cryptography

MDPI

Article

ChaCha20–Poly1305 Authenticated Encryption with Additional Data for Transport Layer Security 1.3 †

Ronaldo Serrano *, Ckristian Duran, Marco Sarmiento, Cong-Kha Pham and Trong-Thuc Hoang

Department of Computer and Network Engineering, The University of Electro-Communications (UEC), Tokyo 182-8585, Japan; duran@vlsilab.ee.uec.ac.jp (C.D.); marco@vlsilab.ee.uec.ac.jp (M.S.); phamck@uec.ac.jp (C.-K.P.); hoangtt@uec.ac.jp (T.-T.H.)

* Correspondence: ronaldo@vlsilab.ee.uec.ac.jp

† This paper is an extended version of our paper published in 18th International SoC Design Conference in Korea (ISOCC 2021), Jeju, Korea, 6–9 October 2021.

Abstract: Transport Layer Security (TLS) provides a secure channel for end-to-end communications in computer networks. The ChaCha20–Poly1305 cipher suite is introduced in TLS 1.3, mitigating the side-channel attacks in the cipher suites based on the Advanced Encryption Standard (AES). However, the few implementations cannot provide sufficient speed compared to other encryption standards with Authenticated Encryption with Associated Data (AEAD). This paper shows ChaCha20 and Poly1305 primitives. In addition, a compatible ChaCha20–Poly1305 AEAD with TLS 1.3 is implemented with a fault detector to reduce the problems in fragmented blocks. The AEAD implementation reaches 1.4-cycles-per-byte in a standalone core. Additionally, the system implementation presents 11.56-cycles-per-byte in an RISC-V environment using a TileLink bus. The implementation in Xilinx Virtex-7 XC7VX485T Field-Programmable Gate-Array (FPGA) denotes 10,808 Look-Up Tables (LUT) and 3731 Flip-Flops (FFs), represented in 23% and 48% of ChaCha20 and Poly1305, respectively. Finally, the hardware implementation of ChaCha20–Poly1305 AEAD demonstrates the viability of using a different option from the conventional cipher suite based on AES for TLS 1.3.

Keywords: ChaCha20; Poly1305; TLS; RISC-V

Citation: Serrano, R.; Duran, C.; Sarmiento, M.; Pham, C.-K.; Hoang, T.-T. ChaCha20–Poly1305 Authenticated Encryption with Additional Data for Transport Layer Security 1.3. *Cryptography* **2022**, *6*, 30. https://doi.org/10.3390/cryptography6020030

Academic Editor: Jim Plusquellic

Received: 6 May 2022
Accepted: 14 June 2022
Published: 17 June 2022

1. Introduction

The cryptography algorithms are used in secure end-to-end communications, encrypting the data to reduce attacks on vital information. Currently, the cipher suite most used in communications is based on Advanced Encryption Standard (AES). However, Transport Layer Security (TLS) added the ChaCha20 cipher suite, generating another options that are different from the AES algorithm with more bit rate [1,2] and side-channel resistance [3] in software implementation. Additionally, TLS defines the new Authenticated Encryption with Associated Data (AEAD) using the ChaCha20 cipher with Poly1305 for a message authentication code [4].

The software implementations of ChaCha20 and Poly1305 primitives are optimized for the high performance and protection of side-channel attacks [5–11]. However, the software implementation generates a limitation in terms of bit rate in the TLS application. As such, some approaches have been explored to increase the bit rate of the AEAD implementation. For example, the in-memory implementations of ChaCha20 improve the bit rate and the security for side-channel attacks [12]. Finally, the hardware implementations provide a high bit rate compared to other implementations. However, the few hardware implementations cannot provide a full implementation of the AEAD compatible with TLS 1.3.

The state-of-the-art of hardware implementations of ChaCha20–Poly1305 AEAD is divided into two parts. First, the academic implementations provide different topologies for the ChaCha20–Poly1305 AEAD and the primitives in FPGA and ASIC implementation.

Pfau et al. [13] propose a scalable ChaChaX implementation in FPGA, using register, memory, and pipeline implementations in the quarter round (QR). Henzen et al. [14] presented the Very Large Scale Integration (VLSI) implementation of Salsa20 and ChaCha stream cipher. Kermani et al. [15] implemented a ChaCha20 and proposed hardware for detecting faults in the cipher. Kanda et al. [16] reported low-area high-throughput ChaCha20 and Poly1305 primitives and the AEAD construction in FPGA and ASIC implementation. Second, different intellectual properties (IPs) of the primitives and AEAD construction are provided for the industry. For example, Rambus [17,18] provides a hardware solution for ChaCha20 and Poly1305 with the possibility of compatibility with TLS 1.3. Silex Insight [19] offers a ChaCha20–Poly1305 AEAD crypto-accelerator with Direct Memory Access (DMA).

This work extends the previous implementation [20] when an approach of ChaCha20–Poly1305 AEAD compatible with TLS 1.3 specification is shown. The main contribution of the current work is the complete implementation of the ChaCha20–Poly1305 AEAD in a system capable of using the TLS 1.3, solving problems in the fragment blocks generated in the typical application. In addition, the ChaCha20 and Poly1305 primitives used in the ChaCha20–Poly1305 AEAD construction are explained. The AEAD construction occupies a 10808-LUT and 3731-FF in an FPGA implementation with 1.4-cycles-per-byte in a standalone implementation, increasing the throughput 15× with 75% of overhead resources in comparison with the related work [16]. Additionally, the AEAD is implemented in an RISC-V environment presenting 11.56-cycles-per-byte using a TileLink bus. The comparison with a software implementation in an RISC-V environment shows an bit rate increase of 1104%. Finally, the AEAD implementation is compared with other implementations based on AES compatible with TLS, demonstrating the viability of the hardware solution.

The remainder of this paper is organized as follows. Section 2 related the importance of the TLS in secure end-to-end communications and the use of the ChaCha20–Poly1305 AEAD in the protocol. Sections 3 and 4 show the algorithm and the hardware implementation of ChaCha20 and Poly1305 primitives, respectively. Section 5 presents the construction of the AEAD with the fault detector. In addition, the system used to measure the software and hardware implementation of the ChaCha20–Poly1305 AEAD in TLS 1.3. Section 6 shows the results of the ChaCha20, Poly1305, and the AEAD in an FPGA implementation. In addition, the results of the hardware implementation are compared with a software implementation in an RISC-V environment. Finally, Section 7 summarizes the paper.

