International Journal for Multidisciplinary Research (IJFMR)
E-ISSN: 2582-2160 ● Website: www.ijfmr.com ● Email: editor@ijfmr.com

Description, Implementation and Performance/Security Analysis of ASCON128 V1.2

Srivastava Shivang[1], Wan Yong[2], Heng Hian Wee[3]

[1,2,3]Graduate Student, Nanyang Technological University

Abstract

The aim of the report is to introduce detailed description of the functioning of and to provide an efficient implementation (using bit-sliced implementation of S-Box) in C of encryption, decryption and authentication of the lightweight authenticated encryption algorithm ASCON 128 (sponge based SPN network based on Keccak like operations) as per the v 1.2 submitted to NIST. This is achieved using a sponge construction by setting up initialization state and performing 12 rounds of permutations, XORing consecutively the plaintext blocks (encryption) or ciphertext blocks (decryption) with first 64 bits of the state (after 6 rounds of sponge permutations) and after all consecutive operations on ciphertext/plaintext blocks, XORing the last 128 bits of the resultant state with the secret Key (after 12 rounds of final permutation operations) to produce/re-create the tag. Discussions on Key Features, Performance Analysis (comparison of ASCON 128 v/s AES 128 in GCM mode on our own PC), Performance Comparisons with other NIST lightweight cryptography contest finalists on ARM Cortex M3, and discussions on the Security Analysis (side channel attacks: fault injection and power analysis) of ASCON are included in this report.

Keywords: ASCON, Symmetric Key Cryptography, Lightweight Cryptography

I. INTRODUCTION

In the last few decades, Technological innovations have been increasing exponentially. IOT and lightweight mobile devices are on a rise, which has moved the focus in cryptography to include lightweight cryptography algorithms for such devices to maintain security and privacy. This need has arisen because implementations like AES are not viable (and too expensive) on IOT (lightweight) devices with low computing and memory power. In other words, security for IOT and lightweight devices is missing. AES, for example, needs to be performed in GCM mode (combination of 2 algorithms) to perform authenticated encryption. However, NIST also realized that mistakes like using short keys must be avoided and custom implementations are risky. Hence, NIST in 2019 launched a Lightweight cryptography contest [1]. On March 29,2021 NIST announced the finalists, and on February 7, 2023, NIST finally declared the ASCON family (lightweight authenticated encryption) to be the winner of lightweight cryptography [1].

The agenda of this report is to implement and dig deeper into the topic of ASCON. This report demonstrates the overall functioning and methodology of ASCON 128 (v1.2) on a conceptual level as well as a practical level with an efficient implementation on C. Discussions on the key features of ASCON, performance comparison of ASCON with other NIST Lightweight Cryptography Contest

finalists on ARM Cortex M3 (results taken from https://rweather.github.io/) and Security Analysis (side channel attacks, fault injection and power analysis (results taken from research papers)) are included in the report. The key features of ASCON, the increased speed and the ability to perform two operations at once (no two separate "algorithms" for authentication and encryption) are deduced in this report by performing a comparison of the average speed of execution of the authenticated encryption process using ASCON 128 (using C) v/s AES 128 in GCM mode (using openSSL) on our Personal Computer.

II. METHODOLOGY

**__Disclaimer__: This section II of the report (including II-IA and II-IB) was drafted by referencing the official submission of ASCON v1.2 to NIST.[3] ASCON is not our own algorithm and the authors of this paper do not claim ownership of ASCON. This section attempts to clearly explain the parameter choice, encryption/decryption/authentication model and the permutation operations to be used in the next section (for an efficient implementation), as mentioned in the official submission of ASCON v1.2 to NIST [3]. Any figures (images) or formulae used directly from the original paper are referenced to the original ASCON paper **

II-I PARAMETERS AND MODEL DESCRIPTION

As can be seen in the figure below from the original submission to NIST, the parameters for ASCON 128 v1.2 are:

Secret key: 128 bits; Number used once (to prevent replay attacks and ensure uniqueness of encryption and tags): 128 bits; Data Block Size: 64 bits; IV: 64 bits; Initialization and Finalization permutation operation are performed 12 times; Other consequent permutation operations are performed 6 times.

Table 1: Parameters for recommended authenticated encryption schemes.

Name	Algorithms	Bit size of				Rounds	
		key	nonce	tag	data block	p^a	p^b
Ascon-128	$\mathcal{E}, \mathcal{D}_{128,64,12,6}$	128	128	128	64	12	6

Figure 1: ASCON-128 parameters [2]

II-IA: Encryption

The working of ASCON (encryption) is shown below:

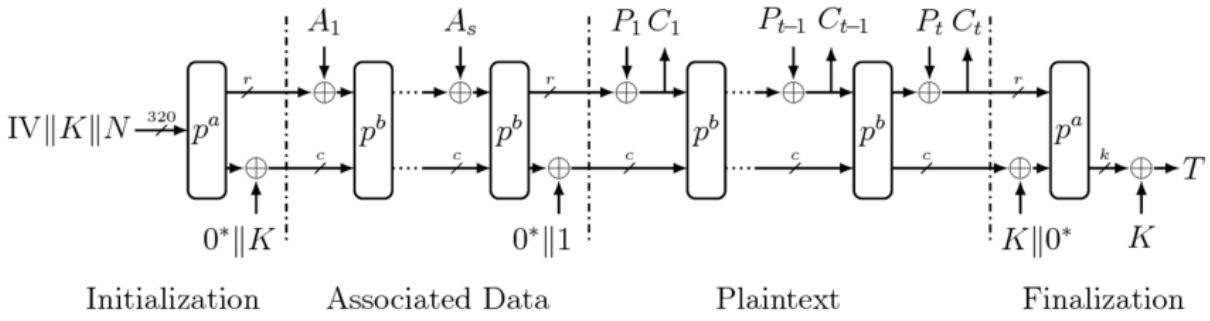

Figure 2A: ASCON working (encryption) [2]

The working for the encryption (based on SPN: sponge construction (or Keccak operations)), can be divided into 4 phases:

*Note: State sees 320 bits at each of Keccak operations. r=64 bits and c=256 bits and r+c=320 bits. p^a implies a=12 rounds of permutation operations, and p^b implies b=6 rounds of permutation operations. T is the total number of plaintext blocks *

Initialization phase:
Step 1: The internal state includes a block concatenation of 320 bits (internal state size): 128 bit Nonce, 128 bit Key and 64 bit IV in the form IV||K||N. This internal state goes through a=12 rounds of permutation operations (p). The first r=64 bits of the resultant of this initialization are used as the first 64 bits of the initialized internal state going forward for XORing with the Associated data (Associated data phase) or plaintext blocks (Plaintext phase) (if Associated data phase is excluded).

Note: 80400c0600000000 is the fixed IV for ASCON 128 in the original submission v1.2 to NIST. The report will later use this IV in its implementation. [3]

Step 2: The 320-bit internal state obtained in step 1 above is XOR'd with the 128-bit secret key padded with 192 0s in the beginning (taking the form 0*||K). Essentially, the last 128 bits of the state are XOR'd with the key (last 2 rows of a 5X64 representation of the state are XOR'd with the key). The resultant c=256 bits of this extra step serve as the last c=256 bits (c=State size (320)- data block size (64) = 256 bits) of the initialized internal state going forward for Associated data phase or Plaintext phase (if Associated data phase is excluded). This step 2 is done to ensure diffusion properties. This step 2 also ensures that the internal state depends on both the IV and Secret Key [4].

Associated Data phase (optional):
In ASCON, additional input data that is authorized (but not encrypted) is known as Associated Data (AD). [4] It is used to verify the authenticity of the message/ verify that this additional data hasn't been tampered with during transmission. [4].

