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 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.

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, 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.

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 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.

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,

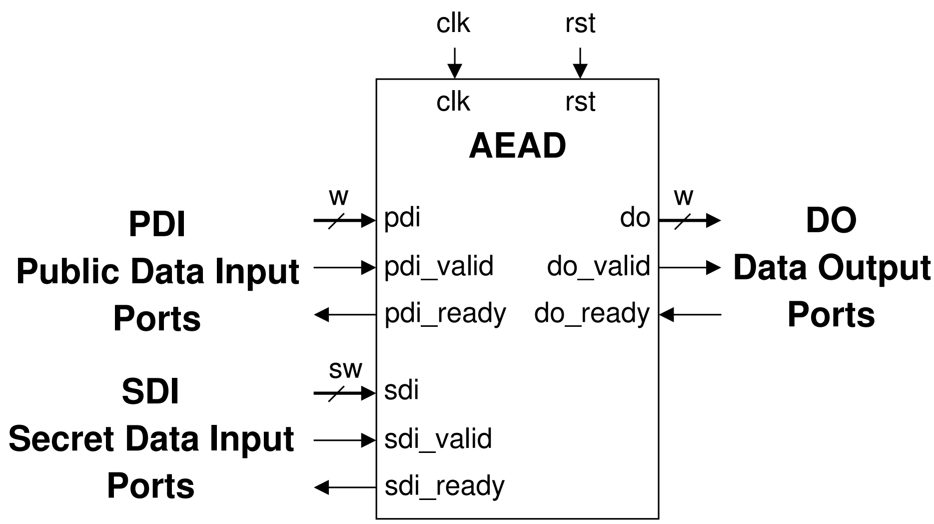


Fig. 1: AEAD Interface

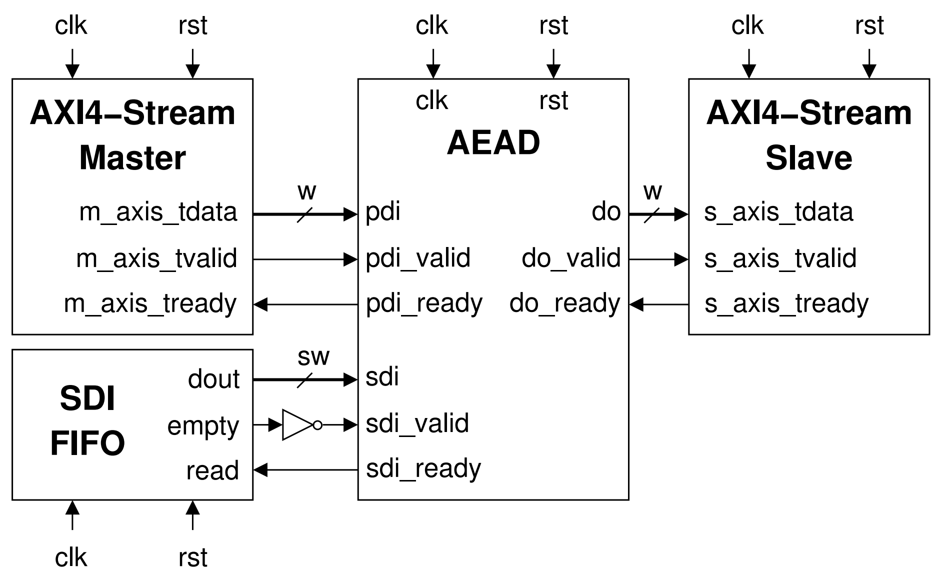


Fig. 2: Typical external circuits: AXI4 IPs

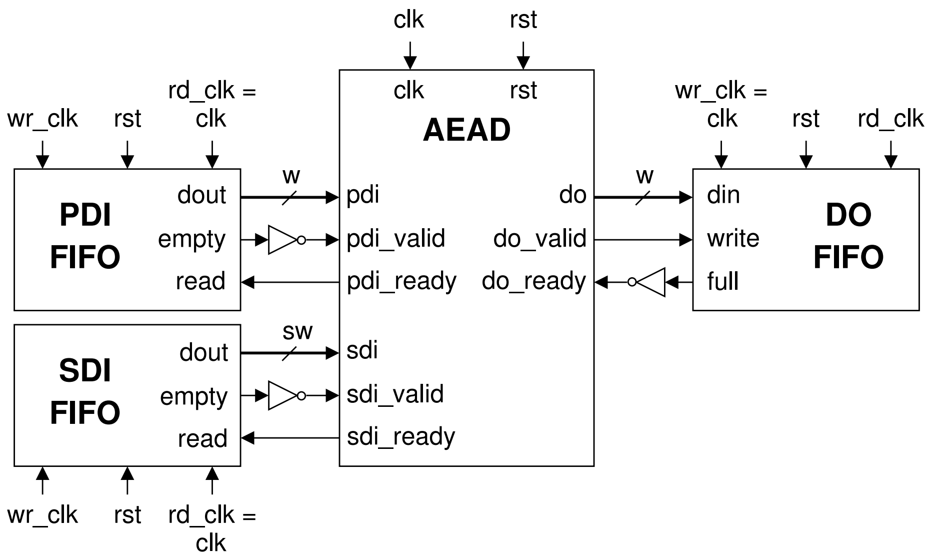


Fig. 3: Typical external circuits: FIFOs

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.

