IoT Security with ChaCha20-Poly1305

Viggo Petersen, 314203
Software Technology Engineering
VIA University College

IT-SCP1-S22

June 21, 2022

Abstract

IT security is a growing concern for embedded devices as more and more are connected to the internet. Often these devices are developed on different systems without a common operating system and have very little security by default. This paper includes a well-documented reference implementation of the TLS-standard algorithms Chacha20 and Poly1305, specific for embedded devices such as Arduino and ESP32. Benchmarks for both were run on an ESP32 and were found to have significant performance improvements over the traditional AES and HMAC algorithms. This minimalistic and performant implementation can help developers improve the security of their embedded projects.

Keywords: IoT and embedded network security, Chacha20 and Poly1305 reference implementation, IoT TLS standard

Contents

1 Introduction

Encryption has become more important than ever before with the rise of IoT, as more devices are connected to the internet. AES (Advanced Encryption Standard) is the most popular choice for encryption despite it having vulnerabilities such as Cache-Collision Timing Attacks[6] and if any breaking attacks were found then the majority of data would be at risk. In order to avoid this problem, various algorithms have been developed including the stream ciphers Salsa and ChaCha have shown to be faster and not vulnerable to the before-mentioned attack. The performance of the algorithms on non-AES supported hardware has been shown by both Daniel. J. Bernstein and Adam Langley. Therefore the main focus will be on a hardware-specific implementation of ChaCha20 and the associated MAC (Message Authentication Code) algorithm Poly-1305, making up the proposed AEAD (authenticated encryption with associated data) for TLS[7].

There exist various implementations of Chacha20 and Poly-1305, few of which are functional on low-level hardware. The creator of the algorithms, Daniel J. Bernstein, includes reference implementations for C and multiple CPU architectures, none of which compile with AVR-GCC.[1][2] There also exist implementations within some Real-Time Operating Systems such as FreeRTOS and ESP IDF, however, these OSes aren't always used on IoT devices and ESP IDF is specific for the ESP microprocessors.

In order to limit the scope of this paper, the outcome will be only the core implementation of the ChaCha20 and Poly1305 algorithms and not the entire TLS specification, as this includes many networking specific security features. This will serve as a reference implementation for developers to secure their AVR or Arduino projects.

The hypothesis is that this project will have an efficient implementation which will make it easier for developers to implement and understand ChaCha20-Poly1305 for their hardware-based projects.

2 Methods

2.1 Chacha20

ChaCha20 is a stream cipher encryption algorithm with a 256-bit key, meaning it encrypts data byte by byte instead of encrypting blocks of data at once. The central component of the algorithm is the quarter round, which is defined by a set of logical operators on four bytes of data as shown on Figure 1 or as the following C-like code from Daniel J. Bernstein's paper ChaCha, a variant of Salsa20[1].

```
// + adds through carry, ^ is XOR, <<< is Left Roll
a += b; d ^= a; d <<<= 16;
c += d; b ^= c; b <<<= 12;
a += b; d ^= a; d <<<= 8;
c += d; b ^= c; b <<<= 7;
```

ChaCha20 will then run through 20 rounds, where a round consists of four quarter rounds on distinct sets of a matrix. The ChaCha matrix (also called the state) consists of four parts: a 128-bit constant c, little-endian 256-bit key k, little-endian 32-bit block counter b and little-endian 96-bit nonce n, as shown below in it's initial state[3].

```
cccccccc  cccccccc  cccccccc  cccccccc
kkkkkkkk  kkkkkkkk  kkkkkkkk  kkkkkkkk
kkkkkkkk  kkkkkkkk  kkkkkkkk  kkkkkkkk
bbbbbbbb  nnnnnnnn  nnnnnnnn  nnnnnnnn
```