Step 1:
In this phase, the first r=64 bits of the initialized internal state after step 1 of initialization phase (first of 5 rows of 5X64 representation of the state) are XOR'd with the first block (64 bits) of the Associated data and fed as the first r=64 bits of the next internal state. The remaining c=256 bits (last 4 rows of 5X64 representation of the state) are taken directly from the last 256 bits of the state after Step 2 of the initialization phase mentioned above. This internal state is fed to b=6 rounds of the permutation operations.

Steps 2 onwards:
The resultant of 6 rounds of the permutation operations is now the new internal state. At this stage, the first associated data block is accounted for. r=64 bits of this new state (first row of 5X64 state representation) is XOR'd with the next (second) block of 64-bits of the Associated data. The remaining c=256 bits are carried forward from the state itself. This modified state is now fed again to 6 rounds of

permutation operations. This process continues for "s" associated data blocks (64 bits each). In other words, this process is continued till the associated data is exhausted. It is easy to see that the state obtained from the last XORing with the s'th block is also fed to the same b=6 rounds of permutations operations. This new state is now used for the final step shown below.

Final step:

First r=64 bits of resultant state after the s'th (last) round of permutation operation is now the first e=64 bits of the internal state for the plaintext phase.

For the remaining c=256 bits, an operation is performed on the state (after the s'th (last) round of permutation operation). This state is XOR'd with 0*||1 (last Bit of the state is XOR'd with 1). That is, 1 is padded by 319 0s in the beginning to mask the last bit of the state, to increase the confusion. The last c=256 bits (last 4 rows of the 5X64 state representation) of the state for the plaintexts are used from the last c=256 bits from the resultant state of this final step's XOR operation. The new beginning 320- bit state for the plaintext phase is now ready (r+c bits)

Plaintext phase:

Step 1:

The internal state (320 bits) is received from the final step of Associated phase. The first r=64 bits (1st row of 5X64 state representation) is XOR'd with the first plaintext block (of block length 64). The resultant of this XOR is the first Ciphertext block (64 bits). This resultant ciphertext block also acts as the first r=64 bits of the internal state to be fed into b=6 rounds of permutations. The remaining c=256 bits of the internal state to be fed into 6 rounds of permutations are carried over as the c bits of the resultant state of the final step of the associated data phase (after XORing with 0*||1). This complete 320-bit state now actually goes through 6 rounds of the permutation operation, as mentioned in this Step 1.

Steps 2 onwards:

The resultant state from 6 rounds of the first permutation operation are now used in a similar way as Step 1 of the Plaintext phase mentioned above. i.e., first r=64 bits of this resultant state XOR'd with the 2nd plaintext block of 64 bits and used as the 2nd ciphertext block. The remaining 256 bits are carried forward from the resultant state (state in the beginning of step 2). This is now fed again to 6 rounds of the permutation operation. This process (of XORing with consecutive plaintext blocks and then feeding into 6 permutation rounds) continues t-1 times until t-1 64-bit plaintext blocks are exhausted, and we have resultant t-1 64-bit ciphertext blocks. The state after this t-1'th permutation operations (performed after deciphering the t-1'th ciphertext block) is used in the final step of the Plaintext phase.

Note: t =total number of plaintext blocks

Final step:

For the last 64-bit plaintext block, we XOR the first r=64 bits of the state resulting from the 6 permutation rounds after t-1'th permutation operations (performed after deciphering the t-1'th ciphertext block). This XOR'd 64-bit result acts as the final ciphertext block and the first r=64 bits of the beginning internal state for the Finalization phase. The remaining c=256 bits for this beginning internal state of the finalization phase are carried forward from the last 256 bits after t-1'th permutation operations. At this stage however,

we do **not** feed this state to the t'th 6 permutation rounds. Thie beginning state for the Finalization phase is now ready.

Note: It is important to note that encryption part of ASCON is finished at this point. What remains now is creating a tag to be used later for proving the authenticity of the messages

Finalization phase:

Step 1:

The first r=64 bits obtained from the XOR'd result (for last plaintext block) of final step of the plaintext phase act as the first 64 bits of the state to be fed into the final a=12 rounds of permutation. The last c=256 bits of the resultant of the final step of the plaintext phase are now XOR'd with K||0*. i.e., the 128-bit Key is padded with 128 0s at the end. Essentially, the first 128 bits of the remaining c=256 bits from the resultant of the final step of the plaintext phase (2nd and 3rd rows of the 5X64 state representation) are XOR'd with the key. bits. After this XOR operation, the resultant 256 bits act as the 256 remaining bits of the state to be fed into the final a=12 rounds of permutation operation.

The resultant r+c=320-bit (r and c obtained from the step 1 above (of the finalization phase)) state is fed into a=12 rounds of permutation operation.

Step 2(Tag creation/ final step):

The last k=128 bits (last 2 rows of the 5X 64 state representation) of the resultant state from the last 12 rounds of permutation operation of step 1 of the Finalization phase are XOR'd with the key to produce the Tag. It is important to note that this tag is unique for each plaintext and keypair (Nonce too), as well as that it serves as authenticity of the message, implying that the message has not been tampered with in transit. [4]

Important pass-over after encryption:

The encrypted passes the Nonce, IV (fixed) to the receiver. It is assumed that the secret key has already been shared between the encryptor and receiver (decryptor). The encryptor also passes to the decryptor the ciphertext blocks and the created tag.

II-IB: Decryption

The description of II-IA was of the encryption process. Given below also is the decryption process

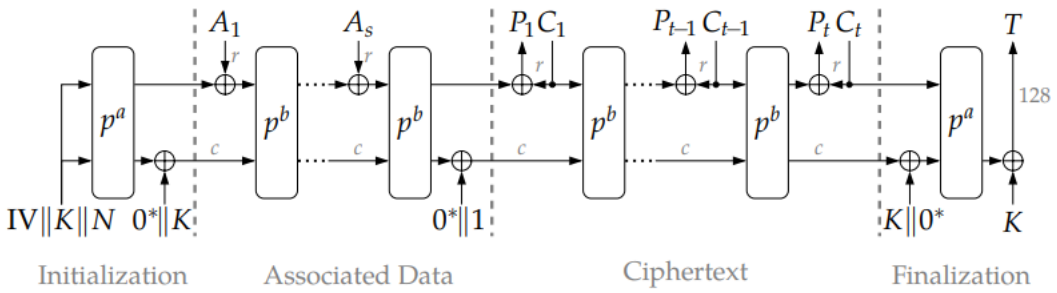

Figure 2B: ASCON working (decryption) [3]

As in encryption, the working of the decryption is defined in 3 phases:

Initialization phase:

This phase is performed by the decryptor in the same as Initialization phase of encryption. Please refer to Initialization phase section of part "II-IA: Encryption" of this report for more details. It is important for decryptor to use the same IV(fixed), Nonce and key as provided by the encryptor to perform this step. It is assumed that a secret key has been shared between the encryptor and the receiver (decryptor) previously. The ciphertext blocks are also passed to the decryptor for decryption, along with the tag for authentication.

Associated phase(optional):

This phase is performed by the decryptor in the same way as Initialization phase of encryption. Please refer to Associated phase section of part "II-IA: Encryption" of this report for more details. It is important to note here that the same Associated text blocks provided by the encryptor must be used for authenticating additional data. (eventually producing equivalent tag, in this case).

Ciphertext phase:

The internal working of this phase is slightly similar to the Plaintext phase of encryption. However, it is important to note that for each step (for each ciphertext block), before feeding the state to b=6 rounds of the permutation operation, the decryptor directly has the 64-bit Ciphertext Block (and not the plaintext block) as provided by the encryptor. Therefore, the receiver XOR's this 64-bit Ciphertext block with the internal state at each stage to decrypt the produce 64 plaintext block. This operation is performed as the decryption step to produce one plaintext block after another (directly XORing for the first ciphertext block and, for the remaining blocks, XORing after b=6 rounds of permutation operations of the state for each ciphertext block). As mentioned before, the Ciphertext block along with the remaining 256-bits of the received state are directly fed to b=6 rounds of permutation operation at each stage. This is to ensure that receiver decrypts the correct consecutive plaintext blocks as well as eventually re computes the accurate tag that matches the encryptors tag.