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).

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. The difference is that Npub is sent to the other side in clear, while Nsec is sent in the encrypted form.

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 (LDRDKEY). The instruction is followed by 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 Round Key, carrying either the main key or a sequence of round keys, precomputed in software, respectively. Round keys are assumed to be arranged in the natural order, starting from the round key with the smallest index.

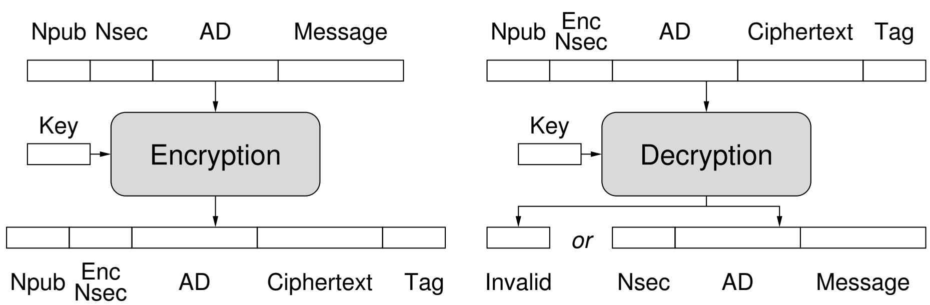


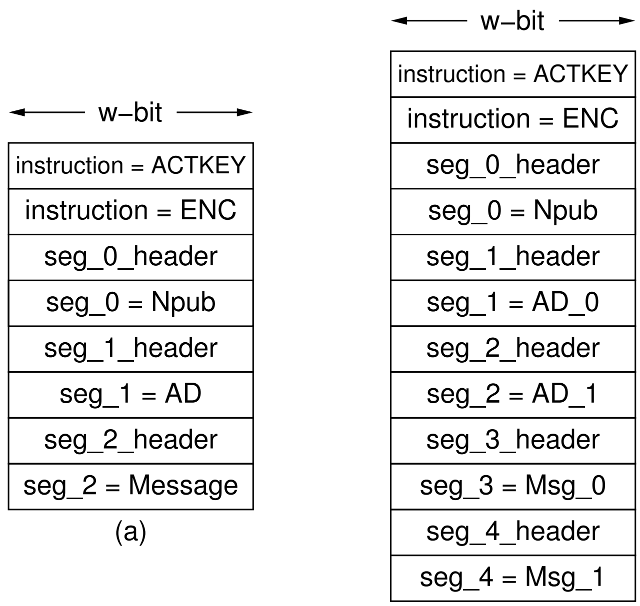
Fig. 4: Input and Output of an Authenticated Cipher. Notation: Npub - Public Message Number, Nsec - Secret Message Number, Enc Nsec - Encrypted Secret Message Number, AD - Associated Data

The proposed format of the Public Data Input is shown in Fig. 6. The allowed instruction types are: Activate Key (ACTKEY), Authenticated Encryption (ENC), and Authenticated Decryption (DEC). 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 (Nsec), Associated Data (AD), and Message. Segment types allowed during authenticated decryption include: Public Message Number (Npub), Encrypted Secret Message Number (Enc Nsec), Associated Data (AD), Ciphertext, and Tag. Any segment type can be omitted, if it is not required by a given cipher. Public and Secret Message Numbers can only use one segment,

(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



(a)

(b)

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

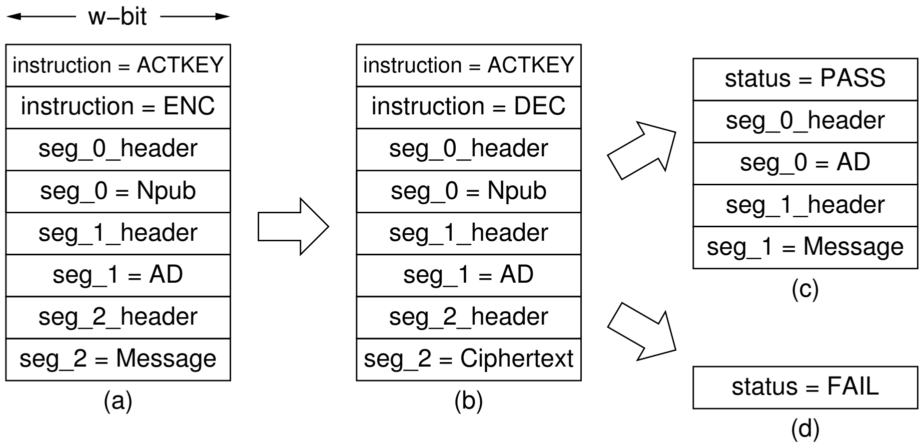


Fig. 7: Format of Public Data Input (PDI) and Data Output (DO) for ciphers that do not use Nsec: a) PDI for encryption, b) DO for encryption = PDI for decryption, c) DO for decryption in case of successful authentication, d) DO for decryption in case of authentication failure

