Regular Article

Implementation of ChaCha20-Poly1305 on Self-Organization Data Framing for Enhancing IoT Communication

Vu-Minh-Thanh Nguyen and Duc-Hung Le

Faculty of Electronics and Telecommunications, The University of Science, Vietnam National University, Ho Chi Minh City, Vietnam

Correspondence: Duc-Hung Le, ldhung@hcmus.edu.vn

Communication: received 14 March 2024, revised 25 July 2024, accepted 13 October 2024
Online publication: 23 October 2024, Digital Object Identifier: 10.21553/rev-jec.368

Abstract– **Our study presents an implementation of ChaCha20-Poly1305 on a self-organizing data framing on enhancing Internet of Things (IoT) communication. This enhancement is to apply manageability, security, and patternability to bare-metal IoT communication. Regarding handling communication data, data framing is designed with a manageable and patternable frame structure. Based on the data framing, the frame structure is established with additional fields for management, identification, and error detection to prevent malfunctions in IoT system. Furthermore, the main data package is encrypted by the combination of two stand-alone lightweight algorithms ChaCha20 and Poly1305 founding comprehensive Authenticated Encryption with Associated Data (AEAD). Based on the data framing, a data exchange scheme based on Virtual Register Management is patterned for manageability and customizability of processing perception data of a device. Regarding processing a frame, a frame parsing process is constructed and applied to node MCUs and the IoT cloud to ensure identification and avoid malfunction in implementation of frame detection. On ARM Cortex-M4 device without AES-NI accelerator, the ChaCha20-Poly1305 AEAD scheme performs better in terms of runtime and stack size, compared to AES-based AEAD methods in software-only implementation. Finally, the data exchange scheme using Virtual Register Management is applied rationally to manage perception data of a specific IoT system monitoring Base Transceiver Stations.**

Keywords– **IoT security, lightweight cryptography, data framing, data exchange scheme.**

1 Introduction

Internet of Things (IoT) networks have received a lot of attention globally in this digital age. These networks contain an enormous quantity of IoT devices. An IoT device, or node, is often made up of devices with limited resources. IoT devices' connections are extremely delicate and susceptible. Many security solutions are made to offer confidentiality, authentication, and reliability in order to protect IoT connections [1]. IoT devices based on Microcontroller Unit (MCU), which have limited resources, can benefit greatly from the security solutions offered by Lightweight Cryptography (LWC). Authenticated Encryption with Associated Data (AEAD) is a complete solution designed for devices with limited resources. Modern high-speed message authentication codes like Poly1305-AES are appropriate for a broad range of uses [2].

When it comes to hardware acceleration, AES is extremely quick; yet, when it comes to software implementation, it is slower and more inefficient in energy usage [3] [4]. High-speed cipher ChaCha20 is considered to be quicker than AES in software-only implementations [5] [6]. ChaCha20-Poly1305 AEAD can offer a complete security solution and better performance for devices/MCU with limited resources and without AES-NI support [7] [8]. ChaCha20-Poly1305 can guarantee secure communications between nodes and the cloud for IoT systems. To ensure sustainable operations and synchronization, it is important to improve local node interactions in addition to enabling secure node-cloud connections. The Universal Asynchronous Receiver-Transmitter (UART) can be used to build embedded IoT device interaction and is useful for making communication [9]. Flexibility, affordability, and high performance can be achieved with UART-based logic systems [10]. However, in order to identify communication and synchronization issues, UART-based transactions must be improved by adding extra field, checksum, or Cyclic Redundancy Check (CRC) bits [11]. For embedded devices, data framing can enhance UART communication [12].

In our proposal, to improve UART-based communication in an IoT node, the main data package is wrapped with additional fields of headers, trailers, and Cyclic Redundancy Check (CRC) for identification, management, and error detection to prevent malfunctions between MCUs and also the whole IoT system. Regarding implementations of ChaCha20-Poly1305 AEAD, the ChaCha20 is simple with Addition-Rotation-XOR (ARX) operations and easy to be manually implemented on MCUs or IoT cloud. By contrast, to handle a complicated 130-bit multiplication of Poly1305, a 130-bit modular multiplication is designed by using limb encoding based on Comba method [13].

Based on the data framing, a data exchange scheme using Virtual Register Management is constructed by reformating data bytes with ID fields to organize an appropriate and customizable casting method for different perception values of a device. On communication implementation between an IoT cloud and a node of STM32 and ESP32, the frame parsing process performs precisely to guarantee identification and prevent malfunction on frame detection. On ARM Cortex-M4 device without AES-NI accelerator, our ChaCha20-Poly1305 performs more efficiently in terms of runtime and stack size, compared to AES-based AEAD schemes in software-only implementation. In a specific IoT implementation of monitoring Base Transceiver Stations (BTSs), a data exchange scheme using Virtual Register Management is established rationally to manage perception data with customization. All things considered, our work proposes using the ChaCha20-Poly1305 AEAD in conjunction with a self-organization data framing to improve bare-metal node-side and node-cloud communication for manageability, patternability, and security in constructing data exchange schemes.

2 The ChaCha20 Algorithm

The ChaCha20, a family of stream ciphers, was developed by D. J. Bernstein from the Salsa20. The ChaCha20 encryption is made up from the quarter round and block function.

Fundamentally, the ChaCha20 quarter round is built on operations of four 32-bit integers denoted by the letters a, b, c, and d, as follows, in C-style notation:

$$\begin{cases} a += b; d \wedge= a; d <<<= 16 \\ c += d; b \wedge= c; b <<<= 12 \\ a += b; d \wedge= a; d <<<= 8 \\ c += d; b \wedge= c; b <<<= 7 \end{cases}$$

The quarter round consists of four main operating groups. In terms of pattern, the first and third groups are similar to one another. They only differ at the left rotation value; the second and fourth groups also exhibit this trait. The ChaCha20 quarter round is applied to the ChaCha20 state. There are 16 32-bit integer numbers in the ChaCha20 state. Each 4 of them are handled by a quarter-round process.

As seen in Algorithm 1, lines 5–12, the ChaCha20 block function applies several quarter-round operations on a ChaCha state. The block's inputs consist of a 256-bit key K, 96-bit nonce N, and 32-bit block counter CNT. The block produces a 64-byte key stream as its output. The ChaCha20 state is initialized with the inputs as a 4x4 matrix-patterned block with 16 32-bit integer integers. A 128-bit constant, denoted as $CONST$ makes up the first four words, which are, respectively, 0x61707865, 0x3320646e, 0x79622d32, and 0x6b206574. The 256-bit key K is represented by the next 8 words. Comparatively, the final 4 words are a 32-bit counter CNT and a 96-bit nonce N. After that, the ChaCha20 process runs 20 rounds in which it applies "diagonal rounds" and "column rounds" to the ChaCha20 state in turn. The "diagonal rounds" and "column rounds" in Algorithm 1 essentially function from lines 9–12 and 5-8. The ChaCha20 state input is added to the key stream output following 20 quarter cycles.