Finalization phase:

This phase is performed in the same way as finalization phase of encryption. Please refer to Finalization phase section of part "II-IA: Encryption" of this report for more details. However, it is important to note that here, the decryptor is re-creating the tag that the original encryptor created. Once the tag is calculated, it is compared to the tag provided by the encryptor claiming authenticity. If the re-calculated tag matches the encryptor's tag, the messages and sender are authenticated.

II-II. PERMUTATION OPERATIONS DETAILS

In the previous discussion (Model description), the permutation operations were mentioned at each stage. In this section of the report, the permutation operations will be discussed.

Given below is the 320-bit Internal State of ASCON $S= x_0||x_1||x_2||x_3||x_4$ [3]:

Figure 3: ASCON 320-bit Internal State [3]

In the above Figure, it can be see that that the Internal State is divided into 5 rows of 64 bits (columns) each. From the discussion about the Model description, it can be seen from the Step 1 of the Initialization Phase that the very first Internal State is in the form IV||K||N. Therefore, the first row x_0 is the IV (64 bits), the next two rows x_1 and x_2 store the secret key (128 bit) and the last two rows x_3 and x_4 store the Nonce (128 bit).

*Note: In the Model description, sometimes first 64 bits was mentioned as the first row and last 128 bits was mentioned as the last 2 rows. A constant mention of the 4X64 bit state representation was made. It becomes clear now what the rows and the state representation mean. It is easy to see that during the Plaintext phase in encryption, the plaintext block is XOR'd with the first row of the state to produce each ciphertext block. Similarly, during the Ciphertext phase in decryption, the ciphertext block is XOR'd with the first row of the state to produce each plaintext block. For the final step to produce or reproduce the tag, the last 2 rows are XOR'd with the key after the finalization phase to produce the tag *

There are 3 major steps in each round of the permutation operation of this 5 row (320 bit) Internal State:

Step 1: Round constant addition (p_C):

As the first step of the permutation operation, a unique 8-bit constant is XOR'd with the third row x_2 of the state in each permutation round. Given below are the constants that are XOR'd with row x_2 in each round (rounds 0-11 in p^{12} (a=12)) and rounds 0-5 in p^6 (b=6)).

$$x_2 \leftarrow x_2 \oplus c_r \, .$$

Table 4: The round constants c_r used in each round i of p^a and p^b.

p^{12}	p^8	p^6	Constant c_r	p^{12}	p^8	p^6	Constant c_r
0			00000000000000000f0	6	2	0	0000000000000000096
1			00000000000000000e1	7	3	1	0000000000000000087
2			00000000000000000d2	8	4	2	0000000000000000078
3			00000000000000000c3	9	5	3	0000000000000000069
4	0		00000000000000000b4	10	6	4	000000000000000005a
5	1		00000000000000000a5	11	7	5	000000000000000004b

Figure 4: Constants Added Lookup [3]

In each round, the 8-bit constant when XOR'd with the state to modify the third row (x_2) of the state, as shown below:

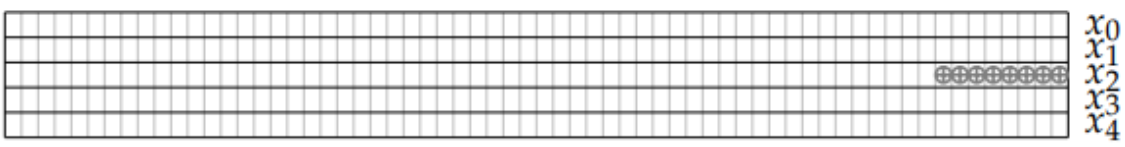

Figure 5: Round constant addition to state [3]

For the first round, the associated constant from Figure 4 is XOR'd with the row x_2 of the internal state. This modified state is fed to the next operation. It is important to note that this step is to introduce confusion to the state [4].

Step 2: Substitution using 5-bit S-box (p_S):

To the modified state from round constant addition, we perform S-box substitution to provide a non-linear transformation to the state and provide confusion. As can be seen in figure 6 below, the 320-bit state can also be looked as 64 5-bit values (where the 5 bits are the (same) i^{th} column in each of the 5 rows):

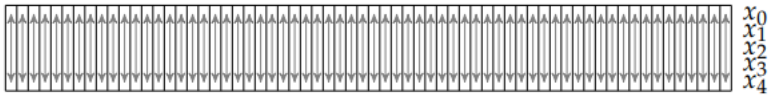

Figure 6-bit representation of State [3]

As seen above in figure 6, there are 64 such 5-bit values that make the state (denoted by vertical arrows in Figure 6). As mentioned above, each 5-bit value contains the values appended in order of all the 5 rows for a fixed column. There are 64 columns in total, giving back the 320 bits. Since the values are 5 bits, the range of the resultant values from appending each row (for (each) fixed column) lies from 0 to 31. Given below in Figure 7A are the 5-bit S(x) (S-box substitution) values. For the resultant 5-bit value by appending each row (for a fixed column), the resultant S(x) is discovered from this lookup table.

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	1f
$\mathcal{S}(x)$	4	b	1f	14	1a	15	9	2	1b	5	8	12	1d	3	6	1c	1e	13	7	e	0	d	11	18	10	c	1	19	16	a	f	17

Figure 7A: 5-bit S-box Lookup table [3]

The S(x) value now replaces x. i.e., the 5-bit value by appending each row for a fixed column is replaced by the 5-bit S box value. This step is done for a total of 64 times (columns), modifying the internal state to provide non-linearity and confusion.

It is important to note that storing such a lookup table will be very costly (energy wise [5]) in a lightweight device. Therefore, the authors of ASCON provide an equivalent set of logical operations for modification of each 5-bit state (bit-sliced implementation of S-box) as shown in figures 7B and 7C below that can be performed instead:

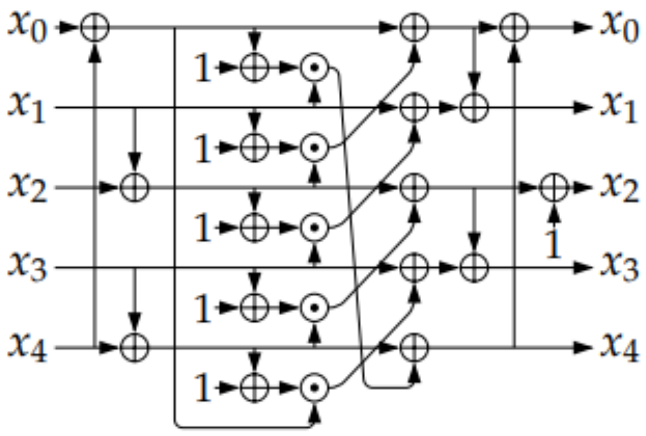

Figure 7B: 5-bit S-box equivalent operations [3]

```
x0 ^= x4;    x4 ^= x3;    x2 ^= x1;
t0  = x0;    t1  = x1;    t2  = x2;    t3  = x3;    t4  = x4;
t0 =~ t0;    t1 =~ t1;    t2 =~ t2;    t3 =~ t3;    t4 =~ t4;
t0 &= x1;    t1 &= x2;    t2 &= x3;    t3 &= x4;    t4 &= x0;
x0 ^= t1;    x1 ^= t2;    x2 ^= t3;    x3 ^= t4;    x4 ^= t0;
x1 ^= x0;    x0 ^= x4;    x3 ^= x2;    x2 =~ x2;
```

Figure 7C: 5-bit S-box equivalent operations in array form[3]

This equivalent implementation using logical operators also protects against a side channel attack where the attacker can note the difference in power during these lookup operations to recover some parts of the key [5]. These logical operations reduce the naïve discretization of a large lookup table lookup process. It is easy to see that instead of performing each operation 64 times, in this case the rows themselves are modified: so each operation in Figure 7C is performed on the 64 columns in parallel (64 parallel S-box operations [5]), providing a huge benefit in speed. The report will use the equations from Figure 7C directly in the bit-slice implementation of the S-Box substitution.