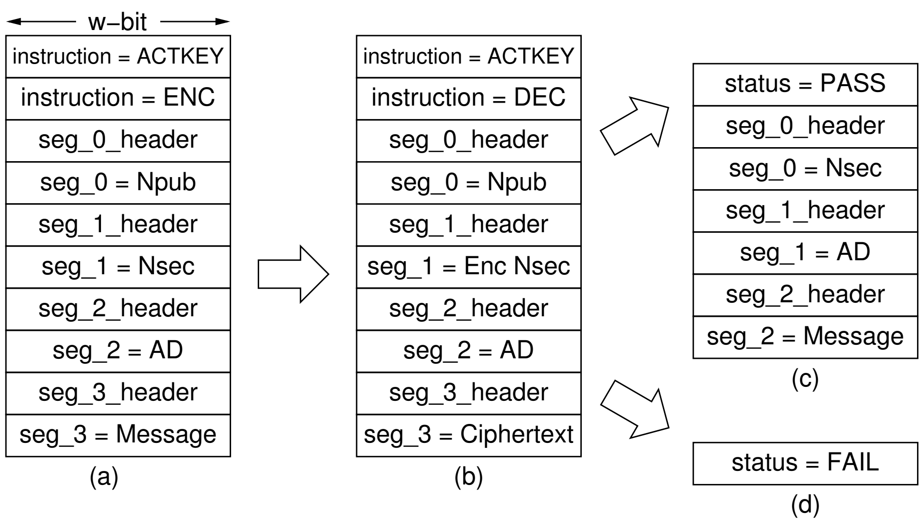


Fig. 8: Format of Public Data Input (PDI) and Data Output (DO) for ciphers that use Nsec: a) PDI for encryption, b) DO for encryption = PDI for decryption, c) DO for decryption in case of successful authentication, d) DO for decryption in case of authentication failure

as their sizes 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. 6).

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/status format is shown in Fig. 9. 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. For instruction, the Opcode field determines which operation should be executed next. For status, the Opcode field is replaced by the Status field, which can be set to only two values, PASS or FAIL.

The segment header format is shown in Fig. 10. 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).

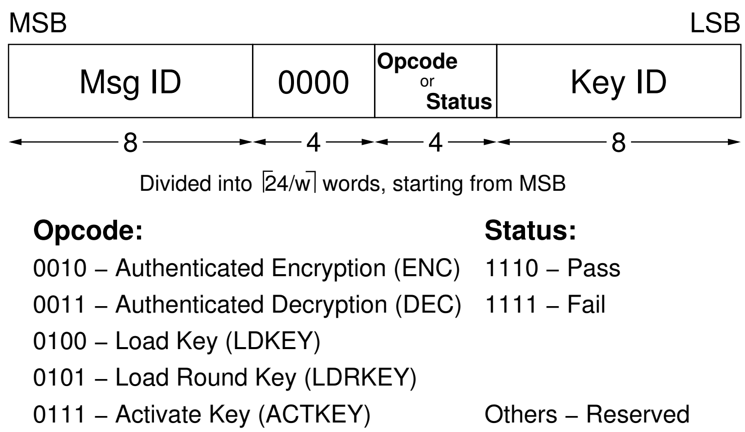


Fig. 9: Instruction/Status Format

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. 11. AEAD consists of AEAD Core and the

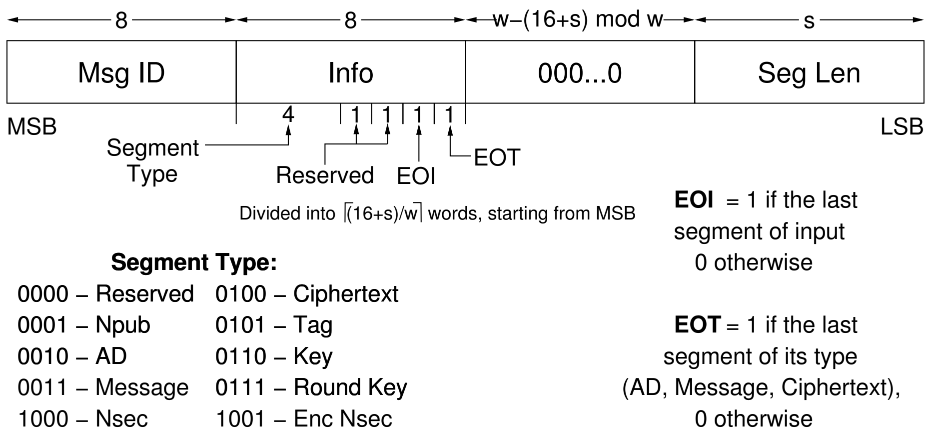


Fig. 10: Segment Header Format

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.