Algorithm 1 The ChaCha20 Encryption Algorithm
C = chacha20_encrypt_block(K, CNT, N, P)

```
INPUT: $K \in \{0,1\}^{256}, CNT \in \{0,1\}^{32}, N \in \{0,1\}^{96}, P \in \{0,1\}^{*}$
OUPUT: $C \in \{0,1\}^{|P|}$
 1: for i = 0 upto $\lfloor L(P)/512 \rfloor - 1$ do
 2:     $S \leftarrow CONS \,||\, KEY \,||\, CNT + i \,||\, N$
 3:     $S_{temp} \leftarrow S$
 4:     for j = 0 upto 10 do
 5:         Qround$(S_{temp}, 0, 4, 8, 12)$
 6:         Qround$(S_{temp}, 1, 5, 9, 13)$
 7:         Qround$(S_{temp}, 2, 6, 10, 14)$
 8:         Qround$(S_{temp}, 3, 7, 11, 15)$
 9:         Qround$(S_{temp}, 0, 5, 10, 15)$
10:         Qround$(S_{temp}, 1, 6, 11, 12)$
11:         Qround$(S_{temp}, 2, 7, 8, 13)$
12:         Qround$(S_{temp}, 3, 4, 9, 14)$
13:     end for
14:     $S += S_{temp}$
15:     $C[i*512..(i*512+511)] = P[i*512..(i*512+511)] \wedge S$
16: end for
17: return $C$
```

The ChaCha20 uses keystreams generated from the ChaCha20 block to encrypt the plaintext. The ChaCha20 encryption is divided into 512-bit sections, based on the plaintext length, for efficient programming, which optimizes memory by using Algorithm 1 at the indexes at line 1 of the for loop. The ChaCha20 block function provides a 512-bit keystream with the same key K, nonce N, and progressively increasing block counter parameters (CNT + i) in each segment. After that, a 512-bit ciphertext section is computed by XORing the relevant plaintext and 512-bit keystream sections for each index i. Ultimately, an encrypted message of the same plaintext length is provided by the ChaCha20 encryption. The encryption and decryption processes are similar. Keystreams are generated by the decryption and XORed with the ciphertext to recover the plaintext.

3 The Poly1305 Algorithm

Poly1305 is a one-time authenticator designed by D. J. Bernstein. Poly1305 generates a 16-byte tag from a 32-byte one-time key and the message. This tag is used to authenticate the message [6].

The one-time key is partitioned into 16-byte "r" and "s" parts respectively. Before being used, the Poly1305 process first clamps the "r". Clamping "r" is to ensure that the top four bits of $r[3]$, $r[7]$, $r[11]$, and $r[15]$ and the bottom two bits of $r[4]$, $r[8]$, and $r[12]$ is cleared. The Poly1305 takes an input set of a 256-bit one-time key and an arbitrary-lengthed message. From the original article of Poly1305 [2] and Santis et al [14], the Poly1305 tag is calculated as follows:

$$T = \left(\left(\sum_{i=0}^{q} m_i \bar{r}^{q-i+1} \bmod p \right) + s \right) \bmod 2^{128}, \quad (1)$$

where T is the 16-byte authentication tag, $(m_i)_{1\leq i\leq q}$ is a 17-byte padded message chunk containing a 16-byte message section padded with a 1-byte 0x01. With a l-byte message, the message is chopped into $q = \lceil l/16 \rceil$ sections. $\bar{r}$ is the result of clamping r. p is the constant prime $2^{130} - 5$. s is the secret key, the 16-byte big-endian partition of the input one-time key K. In the programming approach, the Poly1305 process is implemented following Algorithm 2. The sum of products of the Equation 1 is handled in the for loop, from lines 6-10, based on Horner's method. From Santis et al [14], the additions and multiplications of Poly1305's tag generator is also presented as follows

$$acc = ((\cdots(m_1\bar{r} + m_2)\bar{r} + \cdots + m_{q-1})\bar{r} + m_q)\bar{r} \bmod p. \tag{2}$$

This optimization method performs without computing complicated exponent operations for $\bar{r}$.

In ChaCha20-Poly1305 implementation, the 256-bit session key specifically for the Message Authentication Code (MAC) to form such a key pair (r, s) is generated by the ChaCha20 block function. The block function is called with the following parameters: a 256-bit session integrity key considered as the ChaCha20 key, a block counter set to zero, and a 96-bit nonce. A 512-bit state is then generated. The first 256 bits are used to respectively partition a 16-byte r and s. The rest 256 bits are discarded.

Algorithm 2 The Poly1305 Algorithm
T = poly1305_mac(M, K)

```
INPUT: $M \in \{0,1\}^*, K \in \{0,1\}^{256}$
OUPUT: $T \in \{0,1\}^{128}$
 1: $r \leftarrow K[0..15]$
 2: $\bar{r} \leftarrow clamp(r)$
 3: $s \leftarrow K[16..31]$
 4: $acc \leftarrow 0$
 5: $p \leftarrow (1 \ll 130) - 5$
 6: for i = 1 upto $\lceil L(M)/16 \rceil$ do
 7:     $n \leftarrow M[(i-1)*16)..(i*16)] \,||\, 0x01$
 8:     $acc \mathrel{+}= n$
 9:     $acc \leftarrow (\bar{r} * acc) \bmod p$
10: end for
11: $acc \mathrel{+}= s$
12: $T \leftarrow acc[0..16]$
13: return $T$
```

4 The ChaCha20-Poly1305 Construction

ChaCha20-Poly1305 is an Authenticated Encryption with Associated Data (AEAD) algorithm [6].

The ChaCha20-Poly1305 function computes a 128-bit tag T and ciphertext from a 256-bit key K, 96-bit nonce N, an arbitrary-length plaintext, and Arbitrary length additional authenticated data (AAD):

$$\begin{aligned} \text{CC20\_P1305} : \{0,1\}^{256} \times \{0,1\}^{96} \\ \times(\{0,1\}^*)^2 \rightarrow \{0,1\}^* \times \{0,1\}^{128}, \end{aligned} \tag{3}$$

such that $(K, N, A, P) \mapsto (C, T)$ (illustrated in Algorithm 3). Firstly, with the 256-bit key and 96-bit nonce, the ChaCha20-Poly1305 function generates a Poly1305 one-time key from the ChaCha20 block function. The ChaCha20 encryption function then encrypts the plaintext with the same key and nonce. Finally, the Poly1305 function with the generated one-time key computes a 16-byte authentication tag from a message concatenation of padded AAD, padded ciphertext, a 64-bit casted length of AAD, and a 64-bit casted length of ciphertext. Based on Santis et al [14], the padded AAD and ciphertext is provided from the function pad16 : $\{0,1\}^{8l} \rightarrow \{0,1\}^{8(l+\delta)}$ that pads $\delta = (16 - l) \bmod 16$ zero bytes to a l-byte input. The decryption resembles the encryption process. However, the roles of ciphertext and plaintext are reversed in the ChaCha20 encryption function. The ChaCha20 function is applied to the ciphertext to provide the plaintext.

Algorithm 3 AEAD Construction
(C, T) = CC20_P1305(K, N, A, P)

```
INPUT: $K \in \{0,1\}^{256}, N \in \{0,1\}^{96}, A \in \{0,1\}^*, P \in \{0,1\}^*$
OUPUT: $C \in \{0,1\}^{|P|}, T \in \{0,1\}^{128}$
1: $OTK \leftarrow$ poly1305_key_gen$(K, N)$          {One-time Key}
2: $C \leftarrow$ chacha20_encrypt$(K, 1, N, P)$        {Ciphertext}
3: $M \leftarrow$ pad16$(A)$ | pad16$(C)$ | $|L(A)|$ | $|L(C)|$ {MAC Data}
4: $T \leftarrow$ poly1305_mac$(M, OTK)$             {Poly1305 Tag}
5: return $C, T$
```