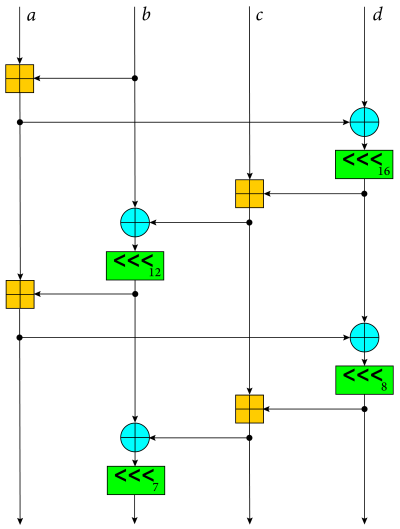


Figure 1: Quarter Round

These rounds will alternate between column rounds and diagonal rounds as shown below with a matrix on indexes. After this process, the final state will be added to the initial state and returned as the ChaCha block. In order to encrypt or decrypt a message, it will be split into 64-byte chunks and XORed with the ChaCha block, and for each chunk of the message the block will be re-calculated and the block counter incremented. The process of encrypting is the same as decrypting because XOR is non-destructive and running it twice will reverse the result.

```
// matrix indexes
 0  1  2  3
 4  5  6  7
 8  9 10 11
12 13 14 15
// column rounds
QUARTERROUND(0, 4,  8, 12)
QUARTERROUND(1, 5,  9, 13)
QUARTERROUND(2, 6, 10, 14)
QUARTERROUND(3, 7, 11, 15)
// diagonal rounds
QUARTERROUND(0, 5, 10, 15)
QUARTERROUND(1, 6, 11, 12)
QUARTERROUND(2, 7,  8, 13)
QUARTERROUND(3, 4,  9, 14)
```

2.2 Poly-1305

Poly-1305 is the MAC used to verify the integrity of the encrypted message by taking a 32-byte generated one-time key and generating a 16-byte tag. The key is split into two pairs, r, and s where each pair must be unique and unpredictable. For the 16-byte set of r applies that the bytes 3, 7, 11, and 15 must have their four most significant bits be 0, and for bytes 4, 8, and 12 their two least significant bits must be 0. Next the message must be divided into 16-byte blocks then a constant prime p is set to 2ˆ130-5 and an accumulator a defined and set to $((a + block) * r)\%p$. Finally, the set s will be added to the accumulator and the tag is extracted. The resulting algorithm will then look as follows, as defined in the spec[3].

```
1  // clamp modifies the key set to set the required bits to 0
2  clamp(r): r &= 0x0ffffffc0ffffffc0ffffffc0fffffff
3  poly1305_mac(msg, key):
4    r = le_bytes_to_num(key[0..15])
5    clamp(r)
6    s = le_bytes_to_num(key[16..31])
7    a = 0  /* a is the accumulator */
8    p = (1<<130)-5
9    for i=1 upto ceil(msg length in bytes / 16)
10     n = le_bytes_to_num(msg[((i-1)*16)..(i*16)] | [0x01])
11     a += n
```

```
12      a = (r * a) % p
13    end
14    a += s
15    return num_to_16_le_bytes(a)
16  end
```

In order to securely generate the 256-bit key used by Poly1305, the first 256-bit of the 512-bit ChaCha Block can be used. To generate this new key, we use the 256-bit key used to encrypt the message, a counter set to 0, and a nonce. For the nonce applies that it must be a unique value for every key so it cannot be randomly generated, instead, a counter or similar non-random number should be used. For ChaCha a 96-bit nonce is used, however, a 64-bit nonce can be used, where the remaining bits will be padded with a constant number, typically 0. For protocols with multiple senders, the constant should be the same for a given sender, however, each sender may have different constants.

2.3 Little endian

Little Endian refers to the way an integer is read as shown in Figure 2. Reading a hexadecimal integer from the highest byte first makes it Big Endian, where as reading from the lowest byte first is Little Endian.

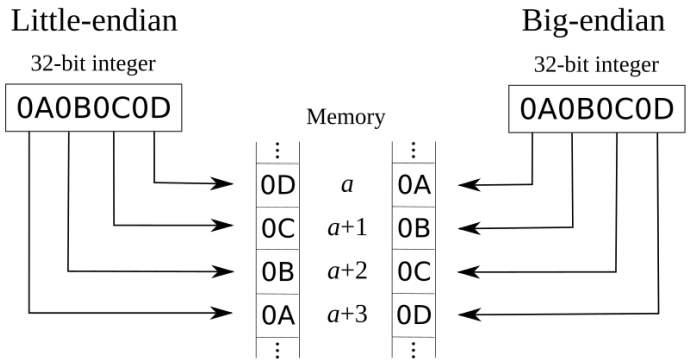


Figure 2: Endian integers

2.4 AVR-GCC

For the implementation, AVR-GCC has been chosen as it is one of the most common compilers and will be compatible with the most common microchips from ATMEGA and ESP. The IETF spec's pseudocode is in a C-like style, along with the reference code by D.J. Bernstein, therefore the code will also be done in C. This means that the core algorithm for ChaCha can be implemented by using the reference, and the Poly1305 specific parts are well enough documented in the IETF spec to implement, test, and validate.

