High-Throughput Low-Area Hardware Design of Authenticated Encryption with Associated Data Cryptosystem that Uses Cha Cha20 and Poly1305

Guard Kanda, Kwangki Ryoo

Abstract: ***In this paper, the hardware design of a low area and a high throughput ChaCha20-Poly1305 that performs the dual authentication-encryption function for a secured communication within hardware devices is presented.***

Cryptographic algorithms- ChaCha20 stream cipher and Poly1305, enhance security margins and achieve higher performance measures on a wide range of software platforms and has proven superior to its counterpart, the AES, in the software domain. This relatively new stream cipher is compared to the benchmark AES, has recently been standardized but their implementations in hardware have had very little to not very desirable results particularly in terms of area. For this reason, it is therefore an active field to make such algorithms hardware friendly.

This research presents a compact, low-area and high throughput chacha20-Poly1305 Authenticated Encryption with Associated Data (AEAD) design. The core architecture consists of the ChaCha20-Poly1305 algorithm. The simplified quarter round designed in the proposed architecture uses the addition, rotation and exclusive-or algorithms operators (gates). This proposed architecture provides an improvement in the operating frequency and area. The architecture was modeled and simulated with Verilog HDL and Modelsim tools for functional and timing correctness. The hardware architecture designed was synthesized with Xilinx's Synthesis Tool (XST) and Synopsis' Design Compiler (DC) using the 0.18µm CMOS standard Cell library. The resulting hardware area in terms of gate equivalent is approximately 11KGE for chacha20 and 21KGE for Poly1305. The design operates at maximum frequency of 420 MHz and 870 MHz for the ChaCha20 and Poly1305 respectively. The proposed design presented in this paper additionally functions at a throughput of approximately 8 Gbps for ChaCha20 with an overall efficiency of 2.35 Kbps/GE when ChaCha20 and Poly1305 are combined into the AEAD_ChaCha20_Poly1305 authenticated encryption core.

Keywords: ChaCha20, Poly1305, Stream Ciphers, ASIC, FPGA

I. INTRODUCTION

The art or process of transforming messages, data or information into forms that are unreadable by none other than the intended recipient is what is refer to as encryption [1]. The efficient design, hardware support and wide application of the Advanced Encryption Standard (AES) has earned it an enviable position in cryptography and encryption [2]. Today, with successes chalked in cryptanalysis, weaknesses that will be identified in this highly popular algorithm in the near future will leave majority of the world in a not so suitable place.

Several attacks on the stream cipher behind the security of over-the-air communication encryption in GSM, the A5/1 and A5/2, have been successful and the National Security Agency's ability to pinpoint a cellular phone and its user's location [3] proves how detrimental this could be should it have fallen into the hands of adversaries. In an attempt to prepare for such an inevitable occurrence, a multi-year project which was dubbed, eSTREAM [4], carried out by the European Network of Excellence for Cryptography (ECRYPT) [5], was started in the year 2004 with the sole aim to identify and promote compact and efficient suits of stream ciphers having the capability of widespread adoption. Two main profiles were used to categorize the eSTREAM portfolio. Stream ciphers that were more suitable for software applications and those that were suitable for hardware were under profile 1 and profile 2 respectively. Initially, the Salsa20 was proposed for both profile 1 and profile 2 and made it to the second phase of the project after which it was drop from profile 2 [6]. This only probable reason why it was taken out was because eSTREAM felt it was too very suitable for hardware devices with highly constrained resource. The main bench mark that was used in measuring a ciphers suitability for hardware was the AES. A variant of the Salsa20, ChaCha20 which possess a much-improved design in terms of diffusion per round [7] was introduced in 2008 by Daniel J Bernstein. This improvement makes the ChaCha20 more resistant compared to Salsa20 [8] but still preserved or sometimes improved its computation time per round.

Connected devices, typically known as smart devices or Internet of Things are currently growing in terms of usage and application. These devices usually exchange between themselves, highly sensitive information that is either environmentally related or socially related. Adversaries who are potential eavesdropper or wiretappers usually take advantage of the security challenges of these devices to get access to sensitive information hence the information being exchanged needs to be secured or the channel carrying the information must be secured, hence development and deployment of strong and highly

Revised Manuscript Received on July 22, 2019.

Guard Kanda, Dept. of Info. & Comm. Eng., Hanbat National University, 34158 Yuseong-Gu Daejeon, South Korea. Email: guardkanda@gmail.com

Kwangki Ryoo*, Dept. of Info. & Comm. Eng., Hanbat National University, 34158 Yuseong-Gu Daejeon, South Korea. Email: kkryoo@gmail.com

efficient cryptographic

Algorithm 1 : ChaCha20 Encryption

Inputs :
256-bit encryption_key(K),
32-bit block_counter (E),
96-bit nonce(N),
n-bit plaintext(M)
Output :
n-bit ciphertext(C)

```
1:  //Initialize Block
2:  b ← [constant, K,E,N]
3:  b'← b
4:  for i=1 to 10 do
5:     quarterRound(b[0],b[4],b[8],b[12])
6:     quarterRound(b[1],b[5],b[9],b[13])
7:     quarterRound(b[2],b[6],b[9],b[14])
8:     quarterRound(b[3],b[7],b[11],b[15])
9:     quarterRound(b[0],b[5],b[10],b[15])
10:    quarterRound(b[1],b[6],b[11],b[12])
11:    quarterRound(b[2],b[7],b[8],b[13])
12:    quarterRound(b[3],b[4],b[9],b[14])
13: endfor
14: b  ← b ⊞ b'
15: C  ← b ⊕ M
16: return C
```

Fig. 1. Algorithm of ChaCha20 Stream Cipher

algorithms such as the ChaCha in the RFC 7539[9] publication are highly required. The main focus of this paper is investigating an efficient (low area, low power, high throughput) hardware implementation of an Authenticated Encryption with Associated Data (AEAD) cryptosystem based on ChaCha20-Poly1305 for both FPGA and ASIC.