5 Proposed Self-Organization Data Framing

To integrate on a serial protocol, data framing is applied on an IoT node constructed from a combination of two MCUs with built-in serial protocol modules such as UART. STM32 is appropriate to perform on-node computations due to its wide range of family, energy efficiency, and various capacities for IoT applications or purposes [15] [16]. Among STM32 devices and families, STM32 devices without AES-NI accelerator are selected to implement ChaCha20-Poly1305 in software-only environment. Since different STM32 devices are suitably chosen for distinct usages, no specific STM32 device or family is pointed out in this proposal. ESP32 is an IoT device with cost efficiency, low power consumption, and capabilities of both serial and wireless communication [17]. From this point and in conjunction with an STM32 device without built-in Wi-Fi module, ESP32 is suitable to be an IoT on-node forwarder or gateway. With the two MCUs with built-in UART hardware, these MCUs are used to construct a physical node to integrate the data framing. This MCU combination is our specific use case to establish the data framing on a physical IoT node, other combinations with distinct serial protocols are still able to use this framing technique. For an MCU, UART is a good low-cost and low-power solution. However, UART connections may give rise to problems like clock drift and temporal predictability [18]. To improve UART connection, additional fields are added to the data frame. Partitions that are needed to determine the start and stop of the frame, compute the Cyclic Redundancy Check (CRC), and manage data should be provided by the frame [12]. The frame structure is built using the following components to meet these needs: $CRCs$, $HEADERS$, $BODY$, and $TRAILERS$ (shown in Figure 1). Two bytes, $H1$ and $H2$, are present in the $HEADERS$ field to identify the frame start. The 2-byte "Data Length", the 1-byte "Command", and the data bytes are concatenated to form the $BODY$ field. The two bytes $T1$ and $T2$, which indicate the frame end, are contained in the $TRAILERS$ field. Lastly, the two bytes $CRC1$ and $CRC2$ for the 16-bit CRC calculation are contained in the $CRCs$ field.

The frame protocol facilitates node-cloud interactions via the Message Queueing Telemetry Transport (MQTT) protocol as well as communication between a node's MCU via the UART protocol. A node's STM32 and the IoT cloud engage in frame transactions in an intermediary way through ESP32, to which the ChaCha20-Poly1305 AEAD is implemented. The data bytes in a frame are subjected to the ChaCha20-Poly1305 AEAD. To create an encrypted frame (shown in Figure 2), the AEAD algorithm computes a 16-byte authentication tag and encrypted data bytes $(c[1], c[2], ...c[n])$. A decryption process on a frame is designed to decrypt data bytes and utilize the 16-byte tag for authentication.

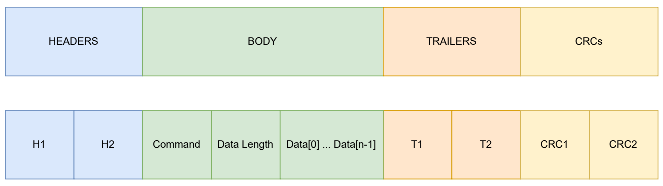


Figure 1. The frame structure of the IoT system.

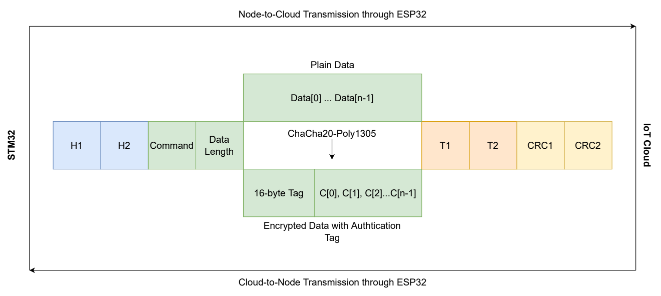

Node-to-Cloud Transmission through ESP32
Plain Data
Data[0] ... Data[n-1]
STM32
H1
H2
Command
Data Length
ChaCha20-Poly1305
T1
T2
CRC1
CRC2
IoT Cloud
16-byte Tag
C[0], C[1], C[2]...C[n-1]
Encrypted Data with Authication Tag
Cloud-to-Node Transmission through ESP32

Figure 2. The ChaCha20-Poly1305 Implementation on the Proposed Frame Protocol.

The ESP32 establishes a fixed-length circular buffer to temporarily store transaction data in order to buffer transmission data. Data from the STM32 is cached via UART, while data from the IoT cloud is cached via MQTT, using two circular buffers. The design of the arbitration block is to decide how to adjudge the buffers in the firmware of the MCU's endless loop (Figure 3). In every infinite-loop period, the arbitrator alternately indexes the buffers. A suitable circular buffer is regularly handled and inspected. Another independent process is where frame parsing is initiated (Figure 4). When a successful, failed, or timeout event is raised, the parsing procedure ends.

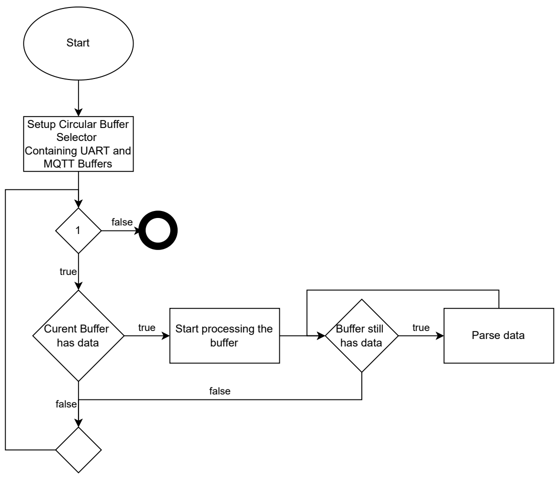

Start
Setup Circular Buffer Selector Containing UART and MQTT Buffers
1
false
true
Curent Buffer has data
true
Start processing the buffer
Buffer still has data
true
Parse data
false
false

Figure 3. Arbitration Process of Circular Buffers.

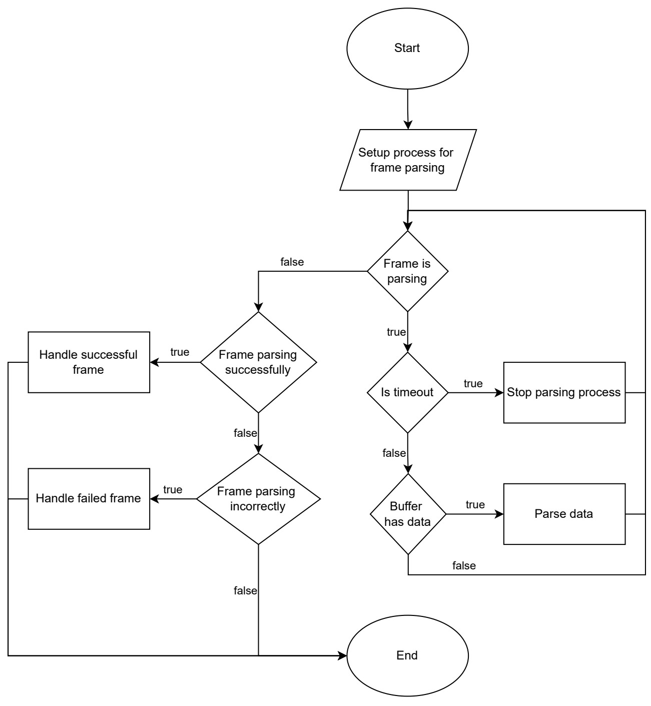


Figure 4. Frame Parsing Process.