Step 3: Linear diffusion (p_L):

The modified state from the S-Box diffusion Step (Step 2) now goes through Linear diffusion to increase diffusion (spread influence of 1 bit of change across several bits). In this step, each row is modified using XOR operations and right rotation ">>>>" operations. i.e., rotating n bits to the right. As seen in Figure 8, for operation on row x_9, x_0 is XOR'd with (x_0 rotated 19 bits to the right) XOR'd with (x_0 rotated 29 bits to the right). For operation on row x_1, x_1 is XOR'd with (x_1 rotated 61 bits to the right) XOR'd with (x_1 rotated 39 bits to the right). For row x_2, x_2 is XOR'd with (x_2 rotated 1 bit to the right) XOR'd with (x_2 rotated 6bits to the right). Similarly, for operation on row x_3, x_3 is XOR'd with (x_3 rotated 10 bits to the right) XOR'd with (x_3 rotated 17 bits to the right). Lastly, for operation on row x_4, x_4 is XOR'd with (x_4 rotated 7 bits to the right) XOR'd with (x_4 rotated 41 bits to the right). Each row is hence modified to spread the influence of a single bit.

$$x_0 \leftarrow \Sigma_0(x_0) = x_0 \oplus (x_0 \ggg 19) \oplus (x_0 \ggg 28)$$
$$x_1 \leftarrow \Sigma_1(x_1) = x_1 \oplus (x_1 \ggg 61) \oplus (x_1 \ggg 39)$$
$$x_2 \leftarrow \Sigma_2(x_2) = x_2 \oplus (x_2 \ggg 1) \oplus (x_2 \ggg 6)$$
$$x_3 \leftarrow \Sigma_3(x_3) = x_3 \oplus (x_3 \ggg 10) \oplus (x_3 \ggg 17)$$
$$x_4 \leftarrow \Sigma_4(x_4) = x_4 \oplus (x_4 \ggg 7) \oplus (x_4 \ggg 41)$$

Figure 8: Linear diffusion operation on each row (XOR and right shift) [3]

The figure 9 below shows that each diffusion transformation is applied to each of the 5 rows:

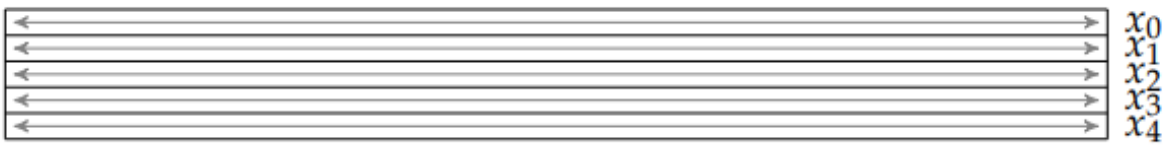

Figure 9: Linear diffusion performed on each row [3]

Steps 1-3 are performed in order 1,2,3 ($p=p_L \cdot p_S \cdot p_C$) either 12 or 6 times (rounds), depending on the current phase.[3]

Notes:

- For initialization and finalization phase, the steps 1-3 of the permutation operations are applied a=12 times (12 rounds). For step 1, Figure 4 (Lookup table) is used to decide the 8-bit constant to be XOR'd with third row x_2 for each of the 12 steps (1 unique constant corresponding to the round number). For the Associated Data and Plaintext phase, steps 1-3 of the permutation operations are applied b=6 times (6 rounds). For step 1 out of 3 here, Figure 4 (Lookup table) is used to decide the 8-bit constant to be XOR'd with row x_2 for each of the 6 steps, (1 constant corresponding to the round number). It is important to note that the last 6 constants of the 12 constants are used as the 6 constants.
- During the Plaintext phase for encryption, the plaintext block is XOR'd with the first row of the state to produce each ciphertext block. This is done after each b=6 rounds of this permutation operation, t times. During the Ciphertext phase for decryption, the Ciphertext block is XOR'd with the first row of the state to produce each plaintext block. This is done after each b=6 rounds of this permutation operation, t times. For the final step to produce or re-create the tag, the last 2 rows of the final state are XOR'd with the key after the finalization phase to produce the tag

III. TEST SIMULATION: IMPLEMENTATION OF ASCON 128 V1.2 (ENCRYPTION AND DECRYPTION) IN C LANGUAGE

***Disclaimer**: Cihangir Tezcan (Middle East Technical University)'s course's (CSEC 508: Applied Cryptanalysis) YouTube page [5] was referred to in this section for guidance regarding a custom implementation of ASCON 128 in C [5]. 3 mistakes were identified and noted to the YouTube Author in the comments section. The already available git repository for ASCON implementation by the authors of ASCON was referenced for guidance, but not directly forked for implementation. [6]. **The code included in the Appendix** is referenced to Cihangir Tezcan's video and to the GitHub repository of the implementation files by the authors for their guidance [6].*

***Limitation of scope**: In this paper, both encryption and decryption functions and authentication functions are implemented from scratch. In this implementation of ASCON, the Associated data phase and Hashing Functions are excluded, and the focus is solely on the encryption, decryption and tag creation/recreation. This is because the authors believe that the Associated Data functions and Hash function can be produced using privity with the other functions, and performed in an analogous manner to the encryption, decryption and authentication functions. Therefore, they are not included in the scope. The authors also realized that the assurance of authenticity and originality can be increased by using Associated Data. For example, if the signature (concatenated as blocks) is included as associated data, it can then be verified by the receiver, which adds an extra layer of proving the source of the message. However, since the functions can be produced in an analogous manner, associated data and hashing are not in the scope. The implementation was done on VS Code in C.*

The ASCON pseudocode provided in the original submission to NIST is referenced to create a custom implementation in C:

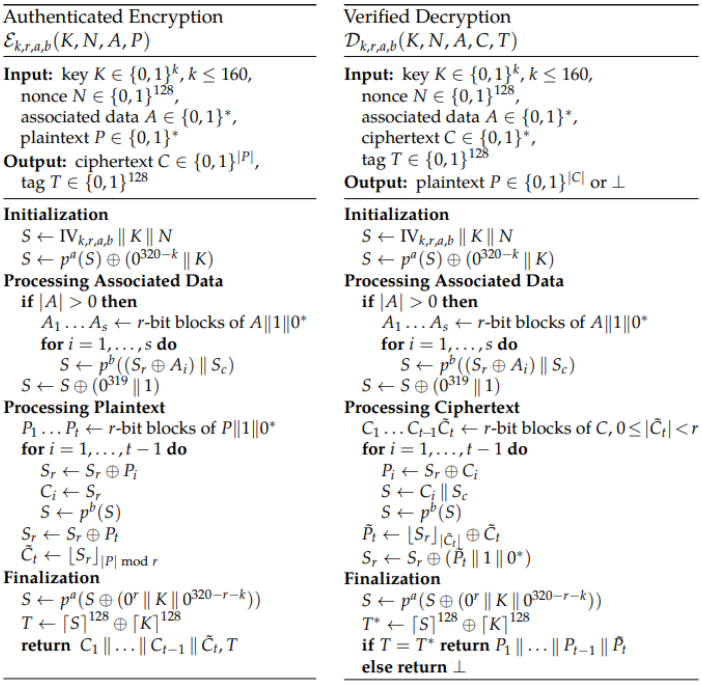

Figure 10: ASCON implementation pseudocode [3]

III-A: Parameter choice and state declaration:

The 64-bit IV used in this implementation is the fixed IV For ASCON 128 v1.2 as provided in the original submission to NIST: "80400c0600000000".