2. Transport Layer Security

Transport Layer Security (TLS) is the most used protocol for providing end-to-end secure channels in computer networks [21]. The Internet Engineering Task Force (IETF) released the standard of TLS. Figure 1 illustrates the TLS percentage of use in websites, reporting 79.4% in the first quarter of 2022 [22], and showing that TLS 1.2 represents approximately half of the use in websites during this period. In addition, the use of TLS 1.3 increased by 600% in use compared to the last year, demonstrating the importance of the new specification in secure end-to-end communications.

In the previous version of TLS, the RC4–MD5, RC4–SHA, and AES–CBC cipher suites were used [23]. However, in TLS 1.3, the cipher suites based on RC4 were removed. In addition, the ChaCha20–Poly1305 cipher suite was introduced in the protocol to increase the performance and resistance of side-channel attacks in software implementations. As such, the cipher suites used in the current version of TLS are AES–GCM [24,25], AES–CCM [26,27], and ChaCha20–Poly1305 [28]. Figure 2 shows the handshake protocol in TLS 1.3. First, the client and server interchange the keys and certificates to establish the connection. When the connection is authenticated, the secure channel is created to exchange information using the AEAD cipher suite selected in the handshake. The speed of the channel mostly depends on the bit rate of the AEAD used in the data exchange.

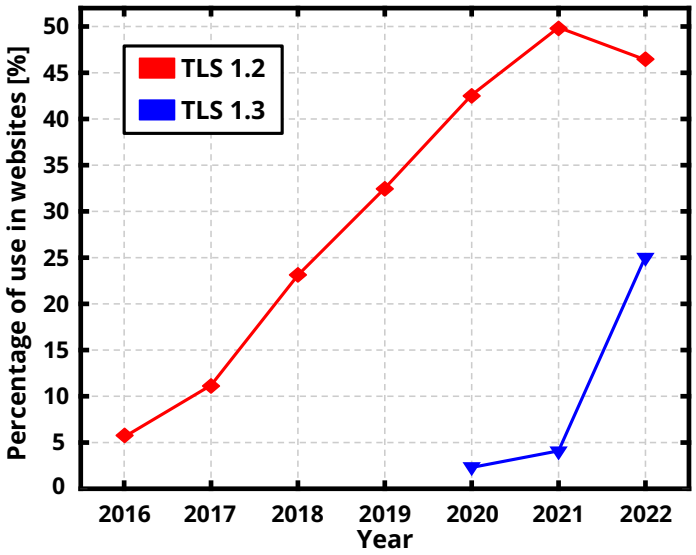

Figure 1. Transport Layer Security percentage of use in websites.

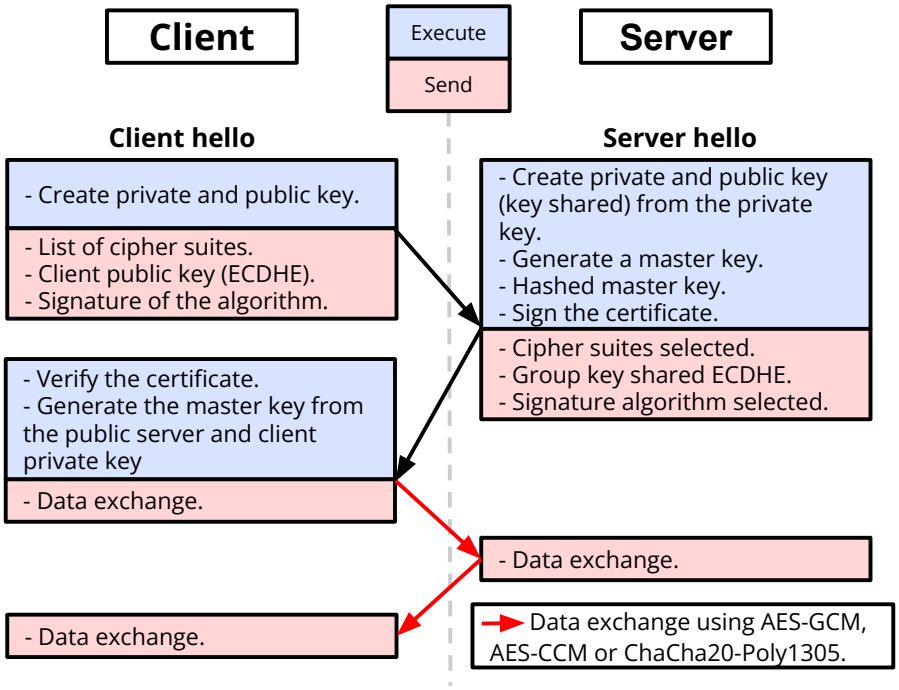

Figure 2. Transport Layer Security 1.3 handshake protocol.

3. ChaCha20

3.1. Algorithm

The ChaCha20 cipher is based on the Salsa20 algorithm [29]. Initially, the algorithm generates an initial matrix (I_M) using a 256-bit key, a 32-bit initial counter, a 96-bit nonce, and four constants defined in the documentation [28]. The I_M has 512 bits and is represented by a 4x4 matrix wherein each matrix entry is interpreted as a 32-bit unsigned integer. Furthermore, the organization of I_M is in little-endian form. The illustration of the matrix is as follows (1).

$$I_M = \begin{bmatrix} 61707865 & 3320646e & 79622d32 & 6b206574 \\ Key_0 & Key_1 & Key_2 & Key_3 \\ Key_4 & Key_5 & Key_6 & Key_7 \\ Counter & Nonce_0 & Nonce_1 & Nonce_2 \end{bmatrix} \quad (1)$$

Algorithm 1 shows the ChaCha20 cipher suite. Initially, an operation matrix (O_M) is initialized with the values of I_M. The ChaCha20 algorithm applies ten double-rounds to the internal state O_M, wherein each double-round consists of eight applications of the quarter-round function. The rounds are finalize when all the states are updated in O_M. The result of the QR operation is saved in the same position as the state taken by the input. Finally, the keystream is obtained by adding the I_M and O_M. In another way, the decryption process is an identical encryption operation that uses the same keystream and cipher stream. Additionally, the counter value is incremented by one per block of plaintext processed to obtain the next I_M.