Based on the frame structure, we propose a scheme to exchange perception data (Temperature, Humidity, Relay Status, etc.) of a device with typical data types (Integer, Real Number, and String). To consider perception data exchange, the perception data exchange is specified by command ID in a command list of an IoT system. In the frame structure, a command ID of perception data exchange is exactly provided at the command field. At the start of the data bytes, an arbitrary-sized `Device ID` is positioned for device identification. In the next position, to indicate the corresponding casting method, a 1-byte ID, which is defined as **Virtual Register ID** in this proposal, is inserted to detect an appropriate procedure (illustrated in Figure 5). In this work, this method using Virtual Register ID is called **Virtual Register Management** and explained as follows. With a 1-byte ID, this management method is able to handle 256 Virtual Registers corresponding to 256 perception values of a device, and each IoT system should limit an appropriate number of Virtual Registers suitable for every resource-constrained device. Regarding managing received perception data for one device, Virtual Register ID specifies a procedure indicating how the data bytes are cast to store appropriately in Integer, Real Number, or String in physical registers of a computer device. With each Virtual Register ID of a device, a default procedure is preset on computer devices and able to be customized. At C-based STM32 devices, the weak functions are used to predefined default procedures of each Virtual Register ID. This allows a default procedure to be overwritten to custom processing of corresponding perception data. At the IoT cloud, a perception value is only cast as an appropriate data type for a perception value based on its preset data type along with Virtual Register ID. Hence, there is currently no room for casting customization on the cloud side. In further, flow-based visual programming such as Node-RED can be applied to provide customization of perception data procedure in the IoT cloud.

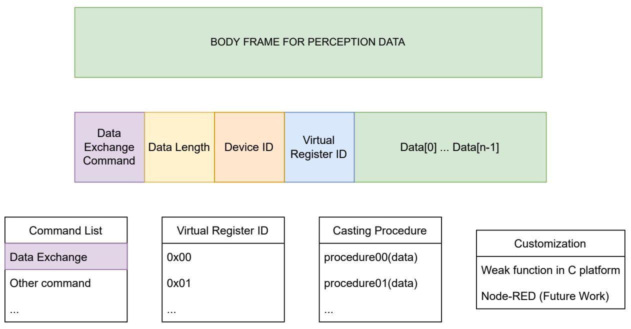


Figure 5. The Proposed Perception Data Exchange Scheme of a Device based on Virtual Register Management.

6 Implementation Results and Discussions

6.1 Implementation of ChaCha20-Poly1305 and Data Framing on a Specific IoT System

The ChaCha20-Poly1305 AEAD and data framing are applied on an IoT system monitoring and controlling two air conditioners of BTSs (illustrated in Figure 6). The system consists of client-side, cloud, and node components. A node is built with an ESP32 and an STM32. The STM32 is used to acquire data from sensors (temperature, current, and voltage) and to directly control the status of two relays that switch on and off air conditioners. Our specific STM32 device of a node of this application is STM32F103C8T6. The ESP32 is positioned to transport data frames between the cloud and STM32. A Virtual Private Server (VPS) is used to serve cloud apps. Server applications are constructed based on a web server and back-end system. The web application is used to give Graphical User Interface (GUI) on the client side. The data framing is positioned on local-node communication. Transaction data is covered by extra fields for error detection, synchronization, and UART improvement across the UART protocol between MCUs. The ChaCha20-Poly1305 is placed on communication of STM32 and the IoT cloud in an intermediary way through ESP32. As a result, in the Internet environment, authentication and encryption are applied to data of the Internet of Things system together with the suggested data frame. For an STM32 device, the Virtual Register Management is applied to handle 7 perception values from sensors and actuators (illustrated in table I). Finally, the data types Integer, Real Numer, and String in C-based platform of an STM32 device and our NodeJS-based IoT cloud are specified in Table II.

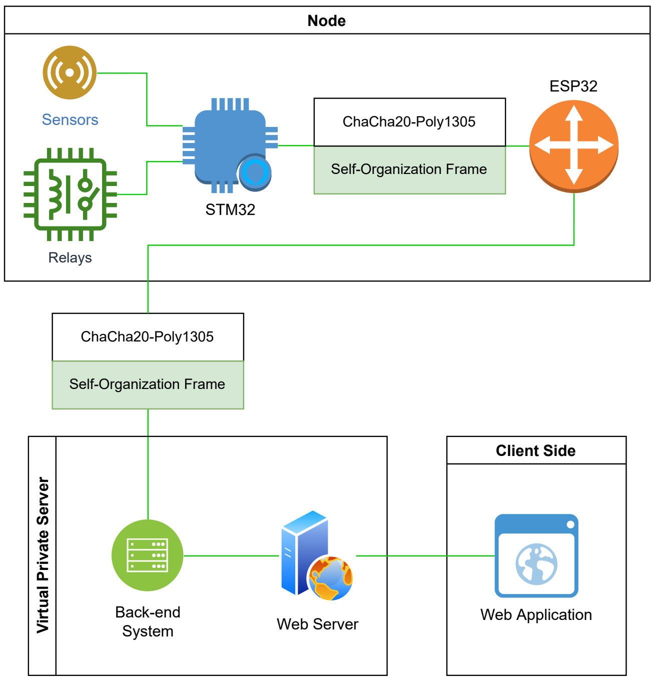


Figure 6. Implementation of ChaCha20-Poly1305 and Data Framing on the IoT System.

Table I
Organization of Virtual Register Management of an STM32 device for BTS Management

Register Name	**Register ID**	**Data Type**	**Description**
Temperature	0x00	Real	Room temperature of the BTS
Current 1	0x01	Real	Current of 1st AC unit
Current 2	0x02	Real	Current of 2nd AC unit
Voltage 1	0x03	Real	Voltage of 1st AC unit
Voltage 2	0x04	Real	Voltage of 2nd AC unit
Relay 1	0x05	Integer	On/Off of 1st AC unit
Relay 2	0x06	Integer	On/Off of 2nd AC unit

Table II
Data Type Specifications on C-based STM32 Devices and the NodeJS-based IoT cloud

Data Type	**C-based STM32 Device**	**NodeJS-based IoT Cloud**
Integer	`int32_t`	Element of `Int32Array`
Real	`float`	Element of `Float32Array`
String	`char` array	`String`

6.2 Data Framing and Security

6.2.1 Frame Parsing: The correctness of the frame parsing process is evaluated by a sample perception data exchange between an STM32 and the IoT cloud in an intermediate way through an ESP32 (illustrated in Figure 7). To start the data exchange process, the STM32 sent a frame containing 71-byte data. The 71-byte data includes 1-byte Virtual Register ID, 4-byte perception data, 16-byte ChaCha20-Poly1305 tag, and an additional 50-byte device identification for database query. In this test case, on receiving a frame of data exchange, the IoT cloud forwards the frame back to STM32 to trigger the frame parsing process on the STM32 device. Through logging, each stage of the data framing process is printed out to supervise states of detecting headers, trailers, and CRC codes. On STM32, millisecond timing is obtained by using the 1-KHz counter of the default system timer, and microsecond timing is measured by using a custom 1-MHz counter of another built-in timer peripheral. On ESP32, the microsecond timing is based on `micros` function of Arduino framework. On the NodeJS-based IoT cloud, millisecond timing is measured from `performance.now` API of NodeJS framework, and microsecond timing is acquired by `timespec_get` function of a `C` program triggered by using `child-process` module. Finally, the frame parsing is correctly implemented based on the algorithm demonstrated in Figure 4.