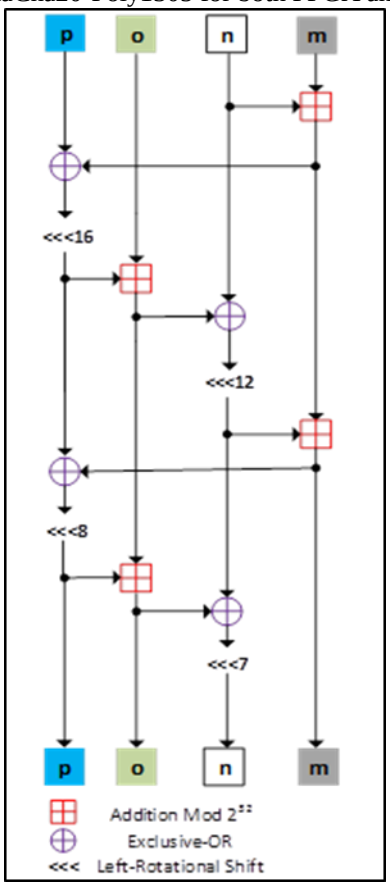

Fig. 2. Graphical Representation of Matrix Manipulation Algorithm

The remainder of this paper is segmented as follows: Section 2 introduces the ChaCha20 algorithm and its constituents, Section 3 describes the Poly1305 authenticator and its mode of operation. In section 4, the proposed hardware architectures for the ChaCha20 and the Poly1305 examined during the research are presented. Simulation and synthesis results of the hardware architectures are in section 5. Finally in section 6, the conclusion, future plan and direction of this research works is presented.

II. CHACHA20 ALGORITHM

The ChaCha20 algorithm, shown in Fig. 1, is mainly composed of the main core round algorithm, known as the Quarter-Round operation seen in Fig. 2. This algorithm works on a 4-by-4 matrix of 32-bits each shown in Fig. 3, resulting in a total of 512-bit data. The upper-left of the matrix is marked index-0 and the bottom right marked index-15. The ChaCha20 as can be deduced from the name required a total of 20 rounds to obtain the final keystream used to create the stream cipher. The rounds are executed as column and diagonal rounds alternatively.

$$\begin{aligned} m &= m + n \\ p &= p \curlywedge m \\ p &= p \lll 16 \end{aligned} \tag{1}$$

$$\begin{aligned} o &= o + p \\ n &= n \curlywedge o \\ n &= n \lll 12 \end{aligned} \tag{2}$$

$$\begin{aligned} m &= m + n \\ p &= p \curlywedge m \\ p &= p \lll 8 \end{aligned} \tag{3}$$

$$\begin{aligned} o &= o + p \\ n &= n \curlywedge o \\ n &= n \lll 7 \end{aligned} \tag{4}$$

Typically, four columns Quarter-Round and four diagonal Quarter-Rounds shown in Fig. 4, are combined to form a Double-Round operation. For this reason, each of the four rounds: diagonal or column, is termed a quarter (one-fourth) of the Single-Round. To perform a complete Double-Round computation, a total of 20 rounds is required. The computation performed with the Quarter-Round will then require a total of 80 rounds to be executed. The main binary operators employed in this algorithm are the addition modulo 232, exclusive-OR (XOR) and binary rotation operations. This is typically referred to as the ARX as shown in Fig. 1. Fig. 3 shows the initial setup of the state matrix. The head or top row of the matrix is occupied by four 32-bit constants, resulting in a 128-bit long of constant values **0x61707865, 0x3320646e, 0x79622d32, 0x6b206574**. This constant value which translates into “expand 32-byte k”. The constant value is designed to reduce the amount of data an attacker can control. The initial state matrix setup has the mid-section (half of the size) of the state matrix being filled by the encryption key, a total of 256 bit.

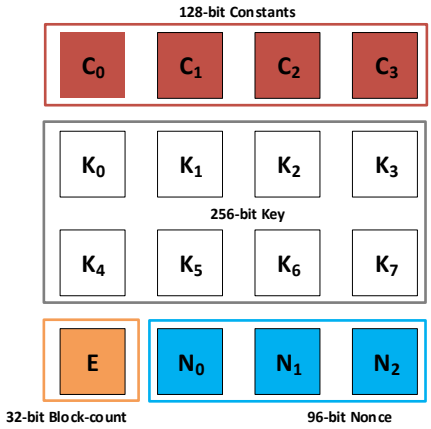

Fig. 3. Initial State Matrix Setup for the ChaCha20 Stream Cipher

Then follows a 32-bit block counter. The block counter that is used to distinguish each 64-byte (512-bit) block of data that is encrypted. This effectively means that the ChaCha20 can encrypt data in excess of 256 gigabytes with the same key.

The nonce which is the last 64 bits of the state matrix block is a unique number that is used to encrypt each block. That is, the nonce should not be repeated for the same key. In other words, the nonce and the counter can be combined to form the same purpose. This means that effectively a 96-bit nonce to encrypt a 256- gigabyte of data.

III. POLY1305 ALGORITHM

Poly1305 is a Message Authentication Code (MAC) that is also used in cryptography to provide authenticity of an encrypted message. The Poly1305 MAC algorithm shown in Fig. 5 was also design and created by Daniel J Bernstein [10]. It verifies how authentic a message is and its integrity as well. A MAC is examined first during a communication between two parties. If the MAC a receiver computed is not same as what which was received from the sender, the sent message has probably been altered and hence the integrity and authenticity of any data or message (Information) can be evaluated using the MAC. Poly1305 together with the ChaCha20 has been standardized in the RFC 7539 [9]. The initial proposal was the Poly1305-AES. This design based its key expansion on the AES block cipher algorithm and hence its name then.

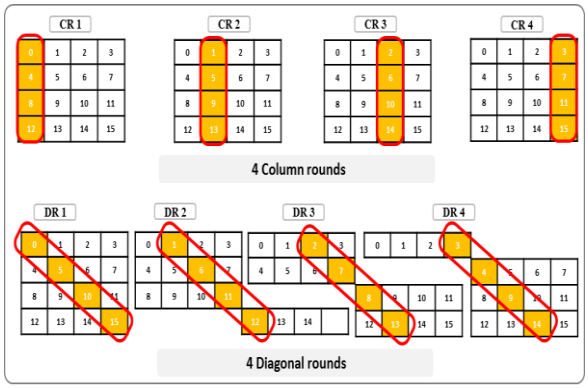

Fig. 4. Column and Diagonal Matrix setup used in Computing Rounds in ChaCha20

It used a 128-bit AES key, an additional 128-bit key) and a nonce of 128-bit, to compute 128-bit authentication Tag off a message of variable length. The variable length message is chopped into 128-bit block chunks which becomes the polynomial coefficient that is evaluated modulo a prime number 2^{130}-5. The name Poly**1305** is derived from the use of the $2[^{130}-5]$, the prime number that is used in performing the modulus arithmetic.