Algorithm 1 ChaCha20 cipher suite algorithm

Require: $K \in (0,1)^{256}$, $N \in (0,1)^{96}$, $C \in (0,1)^{32}$, $PT \in (0,1)^{*}$
Ensure: $CT = \text{ChaCha20}(K, N, C, PT)$
$I_M \leftarrow \text{Init}(K, N, C)$ ▷ The Initial matrix is organized as (1).
for $x \leftarrow 0$ **to** $([P/512] - 1)$ **do**
  $O_M \leftarrow I_M$
  for $y \leftarrow 0$ **to** 9 **do**
    $O_M[0,4,8,12] \leftarrow \text{QR}(O_M[0,4,8,12])$
    $O_M[1,5,9,13] \leftarrow \text{QR}(O_M[1,5,9,13])$
    $O_M[2,6,10,14] \leftarrow \text{QR}(O_M[2,6,10,14])$
    $O_M[3,7,11,15] \leftarrow \text{QR}(O_M[3,7,11,15])$
    $O_M[0,5,10,15] \leftarrow \text{QR}(O_M[0,5,10,15])$
    $O_M[1,6,11,12] \leftarrow \text{QR}(O_M[1,6,11,12])$
    $O_M[2,7,8,13] \leftarrow \text{QR}(O_M[2,7,8,13])$
    $O_M[3,4,9,14] \leftarrow \text{QR}(O_M[3,4,9,14])$
  end for
  $S \leftarrow Serialize(O_M + I_M)$
  for $z \leftarrow 0$ **to** 511 **do**
    $CT[512x + z] \leftarrow PT[512x + z] \oplus S[z]$
  end for
  $I_M[12] \leftarrow I_M[12] + 1$ ▷ Update the value of the counter in the Initial Matrix.
end for
return CT

The column and diagonal rounds consist of the QR function $(A, B, C, D) = \text{QR}(a, b, c, d)$ which acts on the state as follows (2). The addition denoted in the QR algorithm is a carry-less addition on a 32-bit word.

$$\begin{Bmatrix} x = a + b; & y = (d \oplus x) \lll 16; \\ w = x + y; & z = (b \oplus w) \lll 12; \\ A = x + f; & D = (y \oplus A) \lll 8; \\ C = w + D; & B = (z \oplus C) \lll 7; \end{Bmatrix} \quad (2)$$

3.2. Hardware Implementation

Figure 3 shows the implementation of the ChaCha20 primitive. The ChaCha20 implementation is divided into two parts. First, a *BlockFunction* highlighted in blue takes a 256-bit key, 96-bit nonce, and 32-bit counter to generate an I_M in the *Initial regs*. The states of the I_M are organized in little-endian form. The *Operation regs* then obtains the initial states of the *Initial regs*. All matrices have 16 states, and each state is a 32-bit register. A Finite State Machine (FSM) highlighted in red runs the 20 rounds using the QR modules. The ChaCha20 primitive can have 1, 2, and 4 quarter-round operations in parallel to the round. However, the ChaCha20 algorithm presents the impossibility of paralleling between the column and diagonal rounds. As such, the maximum number of quarter-round operations is four. The FSM controls the input of the quarter-round operators in each column and diagonal round, respectively. Each round only depends on the result of the

previous rounds in O_M. In addition, the *Init* and *Next* signals indicate the start of the process and the next 512-bits to processing, respectively. After 20 rounds, the *Matrix Adder* obtains the final matrix (F_M), adding the *Initial regs* and *Operation regs*. Second, a *Crypto Function* highlighted in green takes the F_M generated in the *Block function*. The 512-bit of *Cipher Text* is obtained by the XOR operation between F_M and the *Plain Text*. When the crypto-function is finished, a *Valid* signal is triggered.

Figure 4 illustrates the QR operation. The inputs *a*, *b*, *c*, and *d* correspond to one stage in *Operation regs*. The QR operation consists of four Add-Rotate-XOR (ARX) cells [13,14]. The rotate operation highlighted in red is implemented without combinational logic to reduce the delay in the ARX cell, replacing the operation with a wire permutation.

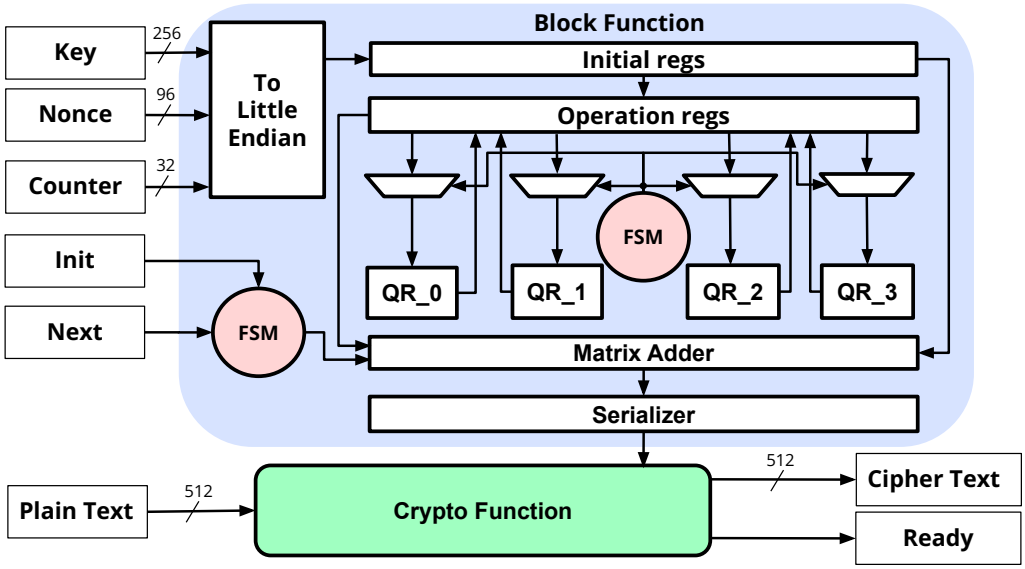

Figure 3. Architecture of the ChaCha20 primitive [20].