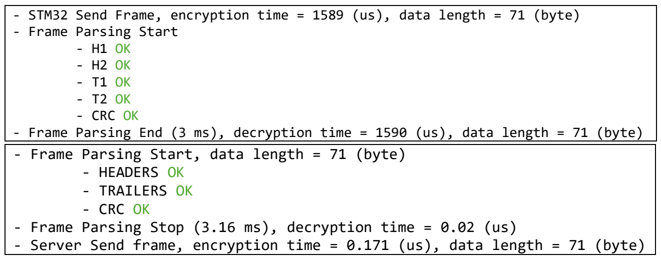
6.2.2 Accuracy of ChaCha20-Poly1305 Processes: The ChaCha20-Poly1305 has been successfully implemented in STM32 and the IoT cloud. According to the AEAD construction of the ChaCha20-Poly1305, the encryption process (illustrated in Figure 8) has precisely performed according to the RFC 8439 [6]'s order:

1) Initialize state for Poly1305 one-time key.
2) Generate Poly1305 one-time key.
3) Execute ChaCha20 encryption function.
4) Execute Poly1305 function for ciphertext.

The decryption process of the ChaCha20-Poly1305 AEAD (illustrated in Figure 9) has also correctly operated according to the RFC 8439 [6]'s order:

1) Initialize state for Poly1305 one-time key.
2) Generate Poly1305 one-time key.
3) Execute Poly1305 function for ciphertext.
4) Execute ChaCha20 decryption function.

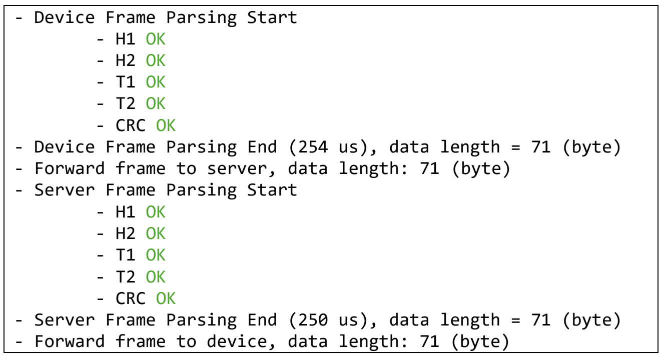
According to a report of KDDI Research [19], there is no weakness found in ChaCha20-Poly1305 AEAD. In addition, Procter [20] proved that the ChaCha20-Poly1305 AEAD is secure if both ChaCha20 and Poly1305 are secure. Salsa20, the origination of ChaCha20, is evaluated with significant security reviews on Differential attacks, Algebraic attacks,... [21] Regarding other variations, attacks on ChaCha6/7/12 are made by Aumasson et al [22], Shi et al [23], Maitra [24], and Choudhuri [25]. However, according to KDDI Research, among several cryptanalyses, Differential Analysis, Linear Cryptanalysis, Distinguishing Attack, Guess and Determine Analysis, Algebraic Attack, and Attacks on Initialization revealed no attack in ChaCha20. Additionally, time-memory-data tradeoff attack theoretically applies to the original version of ChaCha with 2^{140} time complexity using 2^{80} keystreams and 2^{160} memory size. For the IETF version of ChaCha, the time complexity is $2^{234.67}$ employing 2^{117} keystreams and 2^{176} memory size. However, based on practical assumption of KDDI Research, time-memory-data tradeoff attack is impossible to apply to ChaCha with time complexity less than 2^{256} by limiting keystream size to 2^{96}. Regarding the Poly1305, the KDDI Research found that Poly1305 is a ε-almost-Δ-universal where $\varepsilon = 8\lceil L/16 \rceil / 2^{106}$, which proved that Poly1305 is a secure hash function. Poly1305 forged messages are rejected with a probability of $1 - (n/2^{107})$, where n is the maximum length of the Poly1305 input. The maximum forgery probability of Poly1305 is roughly $1/2^{93}$ in (D)TLS [26].



```
- STM32 Send Frame, encryption time = 1589 (us), data length = 71 (byte)
- Frame Parsing Start
        - H1 OK
        - H2 OK
        - T1 OK
        - T2 OK
        - CRC OK
- Frame Parsing End (3 ms), decryption time = 1590 (us), data length = 71 (byte)
```

```
- Frame Parsing Start, data length = 71 (byte)
        - HEADERS OK
        - TRAILERS OK
        - CRC OK
- Frame Parsing Stop (3.16 ms), decryption time = 0.02 (us)
- Server Send frame, encryption time = 0.171 (us), data length = 71 (byte)
```

(a) Log Supervising of STM32 and the IoT cloud



```
- Device Frame Parsing Start
        - H1 OK
        - H2 OK
        - T1 OK
        - T2 OK
        - CRC OK
- Device Frame Parsing End (254 us), data length = 71 (byte)
- Forward frame to server, data length: 71 (byte)
- Server Frame Parsing Start
        - H1 OK
        - H2 OK
        - T1 OK
        - T2 OK
        - CRC OK
- Server Frame Parsing End (250 us), data length = 71 (byte)
- Forward frame to device, data length: 71 (byte)
```

(b) Log Supervising of ESP32

Figure 7. A sample perception data exchange of STM32 and the IoT cloud in an intermediary way through ESP32.

6.3 ChaCha20-Poly1305 Benchmarkings

ChaCha20-Poly1305 benchmarking is conducted on an STM32F407VGT6 board's ARM Cortex-M4 core. With an ARM Cortex-M device, all metrics of the ChaCha20-Poly1305 performance can be obtained based on using an available framework, Cifra [27].

```
- ChaCha20-Poly1305 Encryption Start
        - Set-up of Poly1305 one-time key:
                61707865 3320646e 79622d32 6b206574
                3c78f0e1 f0e10f1e 0f1e3c78 3c78f0e1
                53a60f1e 254a9429 c993264c 5fbf3264
                00000000 43860c18 0e1c9021 20418307

        - Poly1305 one-time key:
                3d8653f0 ebba5876 5feca469 3f911b29
                6444430e 31ed471d 27a666c3 027c1891

        - ChaCha20 Encryption State:
                61707865 3320646e 79622d32 6b206574
                3c78f0e1 f0e10f1e 0f1e3c78 3c78f0e1
                53a60f1e 254a9429 c993264c 5fbf3264
                00000001 43860c18 0e1c9021 20418307

        - Poly1305 function execution for ciphertext.
        - Poly1305 Tag:
                9c690631 ee8bff24 0fac3a90 6ad67d7d
- ChaCha20-Poly1305 Encryption End
```

(a) STM32

```
- ChaCha20-Poly1305 Encryption start
        - Set-up of Poly1305 one-time key:
                61707865 3320646e 79622d32 6b206574
                3c78f0e1 f0e10f1e 0f1e3c78 3c78f0e1
                53a60f1e 254a9429 c993264c 5fbf3264
                00000000 43860c18 0e1c9021 20418307
        - Poly1305 one-time key:
                3d8653f0 ebba5876 5feca469 3f911b29
                6444430e 31ed471d 27a666c3 027c1891
        - ChaCha20 Encryption State:
                61707865 3320646e 79622d32 6b206574
                3c78f0e1 f0e10f1e 0f1e3c78 3c78f0e1
                53a60f1e 254a9429 c993264c 5fbf3264
                00000001 43860c18 0e1c9021 20418307
        - Poly1305 function execution for ciphertext.
        - Poly1305 Tag:
                39980953 7f7c384d 5d5eb0bb 021003d9
- ChaCha20-Poly1305 process end
```