Current use of the Poly1305, in NaCl [11] is with the Salsa20 rather than the AES. Transport Layer Security (TLS) the successor to the Secure Shell (SSH) protocol, implemented the authenticated encryption protocol based on the ChaCha20 algorithm. Current implementations of the ChaCha20-Poly1305 are its use by Google in securing (TLS/SSL) connection between the Chrome browsers on Android phones and Google's servers [12].

IV. HARDWARE ARCHITECTURE AND IMPLEMENTATION

The two architectures evaluated in this research are based on the primitives ChaCha20 and Poly1305. The Hardware Implementation of these algorithm focuses on improving these two core algorithms in term of area, speed and throughput. The ChaCha20 operates by generating a keystream which is the result obtained after adding the initially constructed state matrix to the resulting matrix after the rounds of computation, 20 for the 4xQR architecture and 80 for the 1xQR architecture for this research. This keystream is then combined with the plaintext to be produced the ciphertext. The core of the ChaCha20's computation is the quarter round computation. This particular structure can be implemented in several ways. Examination of the design in both pipeline and parallel architectures were examined. Designs using the pipeline approach reported a larger hardware area while improving operating frequency drastically. This is due to the reduction in the critical path of the architecture.

Algorithm 2 : ChaCha20-Poly1305 Authenticator

Inputs : 256-bit keystream,
message(msg),
Output : 128-bit Tag

```
1:  //Poly1305
2:  r      ← keysteam [255:128]
3:  r      ← clamp(r)
4:  s      ← keysteam [127:0]
5:  accm   ← 0
6:  p      ← (1<<130) - 5
7:  for i=1 to msg_size/16 do
8:    block  ← 0x01 ⊔ msg[127:0]
9:    accm   ← accm + block
10:   accm   ← (accm * r) mod p
11: endfor
12: Tag  ← (accm + s) mod 2^128
13: return Tag
```

Fig. 5. Algorithm of ChaCha20 Based Poly1305 Authenticator

For the parallel architecture examined, the design resulted in a smaller hardware area in terms of gate count or gate equivalent with a significantly reduced operating frequency.

The scalability of this Quarter-Round implies that the hardware design can be divided into simpler computation steps which will allow for design trade-off between area and speed.

A. QuaterRound_8 Architecture

The basic implementation of the quarter round computation is shown in the algorithm in Fig. 6. The basic unit of this computation has the structure of the function defined in (5). The four particular matrix locations to be affected by the diffusion expression in (1) to (4) which forms the Quarter-Round, is passed to the function as shown in (5). The resulting values after the Quarter-Round's computation now becomes the updated values in their respective matrix indexes, ready for the next round of computation. For the design presented in this work, the initial state matrix or vector is built by the concatenation of all the 16 individual bytes labelled in Fig. 3 into a 512-bit long value.

The ChaCha20_State_Generator block is houses the 4xQR and the 1xQR architectures. The initial setup values of the State Matrix are retained in the ***init_state_matrix*** register. For each Single-Round, computations update 16 unique location of the state matrix or vector. For this reason, all four of the column rounds are can be executed in parallel. This implies that a column or diagonal Single-Round computation can occur in a single clock cycle. With the 4xQR architecture which is a Single Round in effect, will require 2 clock cycles for one Double-Round (a combination of 4 column and 4 diagonal Quarter-Rounds).

quarterround(index[a],index[b],index[c],index[d]) (5)

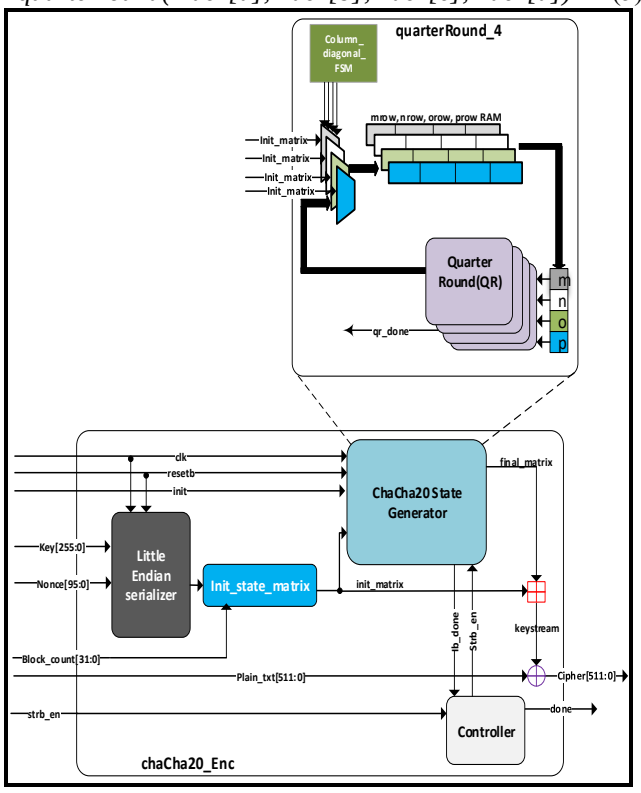

Fig. 6. Hardware Block Diagram for the ChaCha20 4xQR Architecture

This architecture greatly improves the time required to complete the computation from 80 clocks without using the parallel architecture to 20 when the parallel architecture is implemented. Multiplexers and demultiplexer are used for the purposes of routing the appropriate vector or matrix index values to be used for the computation. The is achieved through a Controller logic (FSM) to schedule the in and out of data from the Quarter-Round computation A carry-less addition of the ***final_state_matrix*** and the ***init_state_matrix*** is performed to obtain the keystream after the round's computation. This addition is easily implemented by multiple 32-bit additions and then a final concatenation of the result. This does not influence frequency. Likewise, the cyclic rotational shift is implemented with a straightforward reconstruction by part-selection of the value into a new result.