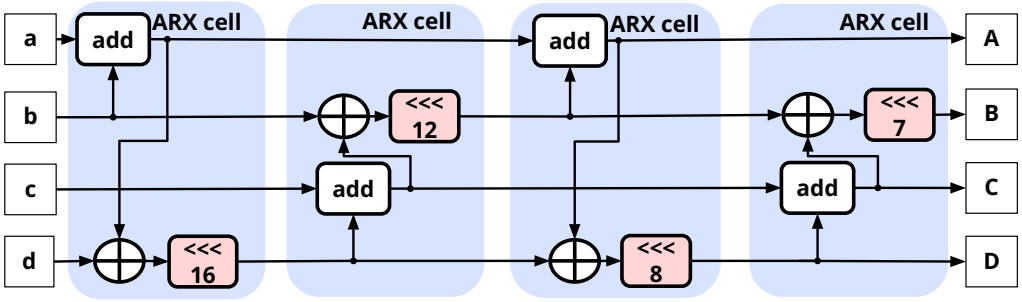

Figure 4. Quarter-round operation [20].

4. Poly1305

4.1. Algorithm

The Poly1305 algorithm is a one-time authenticator using a 32-byte one-time key and an arbitrary-length message to obtain a 16-byte message authentication code (MAC). Algorithm 2 shows the Poly1305 authenticator algorithm. Initially, the key is partitioned into two parts denoted *s* and *r*, respectively. The pair (*s*,*r*) should be unique and unpredictable for each call of the Poly1305 algorithm. However, the *r* and *s* are possible to generate pseudorandomly. Furthermore, *r* can possibly have a constant but needs to be modified [28]. Pad1305 takes the message's length and divides the arbitrary length message in *q* fragments of 16 bytes. The arbitrary-length message is read in little-endian format, and the *r* is clamped. Then, the clamp function clears some bits of *r*, such that $\bar{r} = r_0 + r_1 + r_2 + r_3$—where $r_0 \in \{0, 1, 2, \ldots, 2^{28} - 1\}$, $r_1/2^{32} \in \{0, 4, 8, \ldots, 2^{28} - 4\}$, $r_1/2^{64} \in \{0, 4, 8, \ldots, 2^{28} - 4\}$, and $r_1/2^{96} \in \{0, 4, 8, \ldots, 2^{28} - 4\}$. The accumulator *h* is

initialized, adding the result of the polynomial (3). Furthermore, the MAC is truncated in 128 bits.

$$T = \left(\sum_{i=1}^{q} m_i r^{-q-i+1} \ mod \ 2^{130} - 5 \right) + s \ mod \ 2^{128} \tag{3}$$

Algorithm 2 Poly1305 authenticator algorithm

Require: $K \in (0,1)^{256}, M \in (0,1)^{L}, L \in \mathbb{N}$
Ensure: $MAC = \text{Poly1305}(K, M, L)$
1: $(r, s) \leftarrow K$
2: $m = (m_1, \ldots, m_q) \leftarrow \text{Pad1305}(L)$
3: $\bar{r} \leftarrow \text{Clamp}(r)$
4: $h \leftarrow 0$
5: **for** $i \leftarrow 0$ **to** $(q-1)$ **do**
6: $\quad h \leftarrow h + \text{Polynomial}\ (m, \bar{r}, q, i)$
7: **end for**
8: $MAC \leftarrow h + s \bmod 2^{128}$
9: **return** MAC

4.2. Hardware Implementation

Figure 5 shows the implementation of the Poly1305 core divided into two parts. First, a *PBlock* highlighted in blue generates an initial r and s, using the 256-bit of the *key*. The 128-bit of *Block* is then operated using a Multi-Multiplier and Accumulator (MulAcc), reproducing the polynomial described in (3). The MulAcc implementation consists of a 32-bit unsigned multiplier with a 32-bit accumulator. The Poly1305 primitive has four *MulAcc* in the architecture, processing the 128-bit of the block in one cycle. The documentation [28] determines that the length of the message is arbitrary. As such, the core processes 128 bits in each step. The signal *Block len* indicates the numbers of bytes in each *Block*. Additionally, an FSM highlighted in red controls all *Block* of the message to authenticate and the *MulAcc* interaction. The signal *Init* and *Next* indicate the start of the new message process and another *Block* of the message, respectively. When the last part of the message is introduced, the *Finish* signal is triggered to run the end *MulAcc* operation. Consequently, the data are pushed to the *Final Block*. Second, the *Final Block* highlighted in green takes the data accumulation and s, generating the MAC. The MAC is obtained by adding the s and the data accumulation generated with each *MulAcc*. Furthermore, the result of the sum is truncated into 128 bits. Finally, the signal *Ready* indicates that the *MAC* has been generated.

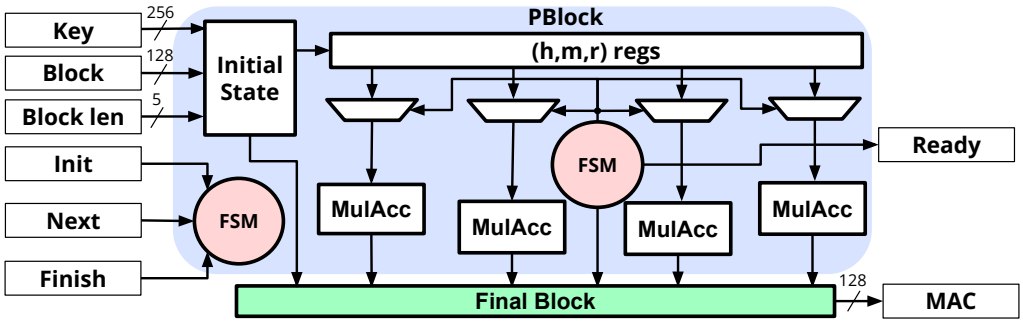

Figure 5. Architecture of Poly1305 primitive [20].

5. System Implementation

In this section, we describe the system used to measure the performance of the implementation and the ChaCha20–Poly1305 AEAD peripheral. In the AEAD implementation, a filter is implemented to mitigate the fault in fragmented blocks in the plaintext.