In a sample run of the code, they 128-bit secret key and Nonce were generated using OpenSSL command "openssl rand -hex 8" used twice (2X64=128 bits) for keys and nonce both. It must be noted that this command generates a CSRNG. It is important to note that once this key is established, it is kept the same for all transactions between the given encryptor and decryptor, and exchanged between the encryptor and decryptor beforehand by using key encapsulation (public key based secret key exchange)

```
b64 IV=0x80400c0600000000;
b64 key[2]={0xece2cafb8397c3c7,0x075b889de2e32b69};
b64 nonce[2]={0xe85bd7b5eca7924e,0x1d2691e5bf4c40c3};
```

Figure 11: Parameters: Random key, Random nonce and fixed IV

In a sample run of the code (which is capable to use any number of plaintext blocks), a plaintext message of 3 r=64 bits plaintext blocks were chosen as follows:

```
b64 plain_text[]={0x1234567890abcdef,0xabcdef1234567890,0xabcdef9876543210};
```

Figure 12: Parameters: Random key, Random nonce and fixed IV

The state was defined/declared to be 5 rows of 64-bits each, as discussed in "Figure 3: ASCON 320-bit Internal State [3]":

```
typedef unsigned __int64 b64;
//Define the internal state
b64 internal_state[5]={0}, t[5]={0};
```

Figure 13: Declaration of internal State

III-B: Permutation operation (p) implementation:

To implementation the permutation operations, a loop was setup to run (x times) the internal state through Constant Addition, S-box Permutations and Linear Diffusion in order. The loop executes x times, where x defines how many times the operation is to be performed (a=12 or b=6). i.e., there are two inputs to the function permutations: the current internal state and x=a=12 or x=b=6, the number of required rounds of the permutation operation.

For function round_constant, the parameters passed are internal state, current round /loop count (i) and number of overall rounds/loop's upper limit(x). This is done because the constant to be added changes based on which round is currently being performed and whether it is a 6 or 12 round operation. For functions s_box and linear_diffusion, the internal state is passed as the argument.

```
void permutations(b64 internal_state[5], int x)
{
    for(int i=0;i<x;i++)
    {
        round_constant(internal_state,i,x);
        s_box(internal_state);
        linear_diffusion(internal_state);
    }
}
```

Figure 14: Main permutation loop

The next section of the report contains the individual functions "round_constant", "s_box" and "linear_diffusion" mentioned in Figure 14.

Round constant addition:

Round constant is now implemented by declaring the 12 constants as mentioned in "Figure 4: Constants Added Lookup [3]".

```
b64 constants[12]={0xf0,0xe1,0xd2,0xc3,0xb4,0xa5,0x96,0x87,0x78,0x69,0x5a,0x4b};
```

Figure 15: Constants to be added to row 3

As mentioned before, in round_constant, the parameters passed are internal state, current round /loop count (i) and number of rounds to be performed. This is done because the constant to be added changes based on which round is currently being performed and whether it is a 6 or 12 round operation.

Next, the row 3 (x_2) is modified based on these two criteria: i and x. The indexing formula used is 12-x+i, which first subtracts 12 or 6 depending on the total required number of rounds to give either 0 (for 12 rounds) or 6 (for 6 rounds) as the first index to be used from the lookup table in Figure 15. Then, i is added to this 'first index' to get the 0'th,1st 2nd ,3rd... 11th element (for a=12 round operations) from the lookup table based on which of the 12 rounds is being performed, or to get the 6th, 7th ...11th element (for b=6 round operations) from the lookup table based on which of the 6 rounds is being performed. The 3rd row x_2 is XOR'd with this constant a=12 or b=6 times (using the for loop in Figure 14), one round after another depending on the function call for the specific required phase (this function is called x=a or x=b times from the outer loop in Figure 14).

```
//Add constants to row 2
void round_constant(b64 internal_state[5],int i,int x)
{
    //add to row 2 the constant depending on the current round and #rounds
    internal_state[2]=internal_state[2]^constants[12-x+i];

}
```

Figure 16: round constant operation implementation

S Box substitution:

The equivalent internal state after constant addition is to be used for S-box substitutions. To implement the 5-bit S-box substitutions, the equivalent logical operations were used directly from Figure 7C instead of the lookup table to modify the 5-bit "states" (bit-sliced implementation of S-box). It is easy to see that instead of performing 64-operations in series to change 64 5-bit "states", the rows are used equivalently in logical operations to perform 64-operations in parallel. As mentioned before, this allows for parallel 64 column operations (using many row manipulations), saves energy for lightweight IOT devices and prevents a differential power-based (DPA) side channel attack [5].

The commands are directly copied from the main submission of ASCON v1.2 [3], and used in the array form:

```
//S box implemtation
void s_box(b64 x[5])
{
//single pass applied to 64 different columns
//modification of 5 bit states taken from the official document directly
x[0] ^= x[4]; x[4] ^= x[3]; x[2] ^= x[1];
t[0] = x[0]; t[1] = x[1]; t[2] = x[2]; t[3] = x[3]; t[4] = x[4];
t[0] =~ t[0]; t[1] =~ t[1]; t[2] =~ t[2]; t[3] =~ t[3]; t[4] =~ t[4];
t[0] &= x[1]; t[1] &= x[2]; t[2] &= x[3]; t[3] &= x[4]; t[4] &= x[0];
x[0] ^= t[1]; x[1] ^= t[2]; x[2] ^= t[3]; x[3] ^= t[4]; x[4] ^= t[0];
x[1] ^= x[0]; x[0] ^= x[4]; x[3] ^= x[2]; x[2] =~ x[2];
}
```

Figure 17: S-box substitution operations implementation

Linear Diffusion

To perform linear diffusion, each row of the internal state received from S-box substitution need to be rotated (many times) and the original row (for each row of the state) is XOR'd with two (different) rotations using the logical operations as mentioned in Figure 8.

First, the rotate bits function ("a" many bits) is designed using right shift operators. There is a slight modification that allows that left most bits are correct as shown below:

```
//Rotation operation function for the linear diffusion layer (rotate a bits)
b64 rotate_bits( b64 x,int a)
{
    b64 rotated_a_bits;
    /*Provide correct rotate operation
    make sure that left most a bits correct, so we use left shift XORd
    */
    rotated_a_bits=(x>>a) ^ (x<<(64-a));
    return rotated_a_bits;
}
```

Figure 18: A function to rotate the internal state “a bits”

Next, the “rotate_bits” function is used twice for each row, and these two (a many times) rotated versions of each row are XOR’d with each other and also XOR’d with the original row. This row operations are done on each of the 5 rows (64 bit rows) as shown below:

```
void linear_diffusion(b64 state[5])
{
    //Modification of row 1:  Σ0(x0) = x0 ⊕ (x0 ≫ 19) ⊕ (x0 ≫ 28)
    b64 t1,t2;
    t1=rotate_bits(internal_state[0],19);
    t2=rotate_bits(internal_state[0],28);
    internal_state[0]^=t1^t2;

    //Modification of row 2:Σ1(x1) = x1 ⊕ (x1 ≫ 61) ⊕ (x1 ≫ 39)
    b64 t3,t4;
    t3=rotate_bits(internal_state[1],61);
    t4=rotate_bits(internal_state[1],39);
    internal_state[1]^=t3^t4;

    //Modification of row 3 Σ2(x2) = x2 ⊕ (x2 ≫ 1) ⊕ (x2 ≫ 6)
    b64 t5,t6;
    t5=rotate_bits(internal_state[2],1);
    t6=rotate_bits(internal_state[2],6);
    internal_state[2]^=t5^t6;

    //Modification of row 4 Σ3(x3) = x3 ⊕ (x3 ≫ 10) ⊕ (x3 ≫ 17)
    b64 t7,t8;
    t7=rotate_bits(internal_state[3],10);
    t8=rotate_bits(internal_state[3],17);
    internal_state[3]^=t7^t8;

    //Modification of row 5 Σ4(x4) = x4 ⊕ (x4 ≫ 7) ⊕ (x4 ≫ 41)
    b64 t9,t10;
    t9=rotate_bits(internal_state[4],7);
    t10=rotate_bits(internal_state[4],41);
    internal_state[4]^=t9^t10;
}
```