B. Poly1305 Architecture

Poly1305 module takes as input, 256-bit key, and an arbitrary length message. The 256-bit key to this module is partitioned into two equal halves as can be seen from the algorithm in Fig. 5. The lower half of the key is assigned to the variable ***'r'*** and the upper half is assigned to the ***'s'*** variable. The value of ***'r'*** is clamped. The matrix for the clamping is shown in (6). Each vector location is an 8-bit value occupying 16 indexes to result in a total of 128 bits, the size of ***'r'***. The upper left corner is indexed 15 and the lower right corner is indexed 0. The terms ***OC*** represents ***Odd Clamp*** which is performed to clear the top four bits of that particular vector or matrix index to a value zero. The ***NC - No Clamp***, represents the areas of the 'r' vector that is not affected by the clamp. The final term which is the ***EC*** represents Even Clamp. The Even Clamp is performed to clear the bottom two bits of the value at that vector index or location. This clearing will make the value evenly divisible by 4. Equation (6) indicates that, ***r[3]***, ***r[7]***, ***r[11]*** and ***r[15]*** fall under the Odd Clamp, the ***r[4]***, ***r[8]***, ***r[12]*** fall under the Even Clamp. To perform the clamp, A straightforward bitwise AND is performed on the vector r with the value 128-bit ***0x0ffffffc0ffffffc0ffffffc0fffffff***.

$$r = \begin{bmatrix} OC\ NC\ NC\ EC \\ OC\ NC\ NC\ EC \\ OC\ NC\ NC\ EC \\ OC\ NC\ NC\ NC \end{bmatrix} \quad (6)$$

The value of the prime number **(P)** used to perform the modulo 2^P arithmetic can be computed directly by performing a left shift of 130 on the value 1, and then subtracting 5 from the resulting value as shown in (7). This result is the 131-bit long hexadecimal number: ***0x3fffffffffffffffffffffffffffffffb***.

$$P = (1 << 130) - 5 \quad (7)$$

$$block=(0x01 \ll (BL*3)) \,||\, serialized\_message\ [127:0] \quad (8)$$

The message is processed in 16-byte chunks. A sub-module is designed to determine the number of 16-bytes to be processed. This sub module only uses left shift and OR gates to compute the number of times the modulo arithmetic will be executed. The message processing is done from the least 128-bit through to the highest bit. A signal short is asserted if the message is not an evenly divisible 16-byte long message.

The short signal is required to enable the right data formatting and construction. A binary shift register is used to perform the block chucks.

A 128-bit right shift is performed after every round of the modulo computation. Prior to the message being processed for the modulo computation,16-byte long message chunks are serialized to little endian. The result is padded with the hexadecimal value 0x01. A register named block of size 131-bit is used to keep the value of the chuck blocks after every 16-byte block is computed. An accumulator register named ***"accum"*** is used to accumulate the result from each 16-byte computation. The initial value of ***accum*** before computation begins is 0. The resulting ***accum*** is multiplied by the ***'s'*** value (one half of the input key mentioned earlier) and reduced by modulo $\mathbf{2^P}$.

This is performed iteratively for the number of even 16-byte messages that can be obtained from the arbitrary-length message. In the event that a short signal is asserted, a final round of iteration is again computed but for this computation, the block value that is generated does get to be serialized as the remaining message is not a 16-bytes long. The appended bit is easily performed with the expression shown in (8). This expression can easily be achieved with logical-OR and left shift operators performed on a 131-bit resulting register. The BL term in (8) represents the byte-length register typically holding the byte-length of the remaining message to be after the number of block chucks have been determined. By default, this value is a constant 16 until a ***short*** signal selects the remaining byte-length for the additional final round of computation when asserted. A simple and straightforward method of computing $\boldsymbol{A = B.C \bmod R}$ where ***R*** is a k-bit natural is by multiplying ***B*** by ***C*** to obtain a 2k-bit product term and then reducing the product term modulo ***R***. The main architectures used to perform the modulo reduction (mod $(2^{130}-5)$) computation is the Double-Add-Reduce (DAR) algorithm whose architecture is shown in Fig. 8 and the DAR with the carry-stored encoding (CSA) [13]. To perform the DAR computation, a modulo arithmetic needs to performed the addition and doubling.

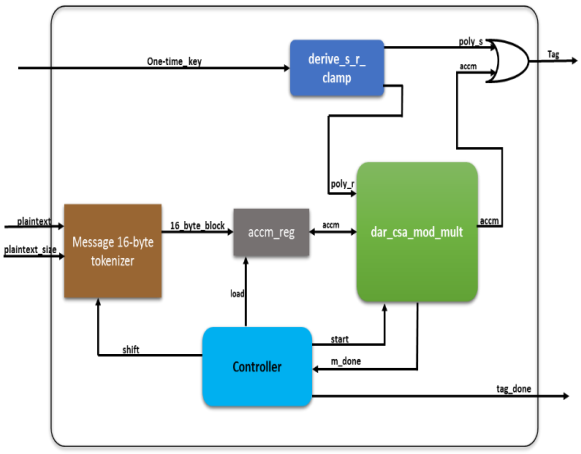

Fig. 7. Block Diagram of Proposed Poly1305 Architecture

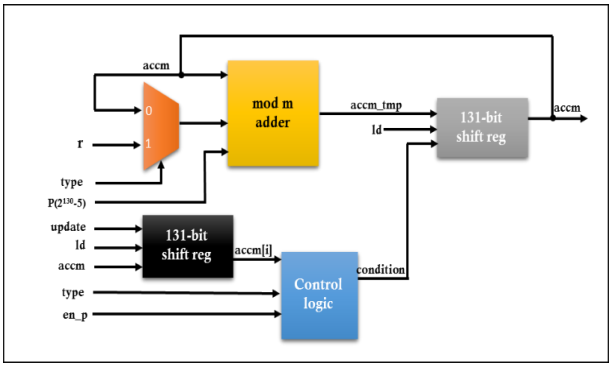

Fig. 8. Block Diagram of the Double, Add and Reduce Multiplier implemented in Poly1305_DAR