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

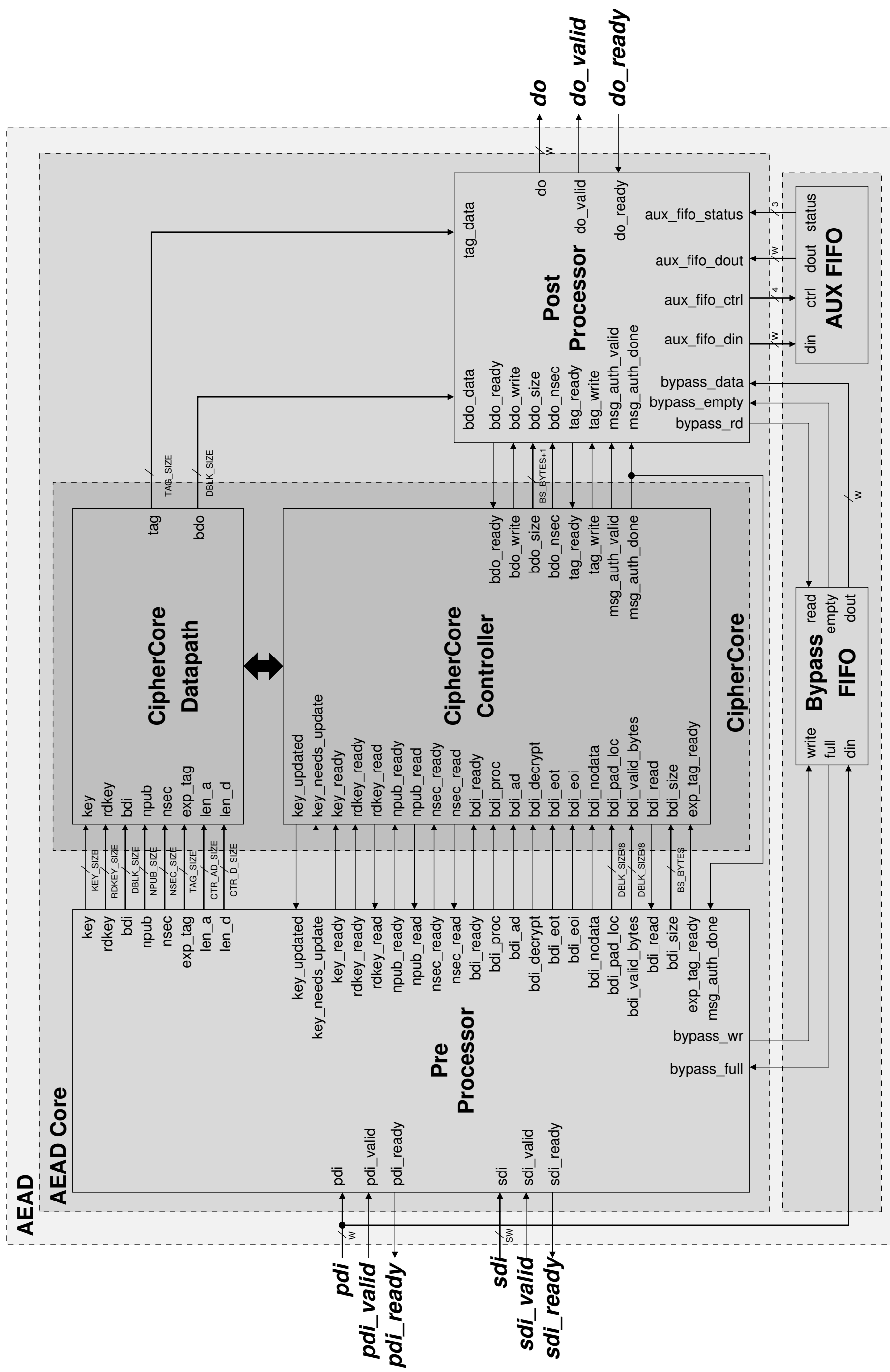


Fig. 11: High-level block diagram of a high-speed implementation

- storing decrypted messages in AUX FIFO, until the result of authentication is known, and
- generating the status block with the result of authentication.

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:

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

This word must be applied before any other instruction word.

Loading of round keys precomputed in software can be performed in a similar way, with the instruction Load Key replaced by Load Round Key, followed by a segment composed of a sequence of round keys.

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 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.

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 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

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 algorithmic 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. 12. 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.

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 message or ciphertext 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

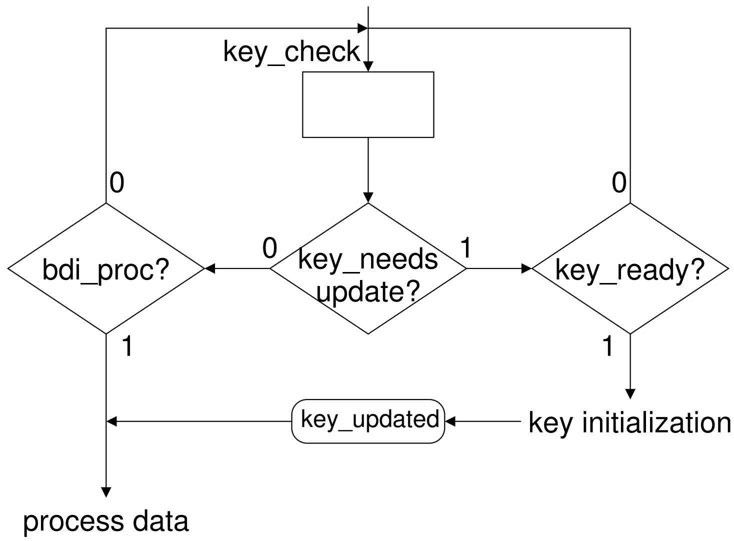


Fig. 12: A part of the Algorithmic State Machine (ASM) chart describing a way in which the CipherCore Controller may handle key loading and key activation

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. 11 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 Appendices A, B, and C.