(b) The IoT cloud

Figure 8. Log Supervising of the ChaCha20-Poly1305 AEAD for encryption of STM32 and the IoT cloud.

```
- ChaCha20-Poly1305 Decryption Start
        - Set-up of Poly1305 one-time key:
                61707865 3320646e 79622d32 6b206574
                3c78f0e1 f0e10f1e 0f1e3c78 3c78f0e1
                53a60f1e 254a9429 c993264c 5fbf3264
                00000000 43860c18 0e1c9021 20418307

        - Poly1305 one-time key:
                3d8653f0 ebba5876 5feca469 3f911b29
                6444430e 31ed471d 27a666c3 027c1891

        - Poly1305 function execution for ciphertext.
        - ChaCha20 Decryption State:
                61707865 3320646e 79622d32 6b206574
                3c78f0e1 f0e10f1e 0f1e3c78 3c78f0e1
                53a60f1e 254a9429 c993264c 5fbf3264
                00000001 43860c18 0e1c9021 20418307

        - Poly1305 Tag:
                39980953 7f7c384d 5d5eb0bb 021003d9

        - DECRYPT SUCCESSFUL
- ChaCha20-Poly1305 Decryption End
```

(a) STM32

```
- ChaCha20-Poly1305 Decryption start
        - Set-up of Poly1305 one-time key:
                61707865 3320646e 79622d32 6b206574
                3c78f0e1 f0e10f1e 0f1e3c78 3c78f0e1
                53a60f1e 254a9429 c993264c 5fbf3264
                00000000 43860c18 0e1c9021 20418307
        - Poly1305 one-time key:
                3d8653f0 ebba5876 5feca469 3f911b29
                6444430e 31ed471d 27a666c3 027c1891
        - Poly1305 function execution for ciphertext.
        - ChaCha20 Decryption State:
                61707865 3320646e 79622d32 6b206574
                3c78f0e1 f0e10f1e 0f1e3c78 3c78f0e1
                53a60f1e 254a9429 c993264c 5fbf3264
                00000001 43860c18 0e1c9021 20418307
        - Poly1305 Tag:
                9c690631 ee8bff24 0fac3a90 6ad67d7d
        - Tag matched!
- ChaCha20-Poly1305 process end
```

(b) The IoT cloud

Figure 9. Log Supervising of the ChaCha20-Poly1305 AEAD for decryption of STM32 and the IoT cloud.

Based on the Cifra framework, the metrics of Runtime, Speed, Code Size, and Stack Size (illustrated in Table III) are performed to compare with other algorithms implemented on ARM and AVR Platforms. Since our works mainly focus on constructing secure and patternable communication of an IoT system, these metric comparisons are only used to indicate the acceptability of the performance of our implemented ChaCha20-Poly1305 algorithms compared to AES-based AEADs in software-only environments. The metric software was compiled by ARM-None-EABI-GCC (ARM GNU Toolchain) version `13.2.1` release `20231009` with flags `-O2 -mthumb -mcpu=cortex-m4`. The runtime is measured in cycles and obtained when applying 64-, 128-, and 16-byte messages for ChaCha20 encryption, Poly1305 MAC, and the whole AEAD processes respectively. The cycle is determined by counting the current Systick counter and its reset times after triggering a test function by using the default Systick settings. The speed of our implemented algorithms is measured when processing a 4096-byte message. The code size is calculated by subtracting the size of the text section of a blank test program from the size of each actual program. The stack size is measured by filling the stack memory with a secret pattern and checking how many patterns were overwritten after performing a function. Overall, our implemented AEAD process of ChaCha20-Poly1305 consumes a 2.6KB code size and 580-byte stack with a speed of 31.5 cycles/byte on ARM Cortex-M4. According to Table III, assembly-based optimization of De Santis et al [14] is the most efficient in both time and space. The method of Hülsing et al [28] performs the fastest speed of ChaCha20, while the code size is disproportionately large, at about 3 times bigger than our size. Regarding AEAD methods of AES on software-only implementation of the Cifra project, this performance may not be appropriate to implement in software-only environment of low-cost devices without AES accelerator. Compared to other AEAD methods, NORX32 is an AEAD design and its core is inspired by ChaCha20 [29]. NORX32 has a good optimization in code size and stack usage, yet the speed needs to be enhanced in comparison with the optimization methods of ChaCha20-Poly1305. Regarding MAC algorithms, Chaskey is a permutation-based MAC algorithm based on ARX construction. The speed and code-size performance of Chaskey in ARM Cortex-M4 is considerable and it is significantly faster than AES-128-CMAC in software-only implementation [30]. Finally, our AEAD implementations are considerably outperformed by assembly-based optimizations, yet our performance is better and more acceptable in speed and stack usage, compared to software-only AEAD implementations based on AES.

6.4 Implementation Techniques of ChaCha20 and Poly1305

The ChaCha20 core is the Quarter Round. On an MCU, ARX arithmetics are performed easily on 32-bit platforms. Therefore, in C implementation, the `ADD` and `XOR` are conducted normally as binary operators, while the left rotation `ROTL` is performed by using the "`_lrotl`" function of the `stdlib.h`. Regarding the Poly1305, because the Poly1305 prime is $2^{130} - 5$, the multiplication result may reach a large 130-bit number that isn't supported for calculation in the up-to-64-bit structure of the constrained ESP32 or STM32 device. To perform 130-bit modular multiplication, a 130-bit number is encoded into 5 26-bit limbs to apply Comba method [13]. The 26-bit 5-limb structure is illustrated in Table IV. With 2 26-bit limbs $(a[i], b[i])_{4 \geq i \geq 0}$, the multiplication of the limbs is a 52-bit number

$$a[i] \times b[i] = X \in \{0,1\}^{52}. \tag{4}$$

Due to the Poly1305 constant prime $p = 2^{130} - 5$, the multiplication result X will always be smaller than the p. In p modulo and condition $X \leq p$, a multiplication of 2^{130} is equaled to multiplying with 5

$$X * 2^{130} = (X \ mod \ p) * (2^{130} \ mod \ p) = X * 5. \tag{5}$$

For the $X \in \{0,1\}^{52}$, the multiplication of $(X \times 5)$ fits in a 55-bit value. In combination with the 4 sums of each $s[i]_{4 \geq i \geq 0}$, the final value fits in a 57-bit value, which is possibly handled in a 64-bit variable of an ESP32 or STM32.

The Algorithm 4 presents our Comba-based Modular Multiplication on a 32-bit MCU to perform Poly1305 multiplication. Carry propagation process is obtained as an optimized case when applying 2 maximum limb values of r and a 16-byte message as in Equation (6). Before triggering the modular multiplication process, 5-byte limbs of r are structured as follows each limb only contains 26 bits of the 128-bit r and ensures the clamping rule (mentioned in Section 3).

$$\begin{cases} r\_limb[0] = r[0..3] \ \& \ \texttt{0x3ffffff} \\ r\_limb[1] = (r[3..6] \gg 2) \ \& \ \texttt{0x3ffff03} \\ r\_limb[2] = (r[6..9] \gg 4) \ \& \ \texttt{0x3ffc0ff} \\ r\_limb[3] = (r[9..12] \gg 6) \ \& \ \texttt{0x3f03fff} \\ r\_limb[4] = (r[12..15] \gg 8) \ \& \ \texttt{0x00fffff} \end{cases} \tag{7}$$

Regarding the message, a 16-byte message is encoded into 5 limbs as follows

$$\begin{cases} m\_limb[0] = m[0..3] \ \& \ \texttt{0x3ffffff} \\ m\_limb[1] = (m[3..6] \gg 2) \ \& \ \texttt{0x3ffffff} \\ m\_limb[2] = (m[6..9] \gg 4) \ \& \ \texttt{0x3ffffff} \\ m\_limb[3] = (m[9..12] \gg 6) \ \& \ \texttt{0x3ffffff} \\ m\_limb[4] = (m[12..15] \gg 8) \ \& \ |(1 \ll 24) \end{cases} \tag{8}$$