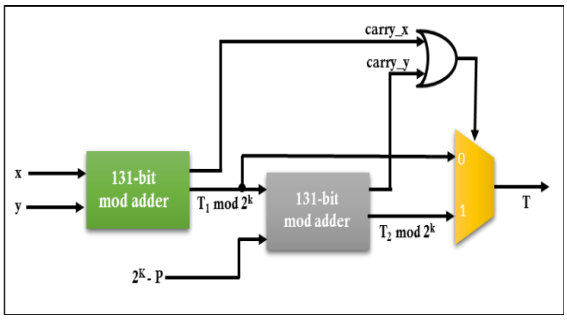

Fig. 9. Block Diagram of Proposed Poly1305 Architecture

The multiplicand term is scanned from left to right. If a bit "1" is encountered through the iteration, we perform both the double the multiplicand and add the result to the product but only perform the doubling of the multiplicand in all other cases without the addition.The architecture shown in Fig. 9, represents the modular addition computation implemented in this design. These algorithms perform the modulo computation alongside an FSM (controller), multiplexors and demultiplexers to form the poly1305 architecture shown in Fig. 7. The two architectures examined in this design was the DAR modulo computation algorithm and a version of that, Double-Add-Reduce algorithm that uses the Carry-Save-Adder (CSA) to speed up the computation. The two designs investigated show that the DAR, occupied a smaller area as most of the circuit components were combinational circuits. This resulted in a very small hardware area while incasing the critical path of the design and hence affecting the execution time and operating frequency of the DAR based Poly1305. To improve this design, the CSA principle is used. For this, the addition is carried out with the CSA register. This showed a far improved result among the two designs. As this design is a blend between the DAR and the CSA, the critical path is greatly reduced improving the frequency and computation time drastically but at a cost of a larger hardware area.

C. Authenticated Encryption with Associated Data

Authenticated Encryption with Associated Data (AEAD) which is a variant of Authenticated Encryption (AE) is the means by which data or information can be encrypted with the assurance of its integrity, authenticity or confidentiality. The mean the functions of encryption and authentication can both occur concurrently.

There are generally three main approaches to executing the authenticated encryption. The form which involves encrypting the data and then using portions of the encrypted data to generate a MAC tag known as the ***Encrypt-then-MAC***. The other form is generating a MAC from the plaintext or data to be encrypted. This MAC is then sent in addition to the encrypted plaintext (ciphertext) in what is known as the ***Encrypt-and-MAC***. The final is where the MAC is generated from the plaintext. The MAC and the plaintext are then combined to form a new intermediate data. This intermediate data is then encrypted to form a ciphertext. This form is termed the ***MAC-then-Encrypt***. The ChaCha20-Poly1305 implements a variant of this form of authenticated encryption used in TLS and its predecessor the SSL [14,15]. The Associated Data that is appended to this form of authenticated encryption is to ensure it is contextually accurate. What this means is that, moving a portion of a valid ciphertext to another portion will turn out to be invalid and cause its detection.

D. AEAD_ChaCha20_Poly1305 Architecture

The final architecture shown in Fig. 10, was implemented using the two modules the ChaCha20 stream cipher and the Poly1305 authenticator is presented in this sub section. The main components modules of the overall architecture use the individually built modules. Since this is a variant of ***MAC-then-Encrypt***, the key for the authentication is generated using the ChaCha20. For this key Generation, the ***block_count*** is kept at zero. After the done signal is asserted, the keystream that is generated will be used to form the key to the Poly1305. The highest 256-bits of the keystream is captured and used as the poly1305 one-time key. The keys are clamped as explained in the section above and the Poly1305 module is enabled to begin execution. When the ***poly_done*** signal is asserted, we have a 128-bit value which will serve as our authentication tag for the particular batch of data being encrypted. The ***Main_Controller*** unit shown in Fig. 10 asserts the signal for the chacha20 module to be execute again to now encrypt the data. The same key, nonce but with the block_count now set to one and increases for each block. The increment can be linear or randomly generated This ensures that the effective nonce is different for each block of 512-bit chunk of data to be encrypted. After this has completed, the module ***AEAD_Recon_Data*** is enabled for a data reconstruction for the tag generation. The reconstructed data is made up of the 4 main parts bulleted below

- The authenticated tag initially generated
- The computed ciphertext
- The size of the Addition Data: AAD, in byte represented as a 64-bit little-endian integer
- The size of the ciphertext in byte represented as a 64-bit little-endian integer

The data is reconstructed by fist placing the ***AAD*** data from bit zero upwards. This is followed by the 64-bit size of the AAD: ***AAD_size***. Next in the concatenation is the ***ciphertext*** that has been generated and then finally the 64-bit little endian integer representing the size of the ciphertext. The design can be parameterized to handle variable sizes. For this design, the message length used is 512-bit and an AAD used is 96-bits. This implies that a reconstructed data of size 736-bit long for

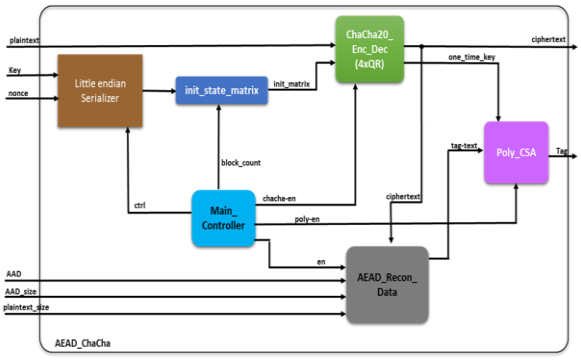

Fig. 10. Block Diagram of the AEAD_ChaCha20_Poly1305 Proposed Architecture

the Poly_CSA. The total clock cycles required to generate the authentication tag and the ciphertext is a total of 1350 cycle. 20 clock cycles for generating the ***one-time-key*** for authentication, 20 clock cycles to generate the ciphertext, 5x (262) cycles required for the modulo reduction.

V. RESULT AND DISCUSSION

The hardware implementation of the implementation of the AEAD_ChaCha20_Poly1305 core architecture was modelled using the Hardware Descriptive Language (HDL), Verilog. The Integrated Synthesis Environment (ISE) tools employed during the design were the Xilinx 14.7 edition and Vivado 2017.2 edition. The design was synthesized on the Virtex 7 FPGA having the XC7V2000T. The Virtex 7 board was used to perform board test to assess the functional correctness and accuracy of the proposed architecture. The size the design occupies in terms of the number of Look-Up Tables (LUTs) and Slices used by the FPGA to implement the design are documented in Table- I. Synopsis design compiler tool was also used to determine the Gate count of the ChaCha20, Poly1305 and the AEAD_ChaCha20_Poly1305 core architectures. For Performance evaluation and efficiency of the proposed architecture, parameters from [16] shown in (9) and (10) were used to determine the analyze core.

$$\text{Throughput} = (\text{Freq. x No. of bits}) / (\text{No. of Cycles}) \quad (9)$$

$$\text{Efficiency} = (\text{Throughput (Mbps)}) / (\text{Area(KGE)}) \quad (10)$$