Because Chacha20-poly1305 is intended for TLS, it is most relevant for the ESP32/ESP8266 microprocessors as these have built-in WIFI[9] and are used for Arduino networking modules. The ESP processors use the ESP IoT Development Framework (ESP-IDF), which is built on FreeRTOS and thus compatible with pure AVR drivers.

3 Results

3.1 Chacha20 implementation

The two algorithms are contained in separate driver files and have no dependencies within the project, in order to ensure high cohesion and low coupling. The Chacha algorithm consists of multiple functions, some of which will be included below.

3.1.1 Overview

Below can be seen the overall set of functions the driver includes.

```
1  typedef struct
2  {
3    uint32_t state[16];
4    uint32_t keystream[16]; // contains quarterrounds block
5    size_t available; // remaining keystream is available
6  } chacha20_ctx;
7
8  void chacha20_setup(
9            chacha20_ctx *ctx,
10           const uint8_t *key,
11           uint32_t length,
12           uint8_t nonce[8] );
13
14 void chacha20_set_counter(
15           chacha20_ctx *ctx,
16           uint64_t counter );
17
18 void chacha20_block(
19           chacha20_ctx *ctx,
20           uint32_t output[16] );
21
22 void chacha20_encrypt_bytes(
23           chacha20_ctx *ctx,
24           const uint8_t *in,
25           uint8_t *out,
26           uint32_t length );
27
28 void chacha20_decrypt_bytes(
29           chacha20_ctx *ctx,
30           const uint8_t *in,
```

```
31          uint8_t *out,
32          uint32_t length );
```

3.1.2 Setup

First is the setup function responsible for setting up the initial state of the matrix with a set of parameters. This initial matrix is represented by the *state* variable in the *chacha20_ctx* struct. The function takes a context, *ctx*, as a parameter, so there can be multiple instances of different encryption contexts. Having these encryption states at the same time would allow e.g. having one for each connection to a client/server. The state is then set up according to the Chacha20 algorithm, where the constant, key, and nonce are converted to 32bit Little Endian integers from 8bit, using *U8TO32_LITTLE*. Additionally, the counter defaults to 0, and the *available* value is set to 0. This value is not a part of the algorithm, instead, it is used by the driver to keep track of a block, allowing the remaining bytes in the state key to be used between encryptions.

The parameters of the function include **ctx* a context reference, **key* a key reference, *length* the length of the key, and *nonce* the nonce as an 8-byte array.

```
1  void chacha20_setup(chacha20_ctx *ctx, const uint8_t *key,
       uint32_t length, uint8_t nonce[8]) {
2    const char *constants = (length == 32 ? "expand 32-byte k" :
         "expand 16-byte k");
3    ctx->state[0]  = U8TO32_LITTLE(constants + 0);
4    ctx->state[1]  = U8TO32_LITTLE(constants + 4);
5    ctx->state[2]  = U8TO32_LITTLE(constants + 8);
6    ctx->state[3]  = U8TO32_LITTLE(constants + 12);
7    ctx->state[4]  = U8TO32_LITTLE(key + 0 * 4);
8    ctx->state[5]  = U8TO32_LITTLE(key + 1 * 4);
9    ctx->state[6]  = U8TO32_LITTLE(key + 2 * 4);
10   ctx->state[7]  = U8TO32_LITTLE(key + 3 * 4);
11   ctx->state[8]  = U8TO32_LITTLE(key + 4 * 4);
12   ctx->state[9]  = U8TO32_LITTLE(key + 5 * 4);
13   ctx->state[10] = U8TO32_LITTLE(key + 6 * 4);
14   ctx->state[11] = U8TO32_LITTLE(key + 7 * 4);
15   ctx->state[12] = COUNTER;
16   ctx->state[13] = COUNTER;
17   ctx->state[14] = U8TO32_LITTLE(nonce + 0);
18   ctx->state[15] = U8TO32_LITTLE(nonce + 4);
19   ctx->available = 0;
20 }
```

3.1.3 Encryption/Decryption

The encryption of data is done using XOR, as it is the only bitwise operation to be reversible. This means that both the encryption and decryption of data are the same, both using the function below. The encryption function uses two

functions that are not displayed in this paper, *chacha20_block*, and *chacha20_xor* because these follow the exact algorithm described in Section 2.1 Chacha20. The block function runs through the quarter-rounds to calculate the current state, and the xor function simply XOR the values with the block. Note that the block is stored inside the *keystream* variable when called on line 19.