5.1. SoC Implementation

Figure 6 illustrates the overview architecture of the system used to implement the AEAD construction [30]. The system consists of a Rocket core [31], a 4-KB ROM, an SPI, and the different interruption platforms. In addition, the system uses a TileLink protocol for the system (SBUS) and peripheral (PBUS) buses [32]. The system contains 1 GB of a Double-Data-Rate (DDR) controller. This controller is driven by an Advanced Extensible Interface 4 (AXI4) bus [33], converted from the memory-channel bus (MBUS).

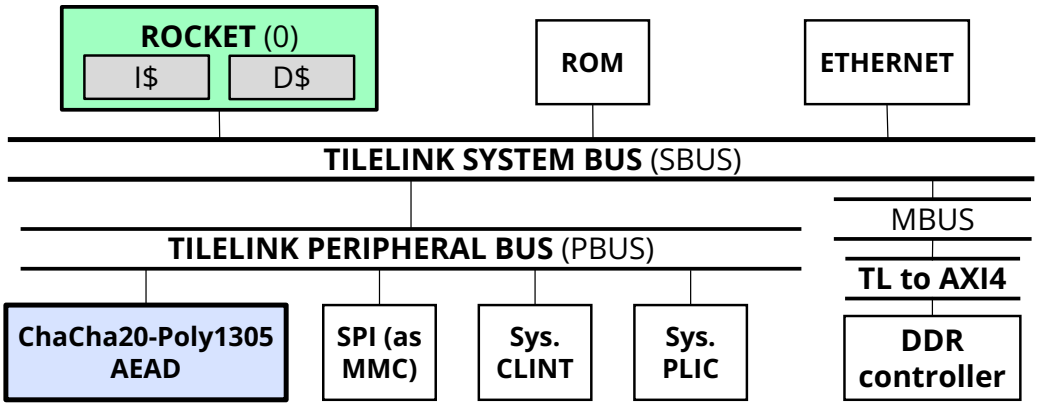

Figure 6. Block diagram of the RISC-V environment.

5.2. ChaCha20–Poly1305 AEAD

Figure 7 illustrates the implementation of the ChaCha20–Poly1305 AEAD peripheral. The standard determines an AEAD construction using a 256-bit key, 96-bit nonce, an arbitrary length of plaintext, and an arbitrary length of Additional Authenticated Data (AAD) [28]. The AEAD implementation uses a TileLink bus in the RISC-V environment described in Section 5.1. The peripheral consists of the ChaCha20 and Poly1305 primitives highlighted in blue and green, respectively.

Additionally, an FSM highlighted in red controls the internal signals of the ChaCha20 and Poly1305 primitives in the peripheral for the correct functionality. Finally, the implementation needs an accumulator and filter to introduce a final block in the Poly1305 primitive and when the plaintext is not a multiple of 512 bits, respectively.

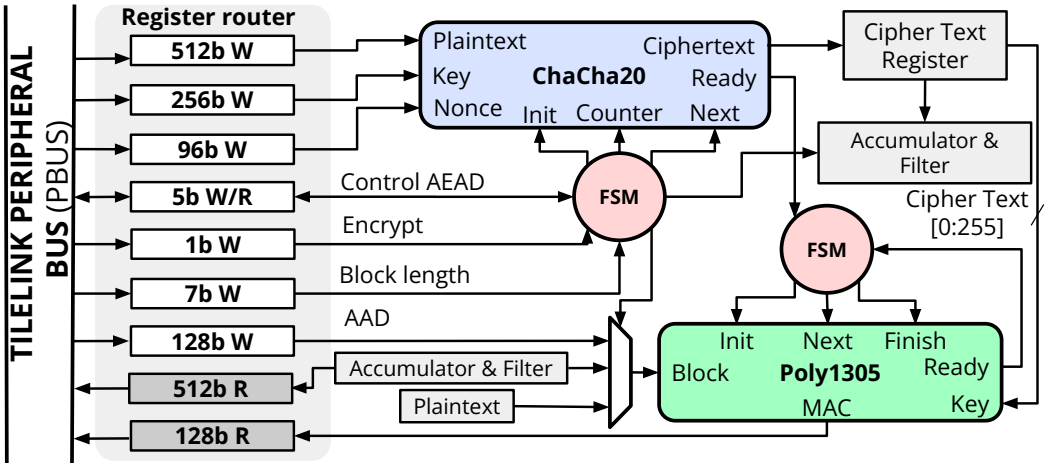

Figure 7. Architecture of the AEAD construction.

Figure 8 illustrates the functionality of the AEAD peripheral in the RISC-V environment. Steps 1–7 describe the conventional operation with the system memory. The first step defines the initial configuration of the AEAD in decryption or encryption depending on the scenario. In AEAD construction, a one-time key is used in the Poly1305 primitive with two

options for creation in the second step. A pseudo-random number generator creates the one-time key. In another way, the ChaCha20 primitive is used with the same key and nonce of the keystream. However, the counter and the plaintext of the ChaCha20 are initialized by zero. The one-time key is the first 255 bits of ciphertext generated for the ChaCha20 primitive. Additionally, the AAD and length in bytes are introduced and processed in the Poly1305 primitive and the accumulator, respectively. One bit in the signal *Control AEAD* indicates that the AEAD construction is ready for the next block when the block is processed. Furthermore, the last block of the AAD is indicated with one bit in the signal *Control AEAD* in the peripheral. Steps 3–5 describe the AEAD process to generate the ciphertext or plaintext in the encryption or decryption mode of the implementation. The mode of the AEAD implementation is defined in the signal *Encrypt* in the peripheral. The counter of the ChaCha20 primitive starts in one of the first blocks processed, and the plaintext length is necessary per each block introduced. When the final block of the plaintext is not a multiple of 512 bits, the peripheral filter of the response of the ChaCha20 eliminates the extra bits generated in the rounds of the primitive. In each result of the ChaCha20 primitive, the results are introduced in the Poly1305 primitive in blocks of 128 bits. Step 6 introduces the final block in the Poly1305 primitive. The final block consists of the result of the accumulator in the peripheral. When the first part is the length of the ADD, and the second part is the length of the plaintext blocks. Finally, the *MAC* is generated in step 7.