The operating frequency of the proposed design Quarter-Round(1xQR) and Single-Round (4xQR) recorded 182.40 MHz and 161.02 MHz respectively. The designs of the Poly_DAR and Poly_CSA shows that the use of the Carry Save Adder in performing the modular addition increased the frequency drastically recoding about 121% rise in frequency. The Timing simulations of the ChaCha20 Quarter-Round (1xQR) is shown in Fig. 11 from which the final keystream and ciphertext obtained is shown in the simulation alongside the input key, plaintext and the nonce. Fig. 12 shows a timing simulation of the ChaCha20 Single-Round (4xQR) which indicates the final state values of the state matrix prior to the addition of the initial state matrix values. This simulation presents the individual values that obtained at the end of the 20-clock cycle computation. Fig. 13 shows the timing simulation diagram of the Poly1305 module with details regarding "r" and "s" key generation, the bulk message sliced

into block chunks and also the CSA algorithm that performs the modulus computations. Table- II presents the result of ASIC implementation of the ChaCha20, Poly1305 and the AEAD_ChaCha20_Poly1305. Throughput values recorded in Table- II demonstrates that ChaCha20 is comparable to candidates of the eSTREAM project that are hardware oriented.

Table- I: AEAD Based ChaCha20 and Poly1305 FPGA Implementation Results

FPGA Device	Architecture	Design	Area (Slices)		Frequency [MHz]	Throughput [Gbps]
			Registers	*LUTs*		
VIRTEX 7 XC7V2000T (FLG1925)	ChaCha20	4xQR	566	1692	161.02	4.122
		1xQR	780	940	182.40	1.167
	Poly1305	Poly_CSA	1087	1742	510.75	0.249
		Poly_DAR	963	1376	230.48	0.112
	AEAD	AEAD_ChaCha20_Poly1305	3383	4921	162.16	0.061

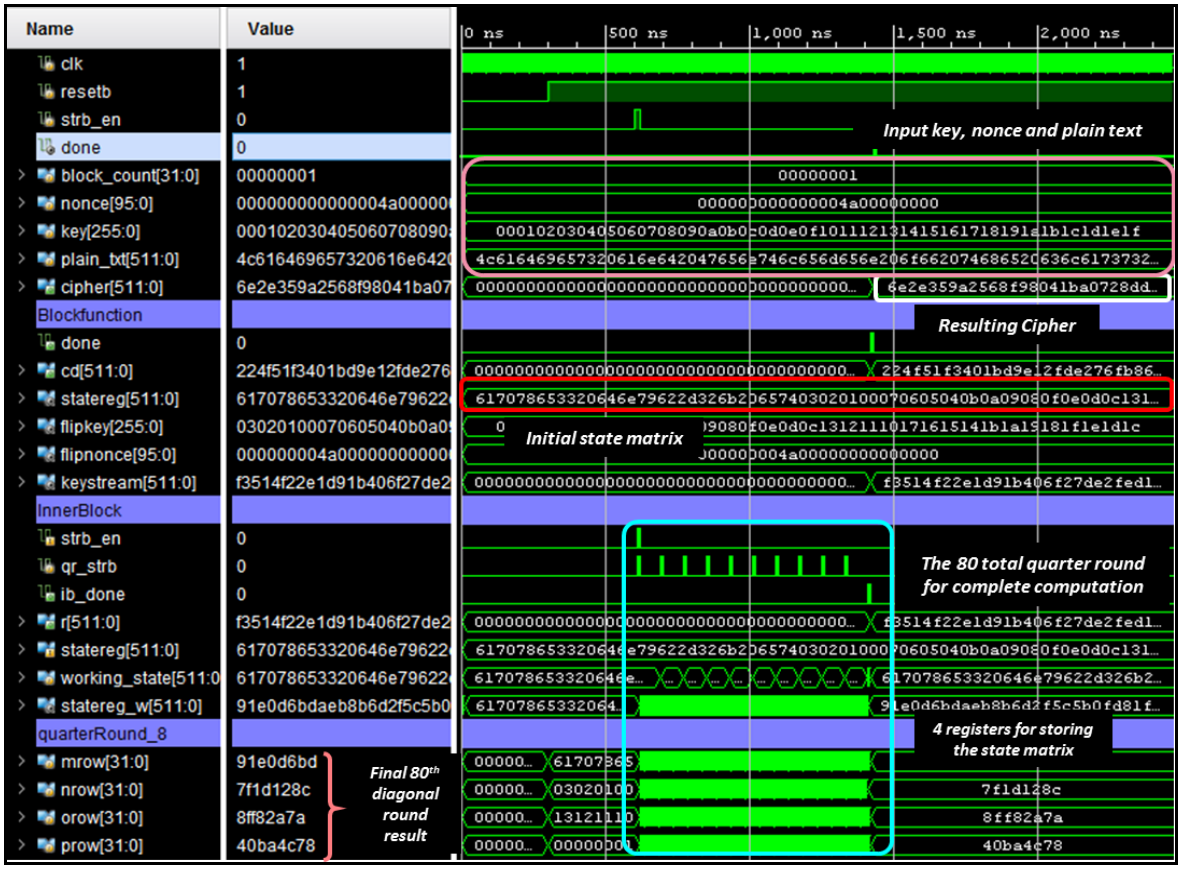

Fig. 11. Timing Simulation of ChaCha20 Implementation using 1xQR Architecture

Table- II: AEAD Based ChaCha20 and Poly1305 FPGA Implementation Results

Category	Design	Technology [nm]	Area [KGE]	Frequency [MHz]	Throughput [Gbps]	HW-Efficiency [Kbps/GE]
ChaCha20	4xQR	180	20.71	312	7.987	385.66
	1xQR	180	11.00	420	2.688	244.36
	4xQR [17]	180	28.11	215	5.505	195.84
	1xQR [17]	180	16.69	196	1.252	75.03
Poly1305	Poly_CSA	180	20.89	870	0.425	20.68
	Poly_DAR	180	16.00	556	0.272	17.12
AEAD	AEAD_ChaCha20_Poly1305	180	50.10	310	0.118	2.35