The *available* variable is used to first check if any remaining *keystream* is available from the last run, if so it is used before a new block state is generated, which ensures more re-usability between encryptions. After this, the remaining length of the message is run through and encrypted. In this loop the *amount* variable is used to keep track of the remaining length of the message and if any remains once the encryption is done the length will be stored in the *available* variable.

The parameters of the function include **ctx* a context reference, **in* the input stream reference, **out* the output stream reference and *length* the length of the message to be encrypted.

```
1  void chacha20_encrypt_bytes(chacha20_ctx *ctx, const uint8_t *
       in, uint8_t *out, uint32_t length) {
2    if (!length) {
3      return;
4    }
5    uint8_t *const k = (uint8_t *)ctx->keystream;
6
7    // If remaining keystream is available, use it
8    if (ctx->available) {
9      uint32_t amount = MIN(length, ctx->available);
10     chacha20_xor(k + (sizeof(ctx->keystream) - ctx->available),
           &in, &out, amount);
11     ctx->available -= amount;
12     length -= amount;
13   }
14
15   // XOR remaining message if any
16   while (length) {
17     uint32_t amount = MIN(length, sizeof(ctx->keystream));
18     // Update keystream with block
19     chacha20_block(ctx, ctx->keystream);
20     chacha20_xor(k, &in, &out, amount);
21     length -= amount;
22     ctx->available = sizeof(ctx->keystream) - amount;
23   }
24 }
```

3.1.4 Example

Using the algorithms is made as simple as possible but also versatile enough to fit most use cases. Below is a sample snippet of the simplest use of Chacha encryption and decryption, where C will handle the conversion from string to

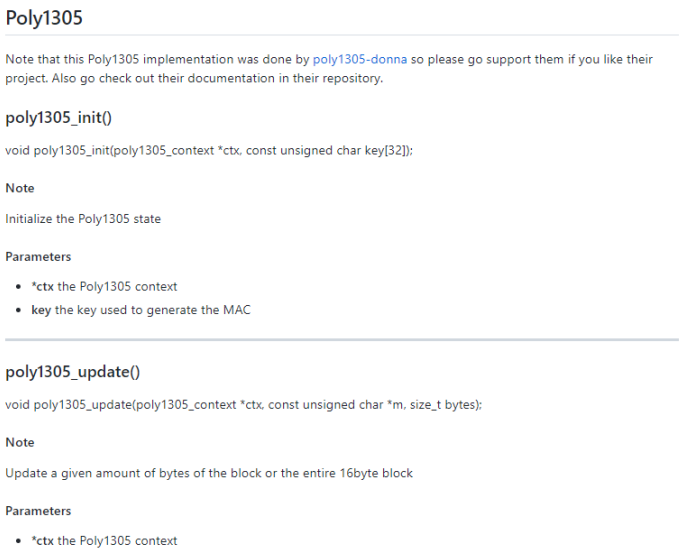
unsigned integer array. This conversion is likely done by the compiler, therefore cases where the variables come in as strings it may be required to programmatically convert them to integers.

```
// Set up all required variables
uint8_t text_key[32] = "12345678901234567890123456789012"; // some 32byte key
uint8_t text_plain[20] = "Some secret message";
uint8_t nonce[8] = "12345678"; // 8byte nonce
chacha20_ctx ctx;
uint32_t len = sizeof text_plain;
uint8_t *output = alloca(len);
memset(output, 0, len);

// setup state and encrypt
chacha20_setup(&ctx, text_key, sizeof(text_key), nonce);
chacha20_encrypt_bytes(&ctx, text_plain, output, len);
printf("%s\n", output); // Prints encrypted message

// reset state and decrypt
chacha20_setup(&ctx, text_key, sizeof(text_key), nonce);
chacha20_decrypt_bytes(&ctx, output, output, len);
printf("%s\n", output); // prints 'Some secret message'
```

3.2 Poly1305 implementation

For Poly1305 there already exists an implementation for various architectures, one of which compiles with AVR-GCC. Instead of implementing something which already exists, this library was used instead.

Additional documentation was written, as can be seen in Figure 3, where each function and its parameters were documented. This documentation was written in the paper's Github repository along with code examples of how to correctly use the library. Code for benchmarking the library was also written to execute on start-up, along with tests to ensure everything is functional.