7 Universal Testbench and Test Vector Generation

Our supporting codes for verification 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
- 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] (see Appendix D).

8 Generation and Publication of Results

Generation of results is possible for AEAD, AEAD Core, and CipherCore (see Fig. 11). 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,
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 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 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 [17], and their use is explained in detail in Appendix E. 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.

8.1 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.

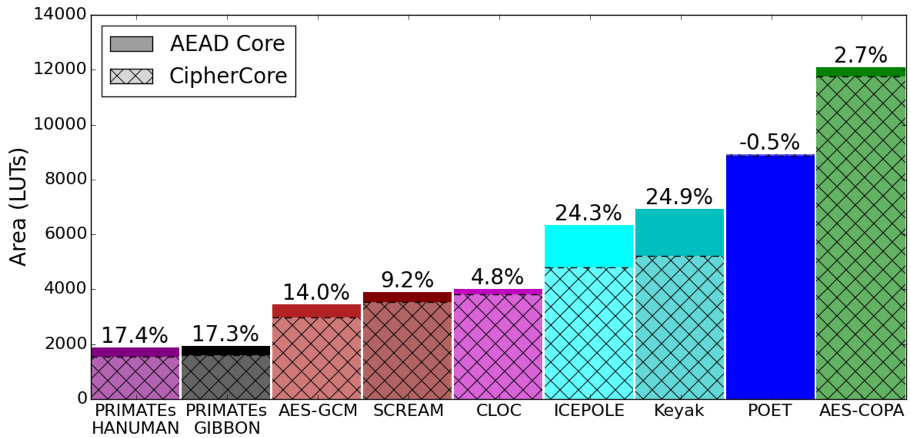


Fig. 13: AEAD Core vs. CipherCore Area Overhead for Virtex 6 FPGA family

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. 13, 14, 15, and 16.

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 28%. 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%.

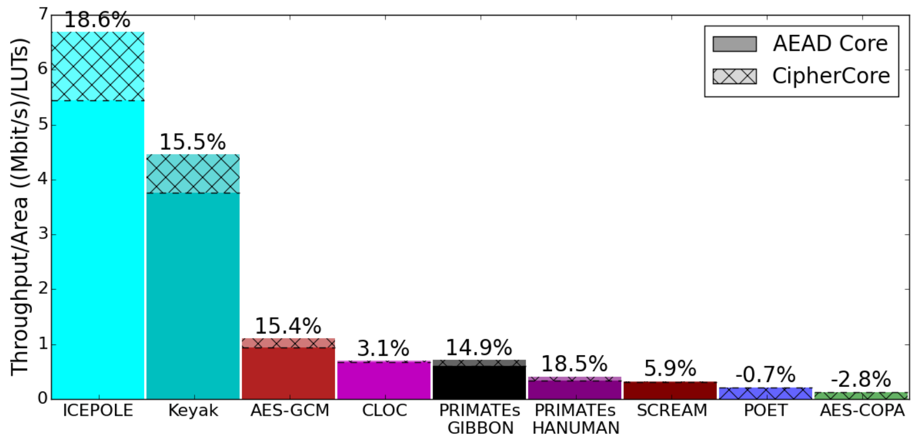


Fig. 14: AEAD Core vs. CipherCore Throughput/Area Overhead for Virtex 6 FPGA family