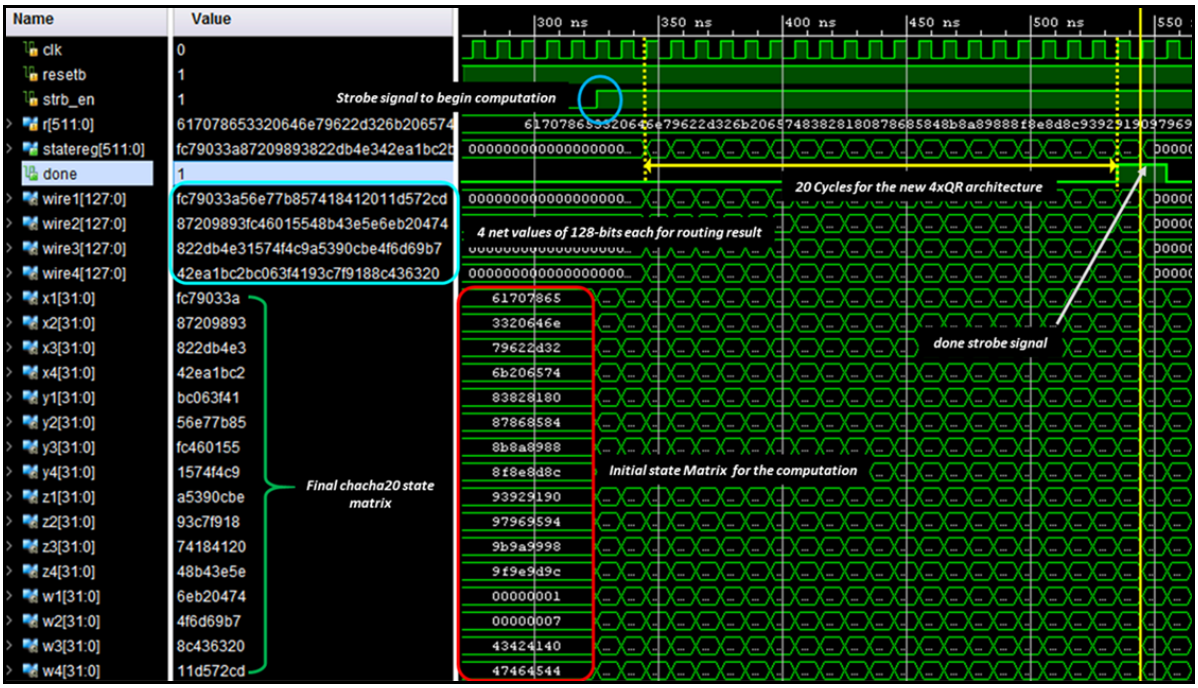

Fig. 12. Timing Simulation of ChaCha20 Implementation using 4xQR Architecture

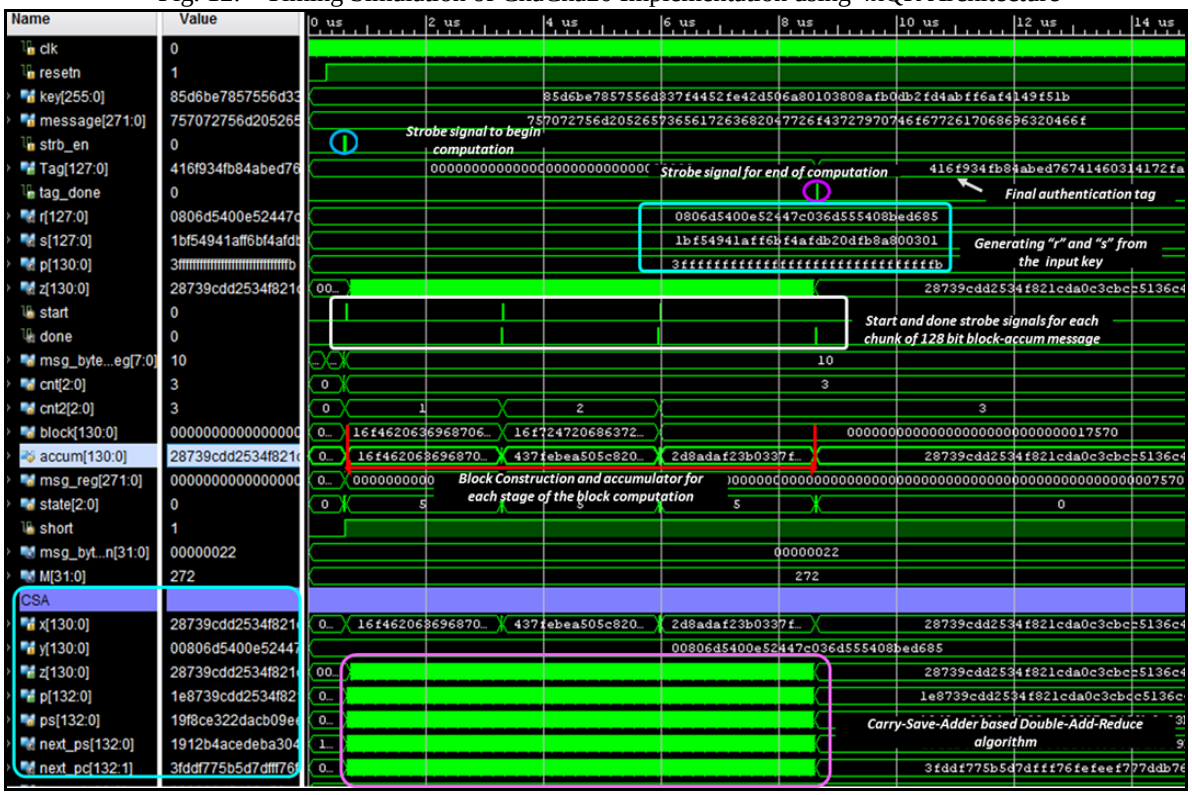

Fig. 13. Timing Simulation of the Poly1305_CSA Architecture

From the ASIC implementation, the proposed architectures the ChaCha20 1xQR and the 4xQR both demonstrate higher performance measures. The 1xQR architecture, which completes its operation in 80 clock cycles and the 4xQR which completes in 20 cycles recorded a 114% and 45% increase in throughput respectively compared to the design in [17]. The highest throughput was recorded with the ChaCha20 Single-Round(4xQR) with a value of approximately 8 Gbps. To determine the Area of the ASIC implementations, the total gate count was divided by the area of a two-input, one-output NAND gate of size 9.7µm2 in the 0.18µm standard CMOS cell library

VI. CONCLUSION AND FUTURE WORK

This paper presented the VLSI / hardware architecture implementation of the stream cipher ChaCha20 and the authenticator Poly1305. In all, a total of 5 architectures were examined in this paper. The proposed authenticated encryption cryptographic core architecture occupies smaller hardware area compared to some existing designs particularly that of [17].