Poly1305

Note that this Poly1305 implementation was done by poly1305-donna so please go support them if you like their project. Also go check out their documentation in their repository.

poly1305_init()

void poly1305_init(poly1305_context *ctx, const unsigned char key[32]);

Note

Initialize the Poly1305 state

Parameters

- ***ctx** the Poly1305 context
- **key** the key used to generate the MAC

poly1305_update()

void poly1305_update(poly1305_context *ctx, const unsigned char *m, size_t bytes);

Note

Update a given amount of bytes of the block or the entire 16byte block

Parameters

- ***ctx** the Poly1305 context

Figure 3: Poly1305 documentation

3.3 Function documentation

Every function has been well documented using the Javadoc styling, as can be seen in the snippet below. The Github page for the project, which can be found in appendix 7.1 on page 19, also documents every function and has examples of how to use them.

```
1  /**
2  * @brief Sets up the inital chacha context state, including the
       constant, key, counter and none
3  *
4  * @param ctx the Chacha20 context
5  * @param key the encryption key
6  * @param length the length of the key
7  * @param nonce the nonce
8  */
9  void chacha20_setup(chacha20_ctx *ctx, const uint8_t *key,
       uint32_t length, uint8_t nonce[8]) {
10 ...
11
12 /**
13 * @brief Chacha20 encrypt a plain text message by generating
       keystream and XOR'ing the stream and plain text message
14 *
15 * @param ctx the Chacha20 context
16 * @param in the plain text message to be encrypted
17 * @param out the output variable
18 * @param length the length of the message
```

```
19 */
20 void chacha20_encrypt_bytes(chacha20_ctx *ctx, const uint8_t *
       in, uint8_t *out, uint32_t length) {
21 ...
```

3.4 Benchmarks

To validate the functionality and performance of the implementations a series of tests and benchmarks were conducted. The unit tests to ensure the functionality was created by Insane Coder[8] in their implementation and it tested the examples from the Chacha20-poly1305 IETF specification[3].

```
1  void test_chacha_run() {
2    test_keystream("00000...", "0000000000000000", "76b8e...");
3    test_keystream("...00001", "0000000000000000", "4540f...");
4    test_keystream("00000...", "0000000000000001", "de9cb...");
5    test_keystream("00000...", "0100000000000000", "ef3fd...");
6    test_keystream("...d1e1f", "0001020304050607", "f798a...");
7
8    test_encipherment("00000...", "0000000000000000","00000...");
9    test_encipherment("...00001", "0000000000000002","416e7...");
10   test_encipherment("1c924...", "0000000000000002","27547...");
11 }
```

The benchmarks were intended to calculate the MB/s of each algorithm, however, they had to be implemented using chunks of 1000 bytes due to the ESP32 modules only having between 320KiB and 512KiB. The code would therefore generate a string of 1000 characters and encrypt it 1000 times and measure the time spend. This test will then be executed 10 times so an average can be calculated, and a terminal snippet of this can be seen in appendix 4 on page 19.

```
1  void benchmark_chacha(char *str) { // str is 1KB
2  ...
3    for (int j = 0; j < 10; j++) \{
4      clock\_t t = clock();
5      for (int i = 0; i < 1000; i++) \{
6        chacha20\_encrypt\_bytes(\&ctx, (uint8\_t*)str, output,
             len);
7      \}
8      t = clock() - t;
9      double time\_spent = (double)t / CLOCKS\_PER\_SEC;
10     printf("Took \%fs\textbackslash\{\}n", time\_spent);
11   \}
```

The results of these benchmarks can be seen in the tables below.

3.4.1 Chacha20 benchmark

Run	Execution time	Data size
1	0.340s	1MB
2	0.340s	1MB
3	0.330s	1MB
4	0.340s	1MB
5	0.340s	1MB
6	0.340s	1MB
7	0.330s	1MB
8	0.340s	1MB
9	0.340s	1MB
10	0.340s	1MB

The average execution time

$$\frac{(2 * 0.33 + 8 * 0.34)}{10} = 0.338s$$

and the MB/s is

$$\frac{1}{0.338} = 2.9586MB/s$$

3.4.2 Poly1305 benchmark

Run	Execution time	Data size
1	0.220s	1MB
2	0.210s	1MB
3	0.220s	1MB
4	0.210s	1MB
5	0.220s	1MB
6	0.220s	1MB
7	0.210s	1MB
8	0.220s	1MB
9	0.220s	1MB
10	0.210s	1MB

The average execution time

$$\frac{(6 * 0.22 + 4 * 0.21)}{10} = 0.216s$$

and the MB/s is

$$\frac{1}{0.216} = 4.6296MB/s$$

4 Discussion

4.1 Chacha20 implementation

The Chacha20 implementation in section 3.1 is based on Daniel J. Bernstein's reference implementation[1] and the article by Insane Coder[8]. It aims to combine the best features of both, for the code to be efficient, easier to read, understand and implement.