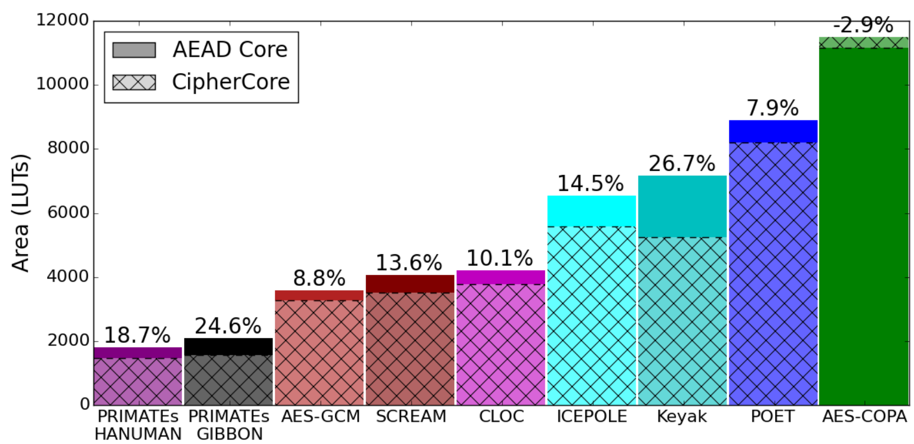


Fig. 15: AEAD Core vs. CipherCore Area Overhead for Virtex 7 FPGA family

9 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

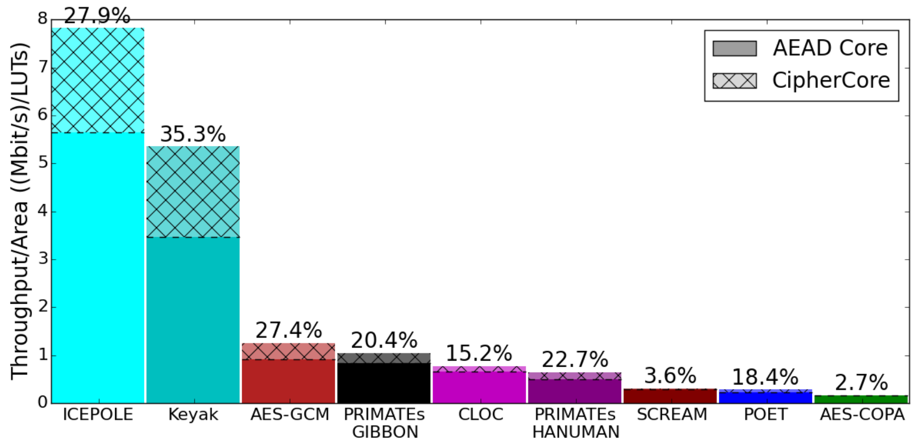


Fig. 16: AEAD Core vs. CipherCore Throughput/Area Overhead for Virtex 7 FPGA family

- support for two-pass algorithms
- accepting inputs with padding 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 our Hardware API is facilitated by:

- Detailed specification
- Universal testbench and Automated Test Vector Generation
- PreProcessor and PostProcessor Units for high-speed implementations
- Scripts and 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, Sep.) 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 Inter-*

national Conference on Field Programmable Logic and Applications - FPL 2010. IEEE, 2010, pp. 414–421.
21. Cryptographic Engineering Research Group (CERG) at GMU. (2015, Sep.) GMU ATHENa Database of Results. [Online]. Available: https://cryptography.gmu.edu/athenadb/fpga_auth_cipher/rankings_view

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	128	Npub size (bits)
x		NSEC_ENABLE	0	Enables *nsec* port
x		NSEC_SIZE	8	Nsec 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		RDKEY_ENABLE	0	Enables *rdkey* port and disables *key* port
x		RDKEY_SIZE	128	Round 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) Enable padding for associated data (AD) block. See Table A2 for more details.
x	x	PAD_D	1	(Active when PAD=1) Enable padding for data block. See Table A2 for more details.
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 A3 for more details.
x	x	CIPHERTEXT_MODE	0	Ciphertext output handling mode. See Table A4 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). Note: bdi_size is provided in a reversed order for decryption. This means that the remainder is provided in the first block instead of the last block.

Not all combination of generics are supported. In particular,

- REVERSE_DBLK = 1 is only supported when CIPHERTEXT_MODE = 2.
- Values of ABLK_SIZE and DBLK_SIZE have to meet the following conditions:
  - *ABLK_SIZE mod W = 0* and *DBLK_SIZE mod W = 0*, or
  - *ABLK_SIZE mod W = W/2* and *DBLK_SIZE mod W = W/2*

Table A2: Extended description of PAD_AD and PAD_D.

Generic Value	0	1	2	3	4
Enable padding		x	x	x	x
Add extra block when AD/D is Empty			x		x
Add extra block when AD/D is multiple of a block size				x	x

Table A3: Extended description of PLAINTEXT_MODE

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.

Note: (*) default option. (**) Npub related signals are disabled.

Operations specific to each CIPHERTEXT_MODE value are further described below:

(0) C_T

during encryption
- len_d= $|M|$
- The tag output of the CipherCore Datapath is used
- The PostProcessor waits for 1 at the tag_write output of the CipherCore 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 CipherCore Datapath is used.
- The exp_tag_ready input of the CipherCore Controller is used.

Table A4: Extended description of CIPHERTEXT_MODE.

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.

Note: (*) default option.

(1) CT

during encryption

- len_d = $|M|$
- The tag output of the Datapath is not used
- The PostProcessor does not wait for 1 at the tag_write output of the CipherCore 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 CipherCore Datapath is not used.
- The exp_tag_ready input of the CipherCore Controller is not used.

(2) Cexp_T

during encryption

- len_d = $|M|$
- The tag output of the CipherCore Datapath is used
- The PostProcessor waits for 1 at the tag_write output of the CipherCore Datapath
- The size of C in the ciphertext segment header = $|M|$ but the PostProcessor 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 PreProcessor reads and passes to the CipherCore Datapath $block\_size * \lceil |M|/block\_size \rceil$ bits of the ciphertext
- len_d = $|M|$
- The exp_tag input of the CipherCore Datapath is used.
- The exp_tag_ready input of the CipherCore Controller is used.