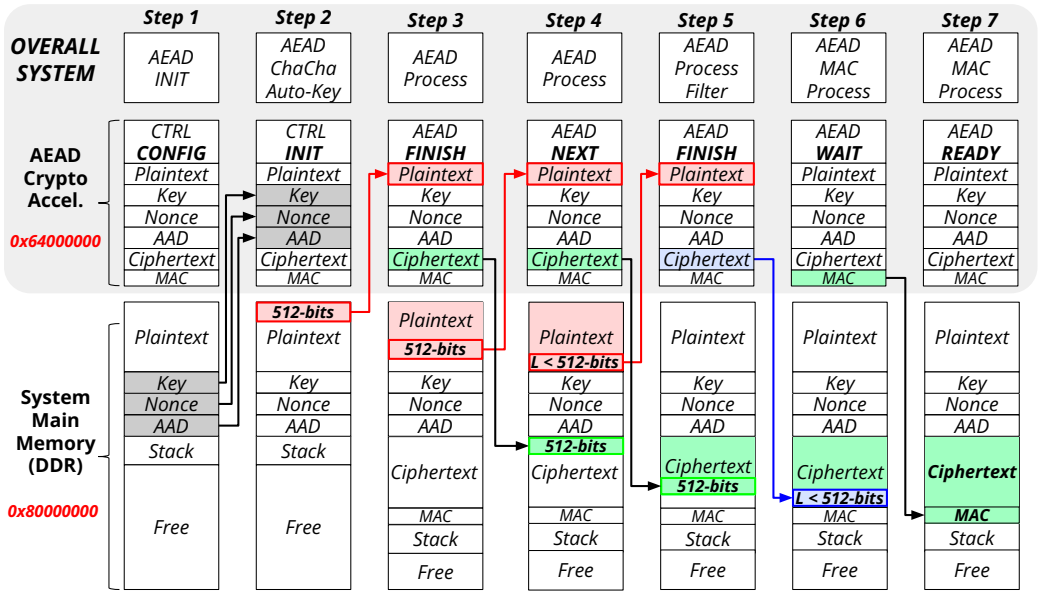

Figure 8. Functionality of the AEAD peripheral into the RISC-V system.

6. Results and Discussion

This section shows the implementation results in Xilinx Virtex-7 XC7VX485T FPGA. The bit rate of ChaCha20, Poly1305, and the AEAD hardware implementations are compared with the software implementation in an RISC-V environment. Finally, the AEAD construction is compared with the related works and the cipher suites based on AES in TLS 1.3.

Figure 9 illustrates the performance comparison between the software and hardware implementations. The performance of the ChaCha20, Poly1305, and AEAD hardware implementations increased by 968%, 195%, and 1104% compared to the software implementation. The software implementation runs in an RISC-V environment using one rocket core [31]. The AEAD implementation result is compared in encryption mode with 16 bytes of AAD.

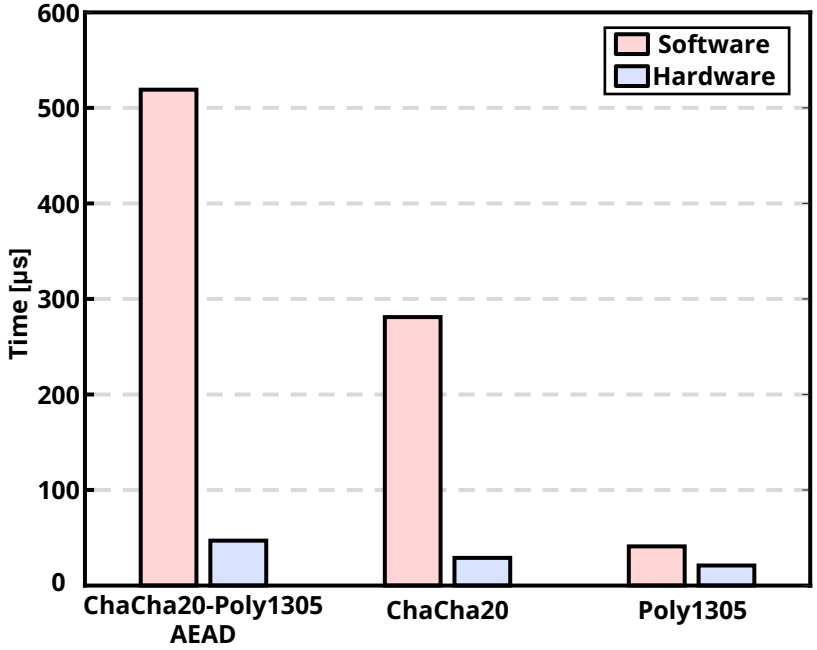

Figure 9. Software and hardware comparison in an RISC-V environment.

Table 1 shows the results of the ChaCha20–Poly1305 AEAD implemented in FPGA. The standalone AEAD presents a 1.4-cycles/byte with 10808-LUT and 3731-FF, using 4-QR and 4-MulAcc modules in the ChaCha20 and Poly1305 primitives, respectively. In addition, the performance with the RISC-V environment is 11.56-cycles/byte. The performance of the AEAD implementation with the system considers the cycles of the process, the control signals, and the data movement in the TileLink bus in the system implemented. The AEAD in a standalone implementation increases the performance by 15× with an overhead of 75.08% of resources. The time for the control signals and the movement of the TileLink bus represents 70.32% of overhead compared to the standalone AEAD implementation. The performance is obtained by 5 MB of data, using 16 bytes of AAD. The principal differences of the related work are the *Accumulator and Filter* sub-modules, removing the errors in fragmented blocks and the performance of each primitive in the AEAD implementation.

Table 1. Performance summary and comparison in FPGA.

	Slices			Performance (Cycles/Byte)	
Module	**LUT**	**FF**	**Total**	**Standalone**	**System**
This work	10,808	3731	14,539	1.4	11.56 ‡
[16]	3383	4921	8304	21.26 †	N/A
[20]	7897	4840	12,737	N/A	21.50 ‡

† Measured with 8 bytes of AAD. ‡ Measured in the same system.