Ensuring the code is easy to understand is difficult as it depends on the user's prior knowledge and experience, and cannot be directly measured. In order to accomplish this, the functions were documented in multiple stages. First is the overview in the repository, which gives the user an overview of all the available functions and their function. Second is the documentation within the code above each function, which allows IDEs to show the expected parameters. Lastly are the single-line comments inside the function explaining the most complicated parts. This documentation should be enough for developers to develop software using the library.

Specifically for low-level hardware, it is important that libraries are small and only the parts which are in use get uploaded to the device. Using libraries can be a problem as these often contain many different algorithms or functionalities, and are not always written with hardware in mind. This also means they can have a lot of dependencies within their library, making it difficult to extract only the needed parts of a library. To avoid this the project library was written with low coupling and high cohesion in mind, making sure each file only contains a single algorithm and has no dependencies within the library. This allows the developers to extract the individual algorithms and use them without figuring out dependencies or installing additional libraries.

As can be seen in *Section 3.1.1 Overview*, all of the functions use reference variables such as **ctx* and **key* in order to improve the performance by not pushing large data sizes to the stack, but instead only pushing reference addresses. Functions also don't return data, instead, they require an output variable reference in order to not override the original data set.

The setup function in *Section 3.1.2 Setup* uses a similar style and naming convention to Daniel J. Bernstein's reference implementation when creating the initial Chacha state. This helps readability when comparing the library to other references following his style. Defined variables and functions, such as *U8TO32_LITTLE* and *COUNTER*, are used as often as possible making the code more readable.

4.1.1 Encryption/Decryption

The encryption *Section 3.1.3* is where the project library varies greatly from Daniel J. Bernstein's implementation. His implementation expects only a single encryption and discards unused bytes from the state, by generating the block as

the first step. The implementation in this project uses the *available* variable to allow multiple encryptions to be done with the same state, given they are shorter than the key stream. This has the benefit that messages can be encrypted in chunks instead of having to handle the encryption in one large action. Because low-power devices often have very little memory, this allows them to encrypt longer messages without having to regenerate the block between every chunk and thereby save clock cycles. The ESP32 module used in this project only has 520KiB, making it very difficult to encrypt larger messages without generating unnecessary blocks using Bernstein's logic. This would require the developer to use the code, to manually load the message in 64-byte chunks, in order to use the entire Chacha state without generating a new unused block. Whereas using the *available* variable allows the developer to load chunks of any size and avoid additional logic.

Because the encryption function contains this additional logic, the XOR logic has been put into a function of its own for better reusability. The *chacha20_xor* runs through a given keystream, message, and length. It is being used first if any keystream is available from previous runs, where it will calculate the offset in the keystream as a starting point. Secondly, it is being used in the primary loop where the new block is generated first and then XORed. In addition to improving reusability, it also lessens complexity by using functions over multiple loops inside the existing loop.

4.1.2 Example

Along with documentation, having examples is very useful in helping the user understand how to use the code. For this reason, a fully functional example code example was added to the main file. The example calls the setup function twice to reset the state and ensure they are identical on encryption and decryption, but this will likely not be the case in real projects as one server will encrypt and another will decrypt. It shows the use of both functions and prints the messages for the user to see. The only aspect which may be confusing is the use of strings being assigned to unsigned integer arrays. This will cause GCC to display a warning with unexpected data assignment, however, it works because C reads the characters as their integer values. Any intermediate C developer working on an embedded project would supposedly understand the code.

4.2 Poly1305 implementation

The Poly1305 library in use is *poly1305-donna*[4] because this library follows the same paper done by Daniel Bernstein[2] and is therefore compatible with the Chacha20 implementation. This library was found to have very few dependencies and compile using AVR GCC, which fulfilled the expectations of the paper. Therefore there was no reason to re-implement the algorithm, despite the minor issues with the library, of which there are three.

First *poly1305-donna* is compatible with multiple architectures by using a compiler variable for setting the CPU bit value, which adds complexity to the

library.

Secondly, the library has a full test set up inside the main file, *poly1305-donna.c*, adding a lot of code to the file. This makes the file more confusing to look through as one of the functions contains a lot of logic and data sets not related to the algorithm.

lastly, the architecture-specific logic is all implemented inside separate header files. In C header files are used for function declarations and macro definitions, typically not large implementations. This makes the code significantly harder to read, having a large header file with no related C files.