Our proposed architecture operates at a higher frequency and performs executions faster, making it suitable for its implementation in modern day ubiquitous hardware devices that exchange very sensitive personal data among devices to provide the needed security and reliability enhancement. The individual designs were simulated to ascertain their functional correctness and synthesized to determine the area or gate count. The 180nm CMOS standard cell library was also used in the ASIC synthesis to have a common ground for design comparison and evaluation. With this very promising results, future direction of this design will include an SoC system integration of this design into the ECIES scheme being developed in [18]. Further research into faster methods of computing the modulo reduction will be investigated in order to improve upon the hardware efficiency. Finally, the built SoC architecture will be tested using a test bed software program being built alongside.

REFERENCES

1. M. Jackob, "History of Encryption" SANS Institute Information Security Reading Room, Maryland, USA, Available: https://www.sans.org/reading-room/whitepapers/vpns/history-encryption-730
2. M. A. Alomari, K. Samsudin, A. R. Ramli, "A Study on Encryption Algorithms and Modes for Disk Encryption," in International Conference on Signal Processing Systems. Singapore, May. 2009, pp. 793-797,
3. D. DeLong, "How the NSA pinpoints a mobile device", Washington Post, Sept. 2006. Available: https://www.documentcloud.org/documents/888710-gsm-classification-guide-20-sept-2006.html
4. "The eSTREAM Project", Sept. 2008. Available from: https://www.documentcloud.org/documents/888710-gsm-classification-guide-20-sept-2006.html
5. "ECRYPT - European Network of Excellence in Cryptography". Available from: http://www.ecrypt.eu.org/ecrypt1
6. S. Babbage, C. De Cannière, A. Canteaut, C. Cid, C. Paar, G. Henri, T. Johansson, M. Parker, B. Preneel, V. Rijmen, M. Robshaw, H. Wu, "eSTREAM Short Report on the End of the Second Phase", Mar. 2012. Available from: http://www.ecrypt.eu.org/stream/PhaseIIreport.pdf
7. D. J. Bernstein, "Chacha, a variant of salsa20", Illinois, Jan. 2008. Available from: http://cr.yp.to/chacha.html
8. D. J. Bernstein, "The Salsa20 family of steam ciphers" in Robshaw, M., Billet, O. (Eds.), New Stream Cipher Design: the eSTREAM Finalist. LNCS, 4986(0302-9743): Jul. 2008, pp. 84-97, https://doi.org/10.1007/978-3-540-68351-3_8
9. Y. Nir, A. Langley "ChaCha20 and Poly1305 for IETF Protocols" IETF RFC 7539, May. 2015. Available: https://tools.ietf.org/html/rfc7539
10. D. J. Bernstein, "The Poly1305-AES message-authentication code" In Gilbert H., Handschuh H. (Eds): Proceedings of Fast Software Encryption, 3557, Feb. 2005, pp. 32-49, Available: https://doi.org/10.1007/11502760_3
11. D. J. Bernstein, T. Lange, P. Schwabe, "The security impact of a New Cryptographic Library" In Hevia A., Neven G. (Eds): Progress in Cryptology – LATINCRYPT, 7533, Oct. 2012, pp. 159-178, Available: https://doi.org/10.1007/978-3-642-33481-8_9
12. E. Bursztein, "Speeding up and strengthening HTTPS connections for Chrome on Android", Apr. 2014. Available from: https://security.googleblog.com/2014/04/speeding-up-and-strengthening-https.html
13. J. Deschamps, J. L. Imana, G. Sutter, Hardware Implementation of Finite-Field Arithmetic, McGraw Hill, 2009.
14. A. Freire, P. Karlton, P. Kocher. "The Secure Sockets Layer (SSL) Protocol Version 3.0", IETF RFC 6101,2011. Available from: https://tools.ietf.org/html/rfc6101/.
15. B. Mihir, N. Chanathip, "Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm", Journal of Cryptology, 21(4), Jul. 2008, pp.469-491, Available: https://doi.org/10.1007/s00145-008-9026-x
16. H. Mahdizadeh, M. Masoumi, "Novel Architecture for Efficient FPGA Implementation of Elliptic Curve Cryptographic Processor Over GF (2163)", In: IEEE Transaction on Very Large-Scale Integration (VLSI) Systems, 21(12), 2013. pp. 2330-2333, Available: https://doi.org/10.1109/TVLSI.2012.2230410
17. L. Henzen, F. Carbognani, N. Felber, W. Fichtner, "VLSI Hardware Evaluation of the Stream Ciphers Salsa20 and ChaCha, and the Compression Function Rumba", 2nd International Conference on Signals, Circuits and Systems, Dec. 2008, pp.1-5. https://doi.org/10.1109/ICSCS.2008.4746906
18. G. Kanda, K. Ryoo, "Securing Ubiquitous Hardware Devices with Elliptic Curve Integrated Encryption Scheme", Journal of Advanced Research in Dynamical and Control Systems, 10(14-SI), Dec. 2018, pp. 314-324.

AUTHORS PROFILE

Guard Kanda was awarded a BSc. Degree in Computer Science from Kwame Nkrumah University of Science and Technology, Ghana, in 2012 and an M.Eng. Degree in Information and communication engineering from Hanbat National University, South Korea in 2016. Since 2017, he has been working towards his Ph.D. degree at the same department in Hanbat National University. His research interests include SoC Design and Verification Platforms, Lightweight Cryptography, Hardware and Embedded Systems Security, and Hardware-Software co-design.

Kwangki Ryoo was awarded his BSc., MSc, and Ph.D. Degrees in Electronic Engineering from Hanyang University, Korea in 1986, 1988 and 2000 respectively. From 1991 to 1994, he was an Assistant Professor at the Korea Military Academy (KMA) in South Korea. He later worked as a Senior Researcher at the Electronics and Telecommunications Research Institute (ETRI), Korea, from 2000 to 2002. He was a Visiting Scholar at the University of Texas in Dallas from 2010 to 2011. Since 2003, he has been a Professor at Hanbat National University, Daejeon Korea. His research interests include Engineering Education, SoC Platform Design and Verification, Image Signal Processing and Multimedia Codec Design, and SoC Design for Security.