Appendix B: PreProcessor Ports

Table B5: 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
npub	out	NPUB_SIZE	*[Optional]* Public message number (Npub). This port is inactive if PLAINTEXT_MODE = 1 or 2.
nsec	out	NSEC_SIZE	*[Optional]* Secret message number (Nsec). This port is inactive if NSEC_ENABLE = 0.
key	out	KEY_SIZE	*[Optional]* Key data. Note: Port is disabled if RDKEY_ENABLE = 1.
rdkey	out	RDKEY_SIZE	*[Optional]* Round key data. Note: Port is disabled if RDKEY_ENABLE = 0.
bdi	out	DBLK_SIZE	Input block data
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 associated 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
rdkey_ready	out	1	*[Optional]* Round key ready signal. This port is ignored if RDKEY_ENABLE = 0.
rdkey_read	in	1	*[Optional]* Round key read signal. This port is ignored if RDKEY_ENABLE = 0.
npub_ready	out	1	*[Optional]* Npub ready signal. This port is inactive if PLAINTEXT_MODE = 1 or 2.

npub_read	in	1	*[Optional]* Npub read signal. This port is inactive if PLAINTEXT_MODE = 1 or 2. Note: npub_read signal must be issued for the current message before the next npub of the next message can be loaded within the PreProcessor.
nsec_ready	out	1	*[Optional]* Nsec ready signal. This port is inactive if NSEC_ENABLE = 0.
nsec_read	in	1	*[Optional]* Nsec read signal. This port is ignored if NSEC_ENABLE = 0.
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 associated data
bdi_decrypt	out	1	*[INPUT INFO]* Current input should be decrypted.
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 Npub, the subsequent block is generally either of type message or associated 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 C6: 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
bdo_size	in	BS_BYTES+1	*[Optional]* Data size of the output block (required when CIPHERTEXT_MODE = 2)
bdo_nsec	in	1	Input data Nsec flag. This signal indicates that the incoming block is an Nsec block.
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

Appendix D: 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
- Modified C codes of the Round 2 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.

Appendix D.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+**

Appendix D.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.

Appendix D.3 CAESAR library prerequisites

OpenSSL library is required to completely compile all the provided CAESAR source codes. In the case that OpenSSL is not already installed on the system, please download the latest OpenSSL code from https://www.openssl.org/source/ and do the following steps:

```
tar zxvf openssl-1.0.2d.tar.gz
cd openssl-1.0.2d
# For MingW user
./Configure mingw --prefix=/usr/local shared
# For Linux user
./Configure --prefix=/usr/local shared
make
make install
```

Appendix D.4 Quick User Guide

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

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) type

   ```
   make
   ```

2. Generate the script using a pre-defined settings. Examples of the pre-defined settings can be found in the `$root/software/AETVgen/gen.py` file. User needs to do:
   (a) Copy one of the example methods and modify the primary argument (*args*). Example methods' parameters are described in Appendix D.4.
   (b) Call the new function from the main method by issuing:

   ```
   gen.py
   ```

3. Copy the three generated test vectors (pdi.txt, sdi.txt and do.txt) in *AETVgen* folder to your test vector if you haven't already set any copy target.

Table D7 provides a list of possible options for the (*args*) argument for *AETVgen* class.

Table D7: AETVgen class parameters

Option	Description	Default value	Valid values
caesarLib	CAESAR library's name	aes128gcmv1	CAESAR's library name
Algorithm's parameter			
testMode	Test mode	0	[0,1,2]
sizeKey	Key size	16	Any integer
enableRoundKey	Enable Round Key (disable Key)	FALSE	True/False
sizeRoundKey	Round key size. Ignore if enableRoundKey == False.	16	Any integer
totalRoundKey	Number of round key. Ignore if enableRoundKey == False.	11	Any integer
sizeNpub	Npub size	12	Any integer
enableNsec	Enable Nsec segment	FALSE	True/False
sizeNsec	Nsec size. Ignore if enableNsec == False.	16	Any integer
sizeTag	Tag size	16	Any integer
blockSize	Algorithm's block size	16	Any integer
blockSizeAD	Algorithm's AD block size	16	Any integer
I/O format			
plainTextMode	Plain text mode	0	[0,1,2,3]
cipherTextMode	Cipher text mode	0	[0,1,2]
reverseDblk	Reverse data block. This option should only be used a reversed order of ciphertext is required, i.e. last block first. This mode was created specifically for PRIMATEs-APE.	FALSE	True/False
padD	This mode should only be set to 4 when cipherTextMode == 2 as encrypted data is expanded by default.	0	[0,4]
maxSizeSegment	Max segment size	100000	Any integer divisible by PIO and blockSize
Input Message Parameters			
minSizeAD	Minimum authenticated data size	0	Any integer
maxSizeAD	Maximum authenticated data size	512	Any integer
minSize	Minimum data size	0	Any integer
maxSize	Maximum data size	512	Any integer
Hardware I/O width			
sizePIO	Size of pdi port	4	Any integer >4
sizeSIO	Size of sdi port	4	Any integer >4
Debugging options			
verbose	Print everythign within the #if DBG âĂę #endif clause in the C code	FALSE	True/False
startTV	Starting test vector. This option should be used when testMode >0 for debugging purposes.	0	Any integer
decrypt	Perform decryption. By default, input and output is generated using only encryption operation. This option allows directs the script to also perform decryption for encrypted data for debugging purpose. However, please note that we do not use this option for generation of our test vectors.	FALSE	True/False
Output file name			
filePDI	Public Data In file	'pdi.txt'	Any text
fileSDI	Public Data Out file	'sdi.txt'	Any text
fileDO	Data Out file	'do.txt'	Any text

Pre-defined Methods have the following format:

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

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 $False$.
- $\$StartTV$ provides the starting point for test vector generation. Valid when $\$TestMode$ > 0.