4.3 Function documentation

As most functions in the project library have been documented, using common Javadoc, they can be loaded by many IDEs and frameworks for automatically generating documentation websites. This is useful as many of the popular IDEs will use the function documentation to display function parameters and suggestions while programming.

4.4 Benchmarks

For both the Chacha20 implementation and the Poly1305-donna library benchmarks were created in order to calculate the MB/s speed of each algorithm on the testing hardware. These can be used to determine how well the algorithms run on any given hardware, as the user can run the benchmarks on their designated hardware. It also serves as a validator that the algorithm runs at the expected speeds compared to AES on embedded systems, given existing benchmarks for AES on said hardware exist.

4.4.1 Chacha20 benchmark

The Chacha20 benchmark provided an average execution time of 0.338 seconds and a MB/s of 2.9586. We can use these numbers to compare the performance to the AES benchmarks done by Oryx Embedded[10]. Looking at the AES-256 benchmarks we can see a speed of between $0.632MB/s$ to $1.262MB/s$ depending on the encryption mode. We can then use these values compared to the Chacha20 benchmarks.

$$max = \frac{2.9586}{0.632} = 4.6813$$

$$min = \frac{2.9586}{1.262} = 2.3444$$

Compared to the AES benchmarks, Chacha20 is between 2.4 and 4.7 times faster running on an ESP32. Based on the IETF spec, *"It is considerably faster than AES in software-only implementations, making it around three times as fast ..."*(1. Introduction)[3], the project implementation is well within the expected speed.

Comparing these benchmarks does come with a few complications. The benchmarks by Oryx Embedded claim to be using megabyte MB but whether they were using 1 million bytes or Mebibytes MiB of 2^{20} bytes is unclear. However, this would only cause a 4.8% performance difference, which is low enough to be insignificant.

Another complication is the use of hardware, the Oryx team used an official ESP32-DevKitC whereas the benchmarks in this paper used an unofficial ESP32-WROOM-32U module. These unofficial modules are known to have slight problems and differences from the official ones. One of these issues came when uploading the code, where the Enable Pin (EN) turned off too quickly for the ESP-IDF uploader to correctly establish a connection to the processor. This was fixed by adding a capacitor between the EN port and ground, as can be seen in Appendix 7.3 on page 20 on figure 5. Based on the performance of the benchmarks there are no signs of this affecting performance once the code has been uploaded correctly.

4.4.2 Poly1305 benchmark

In order to compare the Poly1305 benchmarks, we can use one of the common TLS 1.2 MAC algorithms such as HMAC-SHA256[12]. The SSL library WolfSSL[11] has benchmarks of various algorithms on different architectures, one of which is a benchmark of HMAC-SHA256 on the ESP32-WROOM-32 module. HMAC-SHA256 is capable of 1.733MB/s compared to the 4.6296MB/s of poly1305-donna. We can then calculate the difference in speed.

$$\frac{4.6296}{1.733} = 2.6714$$

Poly1305 is therefore 2.7 times faster on the same hardware, which is a significant increase. Yet, this may not be representative of real-life observations, as TLS uses many different algorithms in combination with each other and these may have better or worse performance. It does show the potential performance increase, especially in combination with Chacha20 over AES.

5 Conclusion

The final product of the paper is a well-documented implementation of both Chacha20 and Poly1305, with benchmarks to confirm the performances of both, verifying that they are as fast as expected. It can therefore be concluded that Chacha20-poly1305 is a valid candidate for IoT and embedded devices, and this project can help developers understand and implement efficient solutions.

6 Literature

References

[1] Daniel J. Bernstein. (2008).
The ChaCha family of stream ciphers
https://cr.yp.to/chacha.html
Retrieved February 2022 online.

[2] Daniel J. Bernstein. (2005). A state-of-the-art message-authentication code
http://cr.yp.to/mac.html.
Retrieved February 2022 online.

[3] Y. Nir, Dell EMC, A. Langley, Google, Inc. (2018). ChaCha20 and Poly1305 for IETF Protocols.
https://datatracker.ietf.org/doc/html/rfc8439
Retrieved March 2022 online.

[4] Andrew Moon. (2016). poly1305-donna.
https://github.com/floodyberry/poly1305-donna
Latest commit e6ad6e.

[5] GNU Compiler Collection. AVR GCC.
https://gcc.gnu.org/wiki/avr-gcc
Retrieved March 2022 online.

[6] Joseph Bonneau, Ilya Mironov. (2006).
https://www.iacr.org/archive/ches2006/16/16.pdf
Retrieved March 2022 online.