Algorithm 4 130-bit Modular Multiplication of Poly1305 on 32-bit platforms.
Poly1305_Modular_Mul(h, r)

```
INPUT: $h \in \{0,1\}^{160} = (h0, h1, h2, h3, h4), r \in \{0,1\}^{160} = (r0, r1, r2, r3, r4)$
OUPUT: $h \in \{0,1\}^{160} = (h0, h1, h2, h3, h4)$
1: $s1 = r1 * 5$
2: $s2 = r2 * 5$
3: $s3 = r3 * 5$
4: $s4 = r4 * 5$
5: $d \in \{0,1\}^{320} = (d0, d1, d2, d3, d4)$
                                                      {h *= r}
6: $d0 \leftarrow h0 * r0 + h1 * s4 + h2 * s3 + h3 * s2 + h4 * s1$
7: $d1 \leftarrow h0 * r1 + h1 * r0 + h2 * s4 + h3 * s3 + h4 * s2$
8: $d2 \leftarrow h0 * r2 + h1 * r1 + h2 * r0 + h3 * s4 + h4 * s3$
9: $d3 \leftarrow h0 * r3 + h1 * r2 + h2 * r1 + h3 * r0 + h4 * s4$
10: $d4 \leftarrow h0 * r4 + h1 * r3 + h2 * r2 + h3 * r1 + h4 * r0$
                                          {carry propagation}
11: $carry \in \{0,1\}^{32} = 0$
12: $carry \leftarrow d0 \gg 26; h0 \leftarrow d0$ & 0x3ffffff
13: $d1$ += $carry; carry \leftarrow d1 \gg 26; h1 \leftarrow d1$ & 0x3ffffff
14: $d2$ += $carry; carry \leftarrow d2 \gg 26; h2 \leftarrow d2$ & 0x3ffffff
15: $d3$ += $carry; carry \leftarrow d3 \gg 26; h3 \leftarrow d3$ & 0x3ffffff
16: $d4$ += $carry; carry \leftarrow d4 \gg 26; h4 \leftarrow d4$ & 0x3ffffff
17: $h0$ += $carry * 5; h0$ &= 0x3ffffff
18: $h1$ += $carry$
19: $h \leftarrow (h0, h1, h2, h3, h4)$
20: return $h$
```

7 Conclusions

Our work has successfully constructed and implemented ChaCha20-Poly1305 scheme on data framing on a specific application of IoT communication. A bare-metal communication from this organization is able to integrate data exchange constructions with manageability, patternability, and security. Additional fields (headers, trailers, and CRCs) of the frame structure are capable of organizing IoT communication with identification and malfunction protection. On STM32 devices without AES-NI accelerator, the ChaCha20-Poly1305 combination ensures better performance in runtime and stack usage, compared to AES-based AEAD methods. Based on the patternable frame structure, a data exchange scheme employing Virtual Register Management is established with customizable casting pro-

Table III
Performance Evaluations of the ChaCha20, Poly1305, and ChaCha20-Poly1305 AEAD on ARM and AVR Platforms

Platform	Algorithm	Cycles	Cycles/Byte	Size [Byte]	Stack [Byte]
ATmega2560 [31]	ChaCha20	17787	268.0	_	268
Cortex-M4 [27]	Salsa20	3311	_	1272	552
Cortex-M0 [32]	ChaCha20	_	39.9	_	_
Cortex-M4 [27]	ChaCha20	3468	_	1328	544
Cortex-M4 [28]‡	ChaCha20	1287	17.6	3174	228
Cortex-M4 [14]	ChaCha20	1487	20.6	734	232
Cortex-M4†	ChaCha20	1651	24.0	1060	236
Cortex-M0 [30]	Chaskey	_	18.3	1308	_
Cortex-M4 [30]	Chaskey	_	7.0	908	_
Cortex-M4 [14]	Poly1305	747	3.6	744	120
Cortex-M4 [14]	Poly1305-ChaCha20	1945	3.6	1322	224
Cortex-M4†	Poly1305	1316	7.5	1184	112
Cortex-M4†	Poly1305-ChaCha20	2997	7.5	2240	236
Cortex-M4 [27]	AES128-GCM	43657	_	2644	812
Cortex-M4 [27]	AES128-EAX	32159	_	2780	932
Cortex-M4 [27]	AES128-CCM	23949	_	2256	780
Cortex-M4 [27]	NORX32	6855	_	1820	320
Cortex-M4 [14]	ChaCha20-Poly1305	3364	28.4	1946	332
Cortex-M4†	ChaCha20-Poly1305	4762	31.5	2596	380

† Our implementation.
‡ These results were obtained by including the π-ChaCha20 function
(cf. https://github.com/joostrijneveld/chacha-arm-cortex-m/) in source code of De Santis et al [14].

$$\begin{cases} r\_limb_{max} \in \{0,1\}^{160} = (\texttt{0x3ffffff,0x3ffff03,0x3ffc0ff,0x3f03fff,0x00fffff}) \\ h\_limb_{max} \in \{0,1\}^{160} = (\texttt{0x3ffffff,0x3ffffff,0x3ffffff,0x3ffffff,0x3ffffff}) \end{cases} \tag{6}$$

Table IV
Comba-Derived Modular Multiplication of a 26-bit 5-limb Structure

	a4	a3	a2	a1	a0
×	b4	b3	b2	b1	b0
	a4*b0	a3*b0	a2*b0	a1*b0	a0*b0
	a3*b1	a2*b1	a1*b1	a0*b1	a4*b1*5
	a2*b2	a1*b2	a0*b2	a4*b2*5	a3*b2*5
	a1*b3	a0*b3	a4*b3*5	a3*b3*5	a2*b3*5
	a0*b4	a4*b4*5	a3*b4*5	a2*b4*5	a1*b4*5
	s4	s3	s2	s1	s0

cedures on handling different perception data types of various IoT applications and purposes. With patternability in the frame organization, advanced data exchange schemes can be constructed for such data exchange of handshaking and firmware over-the-air (FOTA) in future development.

References

[1] S. S. Dhanda, B. Singh, and P. Jindal, "Lightweight Cryptography: A Solution to Secure IoT," *Wireless Personal Communications*, vol. 112, no. 3, p. 1947–1980, Jan. 2020. [Online]. Available: http://dx.doi.org/10.1007/s11277-020-07134-3

[2] D. J. Bernstein, "The Poly1305-AES Message-Authentication Code," in *Proceedings of the Fast Software Encryption*, H. Gilbert and H. Handschuh, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 32–49.

[3] V. V. Deotare, D. Padole, and A. Wakode, "Performance Evaluation of AES using Hardware and Software Codesign," *International Journal on Recent and Innovation Trends in Computing and Communication*, vol. 2, no. 6, pp. 1638–1643, 2014.

[4] P. S. Munoz, N. Tran, B. Craig, B. Dezfouli, and Y. Liu, "Analyzing the Resource Utilization of AES Encryption on IoT Devices," in *Proceedings of the 2018 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*, 2018, pp. 1200–1207.

[5] L. S. Centellas Claros, L. Blanco Coca, and J. P. Sandoval Alcocer, "Comparative study of the symmetric cryptography algorithms AES, 3DES and ChaCha20," *Acta Nova*, vol. 10, no. 3, pp. 283–302, 2022.

[6] Y. Nir and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols," RFC 8439, Jun. 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8439

[7] T. Kao, H. Wang, and J. Li, "Safe MQTT-SN: a lightweight secure encrypted communication in IoT," in *Proceedings of the Journal of physics: Conference series*, vol. 2020, no. 1. IOP Publishing, 2021, p. 012044.