Figure 19: Linear diffusion implementation

III-C: Encryption Initialization phase implementation:

The 64-bit IV (fixed), 128-bit key (CSRNG), 128-bitnonce (CSRNG) are defined and 3 64-bit blocks of plaintext as shown below. A discussion of the choices was made in section IIIA (Parameter choice and state declaration). At this stage, the 3 64-bit ciphertext is declared (but empty) to hold values later.

```
b64 IV=0x80400c0600000000;
b64 key[2]={0xece2cafb8397c3c7,0x075b889de2e32b69};
b64 nonce[2]={0xe85bd7b5eca7924e,0x1d2691e5bf4c40c3};
b64 plain_text[]={0x1234567890abcdef,0xabcdef1234567890,0xabcdef9876543210};
b64 cipher_text[3]={0};
```

Figure 20: Parameters: Cyber Secure Random key, Random nonce, fixed IV, chosen plaintext

Next the defaults settings of the internal state in the form IV||K||N is set up using the IV, key and nonce defined in Figure 20. i.e., first bit row of the state holds the IV, 2nd and 3rd row hold the key, 4th and 5th row hold the Nonce, discussed in the Initialization phase of section

Encryption:

```
//setting up initial internal state
internal_state[0]=IV;
internal_state[1]=key[0];
internal_state[2]=key[1];
internal_state[3]=nonce[0];
internal_state[4]=nonce[1];
initialization(internal_state,key);
```

Figure 21: Default internal state in form IV||K||N

As seen in Figure 21, the initialization function is now called with 2 arguments passed: recently setup default internal state and the secret key.

The initialization function can be seen below:

```
void initialization(b64 internal_state[5],b64 key[2])
{
    //perform the first 12 rounds of permutation
    permutations(internal_state,12);
    //XOR the key with the last 128 bits of the initial state
    internal_state[3]^=key[0];
    internal_state[4]^=key[1];
}
```

Figure 22: Initialization function implementation

As seen in the Figure 22 (above figure), the default internal state as per Figure 21 is now passed through 12 rounds of permutation operation, as setup in section III-B: (Permutation operation (p) implementation). Next, the last 2 rows of the internal state (last 128 bits of the internal state) are XOR'd with the key. The initialized internal state is now ready.

Given below is the realized initialized internal state for the current sample run (with the defined Nonce, Key, IV(fixed) as per section III-A: (Parameter choice and state declaration)), from the output of the code:

```
Encryption initialized state:
f34fb7344d9d3aaa
dda9190ba9698bcb
a2ee01d8460158f2
408fffa0eb5c340d
1c76f4af47b50dec
```

Figure 23a: Realized initialized internal state (encryption)

Given below is the function used to print the internal state at this stage, and future stages (Internal state after encryption phase, encryption finalization phase, decryption initialization phase, decryption phase and decryption finalization phase):

```
//print the current state
b64 print_current_state(b64 internal_state[5])
{
    for(int i=0;i<5;i++)
    {
        //Show the 64 hexademical state. adding 016 ensures we also print 0s on the screen
        printf("%016I64x\n",internal_state[i]);
    }
}
```

Figure 23b: Subroutine to print internal state

In the above function in Figure 23b, the internal state is looped for each of the 5 64-bit rows, and each row (64-bit) is printed. As a result, the printed internal state in Figure 23a has 5 rows (64 bit each), resulting in the 320-bit state.

Note: In this report, the Associated Data phase is skipped. This is because the authors of this paper did not want to authenticate additional data and believe that authenticating the message with its encryption/decryption is sufficient Next, the plaintext phase is implemented.

III-D: Encryption Plaintext phase implementation:

After the Initialized internal state is realized, the encryption function is called with 4 parameters: the initialized internal state, the number of plaintext blocks, the plain text blocks, and the declared but empty cipher text blocks as seen below:

```
initialization(internal_state,key);
printf("initialized state: \n");
print_current_state(internal_state);
encryption(internal_state, 3, plain_text,cipher_text);
printf("Ciphertext: %016I64x %016I64x %016I64x\n",cipher_text[0],cipher_text[1],cipher_text[2]);
```

Figure 24: Plaintext phase (encryption) function call

The actual encryption function is shown below:

```
//length is number of blocks needed to be encrypted t
void encryption(b64 internal_state[5],int length, b64 plain_text[],b64 cipher_text[])
{
    /*first ciphertext is the first row of initialized state XOR'd
    with the first plaintext block*/
    cipher_text[0]=plain_text[0]^internal_state[0];
    internal_state[0]=cipher_text[0];
    for (int i=1;i<length;i++)
    {
        //perform 6 rounds of permutation from the previous state
        permutations(internal_state,6);
        //get ith block of ciphertext
        cipher_text[i]=plain_text[i]^internal_state[0];
        //modify state to be fed to each 6 round permutations
        internal_state[0]=cipher_text[i];
    }
}
```

Figure 25: Plaintext phase (encryption) implementation

Step1 (First Plaintext block): In the above figure (Figure 25), it is observed as before that the function sees 4 arguments: the initialized internal state, the initialized internal state, the number of plaintext blocks, the plain text blocks, and the declared but empty cipher text blocks. Inside the function, the first plaintext block is XOR'd with the top row (first 64 bits) of the initialized internal state to realize the first ciphertext block. Then, the internal state's first 64 bits are modified to be equivalent to the first ciphertext block.

Step 2 (Loop): For the consequent plaintext blocks, a loop runs length-1 times, where length is the number of plaintext blocks (length=3 in this case, so the loop runs 2 more times, since the first plaintext block is already accounted for with the relevant ciphertext block and modified state). Inside each loop, first the internal state from the previous plaintext block's XOR'ing is fed to 6 rounds of the permutation operation. Next, the current (2^{nd} block in the first iteration (i=1) of the loop) ciphertext block (i^{th} ciphertext block) is realized by XOR'ing the first row (first 64 bits) of outputted internal state (outputted from 6 rounds of permutation operations) with the current (i^{th}) plaintext block (2nd block in the first iteration of loop). The first 64 bits (first row) of the internal state is now modified and set equal to the current (i^{th}) computed ciphertext block.

This loop (Step 2) is repeated entirely for the 3rd iteration (using the 3rd plaintext block to realize the 3rd ciphertext block and modify the state). The loop is then repeated for the 4th plaintext block, and so on, till the length= total number of plaintext blocks is sufficed.