Table 2 shows the comparison in FPGA implementation with the other cipher suites of TLS 1.3. The AES implementations present a relatively high performance in terms of resource utilization compared to ChaCha20–Poly1305. However, the high throughput implementations use a parallel AES core [24,27]. The standalone implementation of AES-GCM [25] reports an improvement of 324% in cycles-per-byte operation using tower field optimizations in the AES module. However, the ChaCha20–Poly1305 reports an increase of 166% of cycles-per-byte in the system implementation. The ChaCha20–Poly1305 depicts 0.948 Gpbs in the standalone implementation with 3034 slices, demonstrating competitive performance resources with the AES-based cipher suites.

Table 2. Summary and comparison results with the others' cipher suites in TLS 1.3

	Module	Device	Resources Utilization		Standalone Imp.			System Imp.
			LUT FF	Slices BRAM	Max. Freq. (MHz)	Througput (Gbps)	Cycles/ Byte	Cycles/ Byte
This work	ChaCha20 Poly1305	Virtex7 C7VX485T	10,808 3731	3034 0	166	0.948	1.4	11.56
[24]	AES-GCM	Virtex7	— —	4194 40	125 ‡	4 *	—	—
[25]	AES-GCM	Cyclone V 5CSEMA4U23C6	4572 2407	— 0	36.46	0.275	1.058	19.19
[26]	AES-CCM	Artix7 XC7A200TL	— —	554 76	177	0.119	11.89	—
[27]	AES-CCM	Virtex4 XC4VLX40	1995 0	1200 18	152	1.951 *	0.635	—

‡ Max. Frequency obtained with synthesis results. * Parallel AES implementation.

7. Conclusions

In this paper, we introduced the implementation of the ChaCha20 and Poly1305 hardware primitives in addition to a compatible ChaCha20–Poly1305 AEAD construction with TLS 1.3. The proposed AEAD implementation eliminates the faults introduced by a fragmented block. In addition, the function of the length of the plaintext and the AAD are supported. The accumulator and filter reduce the number of transactions in the bus, increasing 85.76% of the performance in the system implemented with a 14.14% area overhead, using the same system in the previous work. In comparison with the related works, the performance of the AEAD is 1.4 and 11.56-cycles/byte in standalone and system implementations, increasing the performance by 1517% in the standalone implementation. Furthermore, the resources occupied a 10808-LUT and 3731-FF in FPGA implementation with an overhead of 75.08%. The performance reduction for integrating the AEAD into an RISC-V environment using a Tile Link bus is 70.32% compared to the standalone implementation. Finally, the ChaCha20–Poly1305 implementation is compared with another AEAD used in TLS 1.3, demonstrating the competitive performance resources with the AES-based cipher suites.

Author Contributions: Supervision, C.-K.P., C.D., and T.-T.H.; methodology, R.S. and M.S.; investigation, R.S.; writing—original draft preparation, R.S.; writing—review and editing, R.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the New Energy and Industrial Technology Development Organization (NEDO) project JPNP16007.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: No new data were created or analyzed in this study. Data sharing is not applicable to this article.