[7] Adam Langley, Wan-Teh Chang, Nikos Mavrogiannopoulos, Joachim Strombergson, Simon Josefsson. (2016).
https://datatracker.ietf.org/doc/html/rfc7905
Retrieved March 2022 online.

[8] Insane Coder. (2014). Avoid incorrect ChaCha20 implementations.
https://insanecoding.blogspot.com/2014/06/avoid-incorrect-chacha20-implementations.html
Retrieved May 2022 online.

[9] Espressif. (2021).
https://docs.espressif.com/projects/esp-idf/en/stable/esp32/get-started/index.html#introduction
Retrieved May 2022 online.

[10] Oryx Embedded. Crypto Benchmark on ESP32 MCU.
https://www.oryx-embedded.com/benchmark/espressif/crypto-esp32.html
Retrieved June 2022 online.

[11] WolfSSL.
Espressif ESP32 WROOM Benchmark.
https://www.wolfssl.com/docs/benchmarks/
Retrieved June 2022 online.

[12] T. Dierks & E. Rescorla . (2008).
Major Differences from TLS 1.1
https://datatracker.ietf.org/doc/html/rfc5246#section-1.2
Retrieved June 2022 online.

[13] Viggo Petersen.
chacha20-poly1305. (2022).
https://github.com/jhviggo/chacha20-poly1305

7 Appendix

7.1 Project repository

https://github.com/jhviggo/chacha20-poly1305

7.2 Benchmark terminal

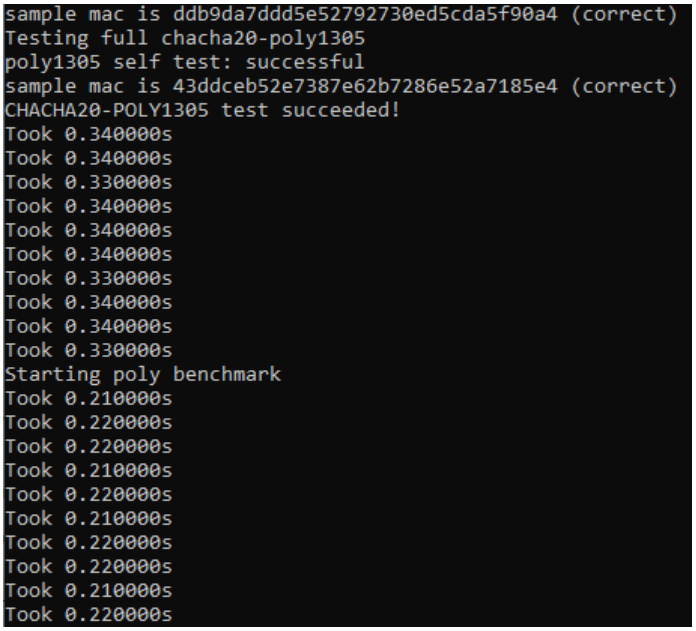


Figure 4: Chacha20-poly1305 Benchmarks

7.3 Hardware module

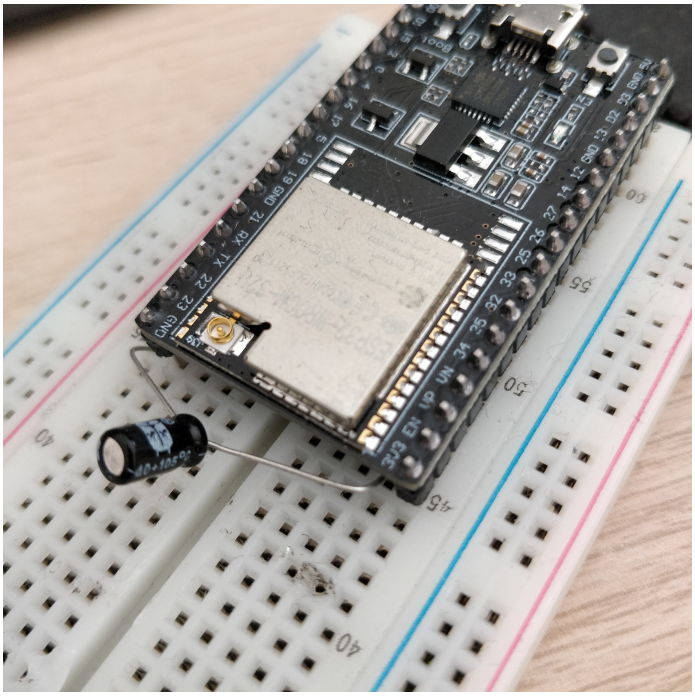


Figure 5: ESP32 module