In this current sample run with the chosen IV (fixed), Nonce, Secret key and chosen 3 plaintext blocks as per section III-A: (Parameter choice and state declaration)), it is easy to see that the loop runs 2 times, after the first ciphertext explicitly computed. For the current sample run, the outputted 3 ciphertext blocks are as follows:

```
Ciphertext: e17be14cdd36f745 0d57223c40eff52f ebabb8b8bd703bb7
```

Figure 26: Realized Ciphertext blocks for sample run

```
printf("Ciphertext: %016I64x %016I64x %016I64x\n",cipher_text[0],cipher_text[1],cipher_text[2]);
```

Figure 27: Commands to print Ciphertext blocks for sample run

III-E Encryption Finalization phase implementation

After receiving the resultant internal state from end of step III-D (encryption/plaintext phase), the finalization function is now called as shown below:

```
finalization_phase(internal_state,key);
```

Figure 28: Finalization phase (encryption) function call

In the above figure 28, the resultant internal state from end of step III-D (encryption/plaintext phase), as well as they secret key are passed to the finalization function.

For reference, the resultant internal state from end of step III-D (encryption/plaintext phase) is shown below for the current run:

```
State after plaintext (encryption) phase
ebabb8b8bd703bb7
37c65a95d5a45b92
93d8e0f2ac4704e1
4ddd46dbfb3bda96
4b0c994e29c44c62
```

Figure 29: Internal state after plaintext(encryption) phase

Next, the finalization function is discussed in the figure given below. It is important to note that the role of this function is to realize the tag.

```
//perform finalization phase to get the key
void finalization_phase(b64 internal_state[5], b64 key[2])
{
    //XOR the first 2 rows of c (2nd and 3rd row of the state) with the key
    internal_state[1]^=key[0];
    internal_state[2]^=key[1];
    permutations(internal_state,12);
    //at the end, the last 2 rows of the state are XOR'd with the key to produce the tag
    internal_state[3]^=key[0];
    internal_state[4]^=key[1];
}
```

Figure 30: Finalization phase (encryption) function implementation

In the beginning of this function, the 2nd and 3rd rows (first 128 bits after the state's first 64 bits (accounted by the last ciphertext)) from resultant internal state from end of step III-D (encryption/plaintext phase) are XOR'd with the 128-bit key. Then this modified internal state is fed to 12 rounds of the permutation operations. Lastly, the last 2 rows (last 128 bits) of the newly modified internal state post permutation operations are XOR'd with the 128-bit key to produce the tag.

The produced/outputted tag for this current sample run with the chosen IV (fixed), Nonce, Key and 3 plaintext blocks is:

```
Generated Tag: e359b7d823bfddec f25174c108e7d738
```

Figure 31: Tag produced for the current sample run

For reference, the observed final state after tag is shown below, for the current sample run:

```
Encryption Final state:
17efa0820f9bd149
aa3407f949d2106f
f55df52443ef0606
e359b7d823bfddec
f25174c108e7d738
```

Figure 33: Final state after authenticated encryption (tag)

III-F Decryption initialization

The encryptor now passes over the chosen IV (fixed), Nonce from step III-A (Parameter choice and state declaration). It is assumed that the secret key was already shared between the two beforehand. The 3 ciphertext blocks are also passed to the person who needs to perform the decryption. The encryptor and

decryptor share the same secret key, so key agreement must have been done long before this step. For this report, the key agreement process is not performed, and it is assumed that they keys are exchanged using key encapsulation process, with the private and public keys. This also explains the use of a Crypto Secure 128-bit random number in the original choice of the key by the encryptor.

The same can be seen below, as well as the newly declared but empty 3 decrypted plaintext blocks and the initialization function call in the decryption process:

```
//decryption
//Parameters provided by encryptor

b64 plain_text_decrypted[3]={0};
internal_state[0]=IV;
internal_state[1]=key[0];
internal_state[2]=key[1];
internal_state[3]=nonce[0];
internal_state[4]=nonce[1];
initialization(internal_state,key);
```

Figure 34: Passing over of params to decryptor, and decryption initialization call

The initialization function used by the decryptor is the same as the one used during encryption. The realized initialized state for the current sample run is shown below, and is equivalent to the encryption initialized state:

```
Decryption inititalized state
f34fb7344d9d3aaa
dda9190ba9698bcb
a2ee01d8460158f2
408fffa0eb5c340d
1c76f4af47b50dec
```

Figure 35: Decryption initialized state

III-G Decryption (ciphertext) phase

Next, the realized initialized state for the decryption process as per Figure 35 along with the number of ciphertext blocks, the cipher text blocks and the declared but empty decrypted plaintext blocks are passed to the decryption function as seen below:

```
decryption(internal_state, 3,cipher_text, plain_text_decrypted);
```

Figure 36: Decryption function call

The figure below shoes the implementation of this decryption (ciphertext phase):

```
void decryption(b64 internal_state[5],int length,b64 cipher_text[],b64 plain_text_decrypted[])
{
    //first ciphertext is the first row of initialized state XOR'd with the first plaintext block
    plain_text_decrypted[0]=cipher_text[0]^internal_state[0];
    internal_state[0]=cipher_text[0];
    for (int i=1;i<length;i++)
    {
        //perform 6 rounds of permutation from the previous state
        permutations(internal_state,6);
        //get ith block of ciphertext
        plain_text_decrypted[i]=cipher_text[i]^internal_state[0];
        //modify state to be fed to each 6 round permutations
        internal_state[0]=cipher_text[i];
    }
}
```

Figure 37: Decryption function implementation

In the figure above (figure 37), as mentioned before, the decryption initialized internal state, number of ciphertext blocks, the cipher text blocks and the declared but empty decrypted plaintext blocks are passed to the decryption function.

Step1 (First Ciphertext block): Inside the decryption function, the first ciphertext block is XOR'd with the top row (first 64 bits) of the initialized internal state to realize/decrypt the first plaintext block. Then, the internal state's first 64 bits are modified to be equivalent to the first ciphertext block (already known).

Step 2 (Loop): For the consequent plaintext blocks, a loop runs length-1 times, where length is the number of ciphertext blocks (length=3 in this case, so the loop runs 2 more times, since the first plaintext block is already accounted for with the relevant ciphertext block). Inside each loop, first the internal state from the previous ciphertext block's XOR'ing is fed to 6 rounds of the permutation operation. Next, the current (2nd block in the first iteration (i=1) of the loop) plaintext block (i^{th} plaintext block) is decrypted by XOR'ing the first row (first 64 bits) of outputted internal state (outputted from 6 rounds of permutation operations) with the current (i^{th}) ciphertext block (2nd block in the first iteration of loop). The first 64 bits (first row) of the internal state is now modified and set equal to the current (i^{th}) ciphertext block (already known).

This loop (Step 2) is repeated entirely for the 3rd iteration (using the 3rd ciphertext block to decrypt/realize the 3rd plaintext block). The loop is then repeated for the 4th ciphertext block, and so on, till the length= total number of ciphertext blocks is sufficed.

In this current sample run with the IV (fixed), Nonce, Secret key and 3 shared ciphertext blocks shared with the person who wishes to decrypt, it is easy to see that the loop runs 2 times, after the first plaintext block is explicitly computed. For the current sample run, the outputted 3 decrypted plaintext blocks are as follows:

```
Decrypted plaintext: 1234567890abcdef abcdef1234567890 abcdef9876543210
```

Figure 38: Realized/ decrypted plaintext blocks for sample run

It is important to note that these 3 decrypted plaintext blocks are the same as the original plaintext blocks chosen by the encryptor.

```
printf("Decrypted plaintext: %016I64x %016I64x %016I64x\n",plain_text_decrypted[0],plain_text_decrypted[1],plain_text_decrypted[2]);
```

Figure 39: Commands to print decrypted plaintext blocks for sample run

III-H Decryption Finalization phase implementation

After receiving the resultant internal state from end of step III-G (decryption/ciphertext phase), the decryption finalization function is now called as shown below:

```
finalization_phase(internal_state,key);
```

Figure 40: Finalization phase (decryption) function call

In the above figure 40, the resultant internal state from end of step III-G (decryption/ciphertext phase), as well as they secret key are passed to the finalization function.