Acknowledgments: This paper is based on results obtained from project JPNP16007, commissioned by the New Energy and Industrial Technology Development Organization (NEDO).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Lim, J.P.; Nagarakatte, S. Automatic Equivalence Checking for Assembly Implementations of Cryptography Libraries. In Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Washington, DC, USA, 16–20 February 2019; pp. 37–49.
2. Saraiva, D.A.F.; Leithardt, V.R.Q.; de Paula, D.; Mendes, A.S.; González, G.V.; Crocker, P. PRISEC: Comparison of Symmetric Key Algorithms for IoT Devices. *Sensors* **2019**, *19*, 4312. [CrossRef] [PubMed]
3. Najm, Z.; Jap, D.; Jungk, B.; Picek, S.; Bhasin, S. On Comparing Side-channel Properties of AES and ChaCha20 on Microcontrollers. In Proceedings of the IEEE Asia Pacific Conference on Circuits and Systems (APCCAS), Chengdu, China, 26–30 October 2018; pp. 552–555.
4. Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018. Available online: https://datatracker.ietf.org/doc/html/rfc8446 (accessed on 10 June 2022).
5. Almeida, J.B.; Barbosa, M.; Barthe, G.; Grégoire, B.; Koutsos, A.; Laporte, V.; Oliveira, T.; Strub, P.-Y. The Last Mile: High-Assurance and High-Speed Cryptographic Implementations. In Proceedings of the IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 18–21 May 2020; pp. 965–982.
6. De Santis, F.; Schauer, A.; Sigl, G. ChaCha20-Poly1305 Authenticated Encryption for High-speed Embedded IoT Applications. In Proceedings of the Design, Automation & Test in Europe Conference Exhibition (DATE), Lausanne, Switzerland, 27–31 March 2017; pp. 692–697.
7. Jungk, B.; Bhasin, S. Do not Fall Into a Trap: Physical Side-channel Analysis of ChaCha20-Poly1305. In Proceedings of the Design, Automation & Test in Europe Conference Exhibition (DATE), Lausanne, Switzerland, 27–31 March 2017; pp. 1110–1115.
8. Lavaud, A.D.; Fournet, C.; Kohlweiss, M.; Protzenko, J.; Rastogi, A.; Swamy, N.; Beguelin, S.Z.; Bhargavan, K.; Pan, J.; Zinzindohoue, J.K. Implementing and Proving the TLS 1.3 Record Layer. In Proceedings of the IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–26 May 2017; pp. 463–482.
9. Islam, M.M.; Paul, S.; Haque, M.M. Reducing Network Overhead of IoT DTLS Protocol Employing ChaCha20 and Poly1305. In Proceedings of the International Conference of Computer and Information Technology (ICCIT), Dhaka, Bangladesh, 22–24 December 2017; pp. 1–7.
10. Barthe, G.; Cauligi, S.; Grégoire, B.; Koutsos, A.; Liao, K.; Oliveira, T.; Priya, S.; Rezk, T.; Schwabe, P. High-Assurance Cryptography in the Spectre Era. In Proceedings of the IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 24–27 May 2021; pp. 1884–1901.
11. Sadio, O.; Ngom, I.; Lishou, C. Lightweight Security Scheme for MQTT/MQTT-SN Protocol. In Proceedings of the International Conference on Internet of Things: Systems, Management and Security (IOTSMS), Granada, Spain, 22–25 October 2019; pp. 119–123.
12. Aamir, M.; Sharma, S.; Grover, A. ChaCha20-in-Memory for Side-Channel Resistance in IoT Edge-Node Devices. *IEEE Open J. Circ. Syst.* **2021**, *2*, 833–842. [CrossRef]
13. Pfau, J.; Reuter, M.; Harbaum, T.; Hofmann, K.; Becker, J. A Hardware Perspective on the ChaCha Ciphers: Scalable Chacha8/12/20 Implementations Ranging from 476 Slices to Bitrates of 175 Gbit/s. In Proceedings of the IEEE International System-on-Chip Conference (SOCC), Singapore, 3–6 September 2019; pp. 294–299.
14. Henzen, L.; Carbognani, F.; Felber, N.; Fichtner, W. VLSI Hardware Evaluation of the Stream Ciphers Salsa20 and ChaCha, and the Compression Function Rumba. In Proceedings of the International Conference on Signals, Circuits and Systems (SCS), Monastir, Tunisia, 7–9 November 2008; pp. 1–5.
15. Kermani, M.M.; Azarderakhsh, R.; Aghaie, A. Fault Detection Architectures for Post-Quantum Cryptographic Stateless Hash-Based Secure Signatures Benchmarked on ASIC. *ACM Trans. Embed. Comput. Syst.* **2017**, *16*, 1–19. [CrossRef]
16. Kanda, G.; Ryoo, K. High-Throughput Low-Area Hardware Design of Authenticated Encryption with Associated Data Cryptosystem that Uses ChaCha20 and Poly1305. *Int. J. Recent Technol. Eng.* **2019**, *8*, 86–94.
17. Rambus Inc. Cipher Accelerators: CHACHA-IP-13 ChaCha20 Accelerators, 2021. Available online: https://www.rambus.com/security/crypto-accelerator-hardware-cores/basic-crypto-blocks/chacha-ip-13/ (accessed on 10 June 2022).
18. Rambus Inc. Hash Accelerators: POLY-IP-53 Poly1305-based MAC Accelerators, 2021. Available online: https://www.rambus.com/security/crypto-accelerator-hardware-cores/basic-crypto-blocks/poly-ip-53/ (accessed on 10 June 2022).
19. SilexInsight. ChaCha20-Poly1305 AEAD Crypto Engine, 2021. Available online: https://www.silexinsight.com/products/security/chacha20-poly1305-ip-core/ (accessed on 10 June 2022).
20. Serrano, R.; Duran, C.; Hoang, T.-T.; Sarmiento, M.; Tsukamoto, A.; Suzaki, K.; Pham, C.-K. ChaCha20-Poly1305 Crypto Core Compatible with Transport Layer Security 1.3. In Proceedings of the International SoC Design Conference (ISOCC), Jeju Island, Korea, 6–9 October 2021; pp. 17–18.
21. Li, J.; Chen, R.; Su, J.; Huang, X.; Wang, X. ME-TLS: Middlebox-Enhanced TLS for Internet-of-Things Devices. *IEEE Internet Things J.* **2019**, *7*, 1216–1229. [CrossRef]
22. W3 Techs. Usage Statistics of Default Protocol Https for Websites, May 2022. Available online: https://w3techs.com/technologies/details/ce-httpsdefault (accessed on 10 June 2022).
23. Rescorla, E.; Dierks, T. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, August 2008. Available online: https://datatracker.ietf.org/doc/html/rfc5246 (accessed on 10 June 2022).

24. Rodríguez, M.; Astarloa, A.; Lázaro, J.; Bidarte, U.; Jiménez, J. System-on-Programmable-Chip AES-GCM implementation for wire-speed cryptography for SAS. In Proceedings of the Conference on Design of Circuits and Integrated Systems (DCIS), Lyon, France, 14–16 November 2018; pp. 1–6.
25. Koteshwara, S.; Das, A.; Parhi, K.K. Architecture Optimization and Performance Comparison of Nonce-Misuse-Resistant Authenticated Encryption Algorithms. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2019**, *27*, 1053–1066. [CrossRef]
26. Hoang, V.-P.; Phan, T.-T.-D.; Dao, V.-L.; Pham, C.-K. A compact, ultra-low power AES-CCM IP core for wireless body area networks. In Proceedings of the International Conference on Very Large Scale Integration (VLSI-SoC), Tallinn, Estonia, 26–28 September 2016; pp. 1–4.
27. Badillo, I.A.; Uribe, C.F.; Cumplido, R.; Sandoval, M.M. FPGA Implementation and Performance Evaluation of AES-CCM Cores for Wireless Networks. In Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig), Cancun, Mexico, 3–5 December 2008; pp. 421–426.
28. Nir, Y.; Langley, A. ChaCha20 and Poly1305 for IETF Protocols. RFC 8439, June 2018. Available online: https://datatracker.ietf.org/doc/html/rfc8439 (accessed on 10 June 2022).
29. Bernstein, D.J. The Salsa20 Family of Stream Ciphers. In *New Stream Cipher Designs: The eSTREAM Finalists*; Springer: Berlin/Heidelberg, Germany, 2008; pp. 84–97.
30. Hoang, T.-T.; Duran, C.; Serrano, R.; Sarmiento, M.; Nguyen, K.-D.; Tsukamoto, A.; Suzaki, K.; Pham, C.-K. Trusted Execution Environment Hardware by Isolated Heterogeneous Architecture for Key Scheduling. *IEEE Access* **2022**, *10*, 46014–46027. [CrossRef]
31. RISC-V Foundation. Rocket Chip Generator, 2019. Available online: https://github.com/chipsalliance/rocket-chip (accessed on 10 June 2022).
32. SiFive, Inc. SiFive TileLink Specification, August 2019. Available online: https://static.dev.sifive.com/docs/tilelink/tilelink-spec-1.7-draft.pdf (accessed on 10 June 2022).
33. ARM. AMBA AXI and ACE Protocol Specification; Jan. 2021. Available online: https://developer.arm.com/documentation/ihi0022/hc?lang=en (accessed on 10 June 2022).