[8] J. P. Degabriele, J. Govinden, F. Günther, and K. G. Paterson, "The Security of ChaCha20-Poly1305 in the Multi-User Setting," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1981–2003. [Online]. Available: https://doi.org/10.1145/3460120.3484814

[9] R. Szabó and A. Gontean, "The development process of an UART chip on FPGA for driving embedded devices," in *Proceedings of the 2015 IEEE 21st International Sym-*

posium for Design and Technology in Electronic Packaging (SIITME), 2015, pp. 175–179.

[10] N. R. Laddha and A. Thakare, "A review on serial communication by UART," *International journal of advanced research in computer science and software engineering*, vol. 3, no. 1, 2013.

[11] K. Rathore, M. Khosla, and A. Raman, "A New Approach to Improve Reliability in UART Using Checksum Algorithm," in *Proceedings of the Fourth International Conference on Computer and Communication Technologies*, K. A. Reddy, B. R. Devi, B. George, K. S. Raju, and M. Sellathurai, Eds. Singapore: Springer Nature Singapore, 2023, pp. 243–252.

[12] M. Daraban, C. Corches, A. Taut, and G. Chindris, "Protocol over UART for Real-Time Applications," in *Proceedings of the 2021 IEEE 27th International Symposium for Design and Technology in Electronic Packaging (SIITME)*, 2021, pp. 85–88.

[13] J. Großschädl, R. M. Avanzi, E. Savaş, and S. Tillich, "Energy-Efficient Software Implementation of Long Integer Modular Arithmetic," in *Proceedings of the Cryptographic Hardware and Embedded Systems – CHES 2005*, J. R. Rao and B. Sunar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 75–90.

[14] F. De Santis, A. Schauer, and G. Sigl, "ChaCha20-Poly1305 authenticated encryption for high-speed embedded IoT applications," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE), 2017*, 2017, pp. 692–697.

[15] T. Zhou and J. Zhang, "Design and Implementation of Agricultural Internet of Things System Based on Aliyun IoT Platform and STM32," *Journal of Physics: Conference Series*, vol. 1574, no. 1, p. 012159, jun 2020. [Online]. Available: https://dx.doi.org/10.1088/1742-6596/1574/1/012159

[16] L. Raju, V. Gurunath, R. Darran, and S. V, "IOT based Energy Management System using STM32 and AWS IOT," in *Proceedings of the 2022 IEEE 2nd International Symposium on Sustainable Energy, Signal Processing and Cyber Security (iSSSC)*, 2022, pp. 1–5.

[17] D. Hercog, T. Lerher, M. Truntič, and O. Težak, "Design and Implementation of ESP32-Based IoT Devices," *Sensors*, vol. 23, no. 15, 2023. [Online]. Available: https://www.mdpi.com/1424-8220/23/15/6739

[18] M. Delvai, U. Eisenmann, and W. Elmenreich, "Intelligent UART module for real-time applications," in *Proceedings of the First Workshop on Intelligent Solutions in Embedded Systems (WISES 2003)*, 2003, pp. 177–185.

[19] KDDI Research, Inc., "Security Analysis of ChaCha20-Poly1305 AEAD," CRYPTREC, 2016. [Online]. Available: https://www.cryptrec.go.jp/exreport/cryptrec-ex-2601-2016.pdf

[20] G. Procter, "A Security Analysis of the Composition of ChaCha20 and Poly1305," Cryptology ePrint Archive, Paper 2014/613, 2014, https://eprint.iacr.org/2014/613. [Online]. Available: https://eprint.iacr.org/2014/613

[21] D. Bernstein, "Salsa20 security," *http://cr.yp.to/snuffle/security.pdf*, 2005.

[22] J.-P. Aumasson, S. Fischer, S. Khazaei, W. Meier, and C. Rechberger, "New features of latin dances: Analysis of salsa, ChaCha, and rumba," Cryptology ePrint Archive, Paper 2007/472, 2007, https://eprint.iacr.org/2007/472. [Online]. Available: https://eprint.iacr.org/2007/472

[23] Z. Shi, B. Zhang, D. Feng, and W. Wu, "Improved Key Recovery Attacks on Reduced-Round Salsa20 and ChaCha," in *Proceedings of the Information Security and Cryptology – ICISC 2012*, T. Kwon, M.-K. Lee, and D. Kwon, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 337–351.

[24] S. Maitra, "Chosen IV Cryptanalysis on Reduced Round ChaCha and Salsa," Cryptology ePrint Archive, Paper 2015/698, 2015, https://eprint.iacr.org/2015/698. [Online]. Available: https://eprint.iacr.org/2015/698

[25] A. R. Choudhuri and S. Maitra, "Differential Cryptanalysis of Salsa and ChaCha – An Evaluation with a Hybrid Model," Cryptology ePrint Archive, Paper 2016/377, 2016, https://eprint.iacr.org/2016/377. [Online]. Available: https://eprint.iacr.org/2016/377

[26] A. Langley, W.-T. Chang, N. Mavrogiannopoulos, J. Strombergson, and S. Josefsson, "ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS)," RFC 7905, Jun. 2016. [Online]. Available: https://www.rfc-editor.org/info/rfc7905

[27] J. Birr-Pixton, "Cifra: Cryptographic primitive collection," https://github.com/ctz/cifra, 2017, [Accessed 13-06-2024].

[28] A. Hülsing, J. Rijneveld, and P. Schwabe, "ARMed SPHINCS – computing a 41KB signature in 16KB of RAM," Cryptology ePrint Archive, Paper 2015/1042, 2015, https://eprint.iacr.org/2015/1042. [Online]. Available: https://eprint.iacr.org/2015/1042

[29] J.-P. Aumasson, P. Jovanovic, and S. Neves, "NORX: parallel and scalable AEAD," in *Proceedings of the Computer Security-ESORICS 2014: 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part II 19*. Springer, 2014, pp. 19–36.

[30] N. Mouha, B. Mennink, A. Van Herrewege, D. Watanabe, B. Preneel, and I. Verbauwhede, "Chaskey: an efficient MAC algorithm for 32-bit microcontrollers," in *Proceedings of the Selected Areas in Cryptography–SAC 2014: 21st International Conference, Montreal, QC, Canada, August 14-15, 2014, Revised Selected Papers 21*. Springer, 2014, pp. 306–323.

[31] M. Hutter and P. Schwabe, "NaCl on 8-bit AVR microcontrollers," in *Proceedings of the Progress in Cryptology–AFRICACRYPT 2013: 6th International Conference on Cryptology in Africa, Cairo, Egypt, June 22-24, 2013. Proceedings 6*. Springer, 2013, pp. 156–172.

[32] N. Samwel and M. Neikes, "ARM implementation of the ChaCha20 block cipher," https://gitlab.science.ru.nl/mneikes/arm-chacha20, 2016, [Accessed 13-06-2024].

Duc-Hung Le received a B.S. degree in Physics and an M.S. degree in Electronic Physics from the University of Science, VNU-HCM, in 2001 and 2005, respectively, and the Ph.D. degree in Advanced Science and Engineering from The University of Electro-Communications (UEC), Tokyo, Japan, in 2013. His research interests include FPGA design, IC design, digital signal processing, and biomedical electronics. Currently, he is with the Faculty of Electronics and Telecommunications, the University of Science, VNU-HCM, Vietnam.

Vu-Minh-Thanh Nguyen received a B.S. degree in Electronics and Telecommunications from the University of Science, VNU-HCM, in 2023. His research interests include IoT Communication and Lightweight Cryptography. Currently, he is with the Faculty of Electronics and Telecommunications, the University of Science, VNU-HCM Vietnam.