Appendix D.5 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.

Appendix D.6 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 There's no specific Python related modification. User needs to provide an appropriate settings to the *AETVgen* for the output to be produced correctly.

Appendix E: Vivado Results Generation Scripts

Starting from version 1.1b1, our supporting codes, available at [17], include a set of scripts that can be used to generate optimized results using Xilinx Vivado. A user of these scripts can choose to implement HDL code of a cryptographic module

- without a wrapper, using the Out-of-Context (OOC) mode of Vivado (OOC mode) [19], or
- with a simple wrapper (aimed at reducing a total number of pins required), using the TopDown mode of Vivado [19].

To generate results for a specific project you must set up the directory structure, list all source files to be included in the project, modify several key files, and finally run a few scripts to generate device specific results. This process is summarized in the step-by-step fashion below. Additionally, Xilinx provides a general tutorial [19], describing the aforementioned design modes in more detail. The instructions below apply to both the OOC mode and the TopDown mode of the results generation, unless otherwise noted.

1. Directory Structure Setup
   (a) Copy the `scripts/VivadoBatch` folder to a new workspace.
   (b) Rename the PROJECT folder to any more specific project name. Note, you can copy and paste this folder as many times as required to accommodate multiple projects.
   (c) Copy all source files to the subfolder "`Sources/hdl`"
2. PRJ File Setup
   For the AEAD_Core implemented in the OOC mode modify `prj/AEAD_Core.prj`.
   For the CipherCore implemented in the TopDown mode, with a wrapper, modify `prj/CipherCore_Wrapper.prj`, respectively.
   In the respective PRJ file, list names of all source files necessary to implement your circuit (including a possible wrapper). Use the format: vhdl work "`[SRC FOLDER]/[FILENAME]`", e.g., vhdl work "`hdl/AEAD_pkg.vhd`"
3. (OOC Only): Blackbox File
   (a) Create `hdl/AEAD_Core_bb.vhd` with only the entity declaration from `AEAD_Core.vhd`.
4. (Optional): Constraints File
   (a) The target clock frequency or placement constraints for the implementation can be modified in either `AEAD_Core_Wrapper_flpn.xdc` (OOC) or `CipherCore_Wrapper_flpn.xdc` (TopDown). Please note, that for any TopDown implementation, only timing constraints are required, while both timing and placement constraints are used when generating OOC results. For more details, please see "Step 5: Defining the Top-Level Constraints" [19] .
5. Wrapper Files
   (a) Set the appropriate generic values for the wrapper files: `hdl/AEAD_Core_Wrapper.vhd` (OOC) and/or `hdl/CipherCore_Wrapper.vhd` (TopDown)
   (b) Ensure that `hdl/AEAD_Core_Wrapper.vhd` has the correct values of G_W and G_SW, set in accordance with `AEAD_Core.vhd`
6. Script Execution and Result Generation
   (a) Type any of the following four command sequences into Vivado Tcl Shell to generate results in the OOC or TopDown mode, targeting Virtex-7 (v7) or Zynq. Note, you will need to launch a new Vivado Tcl Shell each time if you want to run all four command sequences simultaneously.
   OOC Zynq:

```
vivado -mode batch -source genOOC_zynq.tcl -notrace
vivado -mode batch -source runOOC_zynq.tcl -notrace
```

OOC v7:

```
vivado -mode batch -source genOOC_v7.tcl -notrace
vivado -mode batch -source runOOC_v7.tcl -notrace
```

TopDown Zynq:

```
vivado -mode batch -source genTopDown_zynq.tcl -notrace
vivado -mode batch -source runTopDown_zynq.tcl -notrace
```

TopDown v7:

```
vivado -mode batch -source genTopDown_v7.tcl -notrace
vivado -mode batch -source runTopDown_v7.tcl -notrace
```

Note, the gen scripts create Synthesis results and/or OOC constraints and the run scripts produce the implementation results, which can be found in the "Implementation" folder for any device specific OOC/TopDown run.

7. Use python script to view results of all implementations
   (a) Navigate to the `python` folder.
   (b) Execute the getResults.py file, which takes the results folder as an input. Usage: `getResults.py [result_folder]`
   (c) All results found in the `result_folder` will be sent to `output.txt` located in the `python` directory.

Appendix F: Update history for supporting codes

- Version 1.1b1 - Released September 12, 2015
  - Added support for result generation and optimization in batch mode using Vivado
  - Added support for key scheduling done in software
  - Added support for Secret Message Number, Nsec
  - Added npub_read signal for better synchronization between the CipherCore and the PreProcessor. In particular, the CipherCore can indicate whether the current Npub can be overwritten by the PreProcessor or not.
  - Extended PAD_D to support all modes of operation.
  - Extended PAD_AD to support all modes of operation.
  - Extended CIPHERTEXT_MODE to support all modes: 0, 1, and 2.
  - Fixed REVERSE_DBLK behavior. It now correctly operates when CIPHERTEXT_MODE is set to 2.
  - Fixed support for ABLK_SIZE $\neq$ DBLK_SIZE
- Version 1.0b1 - Released July 15, 2015