this action cannot be delayed because doing so would stall the PreProcessor and prevent it from loading any subsequent data block using the PDI input. As a result, `bdi_eot` and `bdi_eoi` must be registered at the latest in the clock cycle when the acknowledgment signal `bdi_read` is generated. Only registered values of these inputs should be checked at a later time.

In the last group of states, *Generate/verify an authentication tag*, during the authenticated encryption, the core should generate a new tag and pass it to the PostProcessor, using ports `tag` and `tag_write`. During the authenticated decryption, `msg_auth_done` should be activated, and the `msg_auth_valid` port should be used to output the result of authentication.

It should be noted that not all signals at the interfaces PreProcessor-CipherCore and PostProcessor-CipherCore need to be used for each particular cipher. If any port is left unconnected, the corresponding port and the associated logic are automatically trimmed off (removed) by the synthesis tool. Thus, the full set of internal signals shown in Fig. 9 and included in the template files available at [17] should be treated as a superset of signals required by all authenticated ciphers, supported by our hardware API and the associated high-speed PreProcessor and PostProcessor modules.

The full description of all generics and ports used by our supporting VHDL codes can be found in the full documentation available at [17].

VII. Universal Testbench and Test Vector Generation

Our supporting codes include:

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

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 [17].

VIII. 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

1) CipherCore has an incomplete functionality and a full-block-width interface, which is not realistic in a typical application where FIFOs are located on the boundary between two subsystems,
2) Using AEAD may cause difficulty with setting BRAM usage to 0 (as often desired in order to easily calculate throughput to area ratio),

In case of AEAD Core, for Virtex 7 and Zynq, we recommend generating results using Xilinx Vivado [18], operating in the Out-of-Context (OOC) mode [19]. 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 [17].

In case of CipherCore, because of a large number of required ports 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. For Virtex 6 and below, we recommend using Xilinx ISE and ATHENa [20]. For Altera FPGAs, we suggest using Altera Quartus II and ATHENa.

Our database of results for authenticated ciphers is available at [21]. After receiving an account in the database, the designers can enter results by themselves.

A. Overheads

So far, eight CAESAR Round 1 candidates (all qualified to Round 2) and the current standard AES-GCM have been implemented using our hardware API. The detailed results, for Xilinx Virtex 6, Virtex 7, and Zynq 7000 families, are available in [21].

The first preliminary results regarding an overhead introduced by extending CipherCore to AEAD Core are summarized in Figs. 11, 12, 13, and 14.

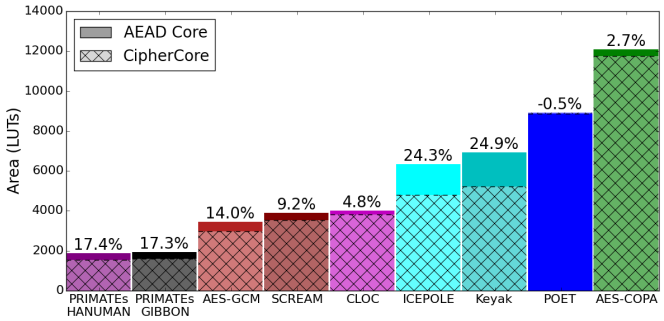

Fig. 11: AEAD Core vs. CipherCore Area Overhead for Virtex 6 FPGA family

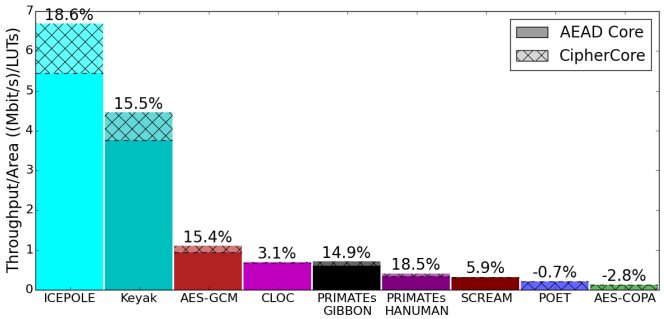

Fig. 12: AEAD Core vs. CipherCore Throughput/Area Overhead for Virtex 6 FPGA family

For Virtex 6, the highest area overheads are incurred for ICEPOLE and Keyak (both in the range of 25%). These large overheads are caused primarily by large cipher block sizes (1024 bits for ICEPOLE and 1344 bits for Keyak), as well as large input word sizes (w=256 and w=128, respectively). For all remaining algorithms, 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. For one algorithm, POET, the area overhead becomes even negative, which can be explained only by the boundary optimizations performed by Xilinx FPGA tools. In terms of the Throughput/Area ratio, the overheads are the highest for ICEPOLE, PRIMATES-HANUMAN, Keyak, AES-GCM, and PRIMATES-GIBBON, all in the range 15-19%. For the remaining algorithms, the overhead does not exceed 6%.

For Virtex 7, the area overheads are the highest for Keyak (due to the large block and word sizes), as well as PRIMATES-GIBBON and PRIMATES-HANUMAN (due to low overall area of these cores), all between 18% and 27%. For all remaining algorithms, the area overhead does not exceed 15%, and becomes even negative for AES-COPA. In terms of the Throughput/Area ratio, the overhead is exceptionally high for Keyak (35.3%). For all remaining algorithms, it does not exceed 30%.

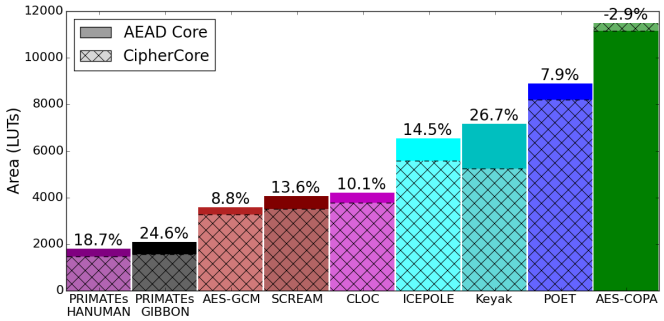

Fig. 13: AEAD Core vs. CipherCore Area Overhead for Virtex 7 FPGA family

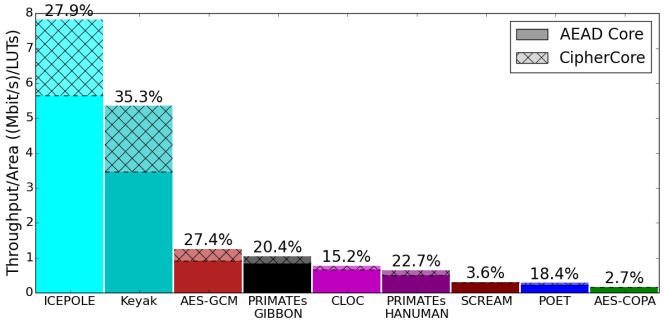

Fig. 14: AEAD Core vs. CipherCore Throughput/Area Overhead for Virtex 7 FPGA family

IX. Unsupported Features and Future Work

The features of our Hardware API that are not yet fully supported by our codes available at [17] include:

- use of Message ID
- 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
- support for multiple streams of data.

X. 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 our 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 our database of results.

Our 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. 32-bit Processor Local Bus: Architecture specifications. [Online]. Available: http://embedded.eecs.berkeley.edu/mescal/forum/7/coreconnect_32bit.pdf

[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] Cryptographic Engineering Research Group (CERG) at GMU. (2015, Jul.) GMU Hardware API. [Online]. Available: https://cryptography.gmu.edu/athena/index.php?id=download

[18] Xilinx. Vivado Design Suite. [Online]. Available: http://www.xilinx.com/products/design-tools/vivado.html

[19] ——, *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

[20] 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.

[21] Cryptographic Engineering Research Group (CERG) at GMU. (2015, Jul.) GMU ATHENa Database of Results. [Online]. Available: https://cryptography.gmu.edu/athenadb/fpga_auth_cipher/rankings_view