For reference, the resultant internal state from end of step III-D (encryption/plaintext phase) is shown below for the current run:

```
State after decryption phase
ebabb8b8bd703bb7
37c65a95d5a45b92
93d8e0f2ac4704e1
4ddd46dbfb3bda96
4b0c994e29c44c62
```

Figure 41: Internal state after decryption phase

The decryption finalization function is the same finalization function used for encryption.

In the beginning of this function, the 2nd and 3rd rows (first 128 bits after the state's first 64 bits (accounted by the last ciphertext)) from resultant internal state from end of step III-G (decryption/ciphertext phase) are XOR'd with the 128-bit key. Then this modified internal state is fed to 12 rounds of the permutation operations. Lastly, the last 2 rows (last 128 bits) of the newly modified internal state post the permutation operations are XOR'd with the 128-bit key to re-compute the tag. The re-created/ re-computed outputted tag for this current sample run with shared ciphertext blocks is:

```
Re computed Tag: e359b7d823bfddec f25174c108e7d738
```

Figure 42: Recomputed tag

It is important to note that this re-computed tag is exactly the same as the tag generated by the encryptor. Therefore, it is easy to say that the sender is now authenticated. Therefore, the recovered decrypted plaintext blocks are accepted by the person decrypting. The communication is complete and successful.

For reference, the observed final state after tag re computation is shown below, for the current sample run:

```
Decryption final state:
17efa0820f9bd149
aa3407f949d2106f
f55df52443ef0606
e359b7d823bfddec
f25174c108e7d738
```

Figure 43: Final state after authenticated encryption (re computed tag)

It is also a good confirmation to compare the final states after encryption and decryption (just for reference). Indeed, they are the same. Note that this is just done for the purpose of this report and the post encryption final state's information will not be available to the person performing decryption.
The entire working code for ASCON 128 v1.2 in C for the encryption decryption processes are included in the appendix.

IV. TEST (SIMULATION) RESULTS

As mentioned before, given below are the chosen parameters for ASCON implementation simulation run. Ii.e., IV (fixed), CSRNG Nonce and Key (128 bits) and chosen 3 blocks of plaintext:

```
b64 IV=0x80400c0600000000;
b64 key[2]={0xece2cafb8397c3c7,0x075b889de2e32b69};
b64 nonce[2]={0xe85bd7b5eca7924e,0x1d2691e5bf4c40c3};
b64 plain_text[]={0x1234567890abcdef,0xabcdef1234567890,0xabcdef9876543210};
b64 cipher_text[3]={0};
```

Figure 44: Parameter choice

Given below are the 3 calculated ciphertext blocks for the 3 plaintext blocks at the end of the plaintext encryption phase. This ciphertext is passed to the person who needs to decrypt:

```
Ciphertext: e17be14cdd36f745 0d57223c40eff52f ebabb8b8bd703bb7
```

Figure 45: ciphertext blocks

Next, shown below is the generated tag for the given parameter choice at the end of the finalization phase. This tag, along with the nonce and IV (fixed) is passed over to the decryptor:

```
Generated Tag: e359b7d823bfddec f25174c108e7d738
```

Figure 46: Generated tag

For decryption, first the decrypted plaintext is compared to the chosen plaintext (for the sake of verification of this algorithm). No difference is realised between the two. I.e., The algorithms work well to decrypt the plaintext exactly, as it should. This original chosen plaintext is **not available** to the person decrypting. It is seen that the decryptor can easily get back the plaintext blocks with the passed-over parameters from the encryptor (knowledge of the shared key beforehand, shared random nonce, (fixed)IV, and calculated ciphertext blocks and tag).

It must be noted that at this stage, even though the decryptor has received the decrypted plaintext, he does not know for sure whether it is correct since he does not have access to the originally chosen plaintext. The way for him to authenticate these plaintext messages is to recompute the tag and compare it with the tag given to him by the encryptor.

```
b64 plain_text[]={0x1234567890abcdef,0xabcdef1234567890,0xabcdef9876543210};
Decrypted plaintext: 1234567890abcdef abcdef1234567890 abcdef9876543210
```

Figure 47: Decrypted plaintext compared to chosen plaintext (No difference)

The tag is now recomputed by the person performing decryption, and that person compares this re computed tag with the tag passed over to him by the encyptor. As can be seen below, the person performing decryption confirms that the recomputed tag is the **same** as the tag received initially (sent by the encryptor), and so the decrypted messages are accepted, and the messages are authenticated. The process is completed with success! No error is sent back.

```
Generated Tag: e359b7d823bfddec f25174c108e7d738
Re computed Tag: e359b7d823bfddec f25174c108e7d738
```

Figure 48: Comparison of the generated tag and re computed tag (No difference)

For completion and verification of the implementation by the authors, it worthwhile to compare the states during encryption and decryption at different stages. It is to be noted that this is only done to verify the functionality of the program and not a step performed by the participants. In the figures 49, 50 and 51, it can be verified that the internal states for both the encryption and decryption processes (initialized state, state after encryption/decryption phase and finalized state) are the **same**:

Encryption initialized state:	Decryption inititalized state
f34fb7344d9d3aaa	f34fb7344d9d3aaa
dda9190ba9698bcb	dda9190ba9698bcb
a2ee01d8460158f2	a2ee01d8460158f2
408fffa0eb5c340d	408fffa0eb5c340d
1c76f4af47b50dec	1c76f4af47b50dec

Figure 49: Comparison of the encryption and decryption initialized states (no difference)

State after plaintext (encryption) phase	State after decryption phase
ebabb8b8bd703bb7	ebabb8b8bd703bb7
37c65a95d5a45b92	37c65a95d5a45b92
93d8e0f2ac4704e1	93d8e0f2ac4704e1
4ddd46dbfb3bda96	4ddd46dbfb3bda96
4b0c994e29c44c62	4b0c994e29c44c62

Figure 50: Comparison of internal states after encryption and decryption phases (No difference)

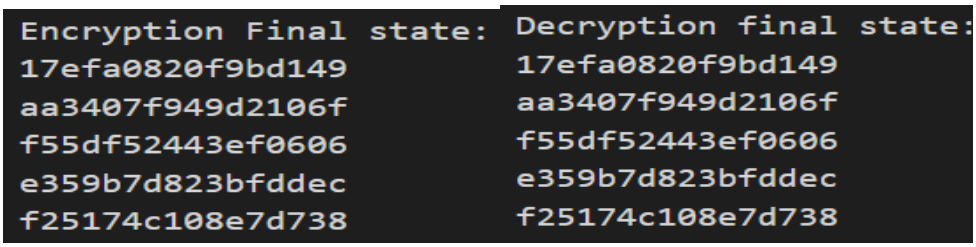

Figure 51: Comparison of final states after encryption and decryption finalization phase (No difference)

V. CONCLUSION FROM RESULTS

It can be concluded from the section II and III of this report that ASCON is a lightweight authenticated encryption algorithm that can be easily implemented in C to encrypt/decrypt and to authenticate the messages using tag. In ASCON, The 64-bit IV is fixed by the authors for ASCON 128 v1.2, 128-bit secret key is generated shared between two participants who wish to encrypt/decrypt (potentially generated using a Crypto secure Random number and shared using key encapsulation: technique involving public key crypto to exchange the key securely, or using a safe communication channel/third party). The 128-bit NONCE is generated randomly each time the encryptor wants to send a message, and shared each time securely with the person who needs to decrypt the message. The encryptor encrypts the message (plaintext blocks) using the code provided in the Appendix and passes the NONCE along with the ciphertext blocks and the tag to the receiver. (Assuming the secret key is already shared and knowing that the IV is fixed in known to both participants). Using this information, the receiver decrypts the plaintext blocks and also re computes the tag to ensure that the re computed tag matches the received tag. If the tags match, the receiver accepts the decrypted plaintext messages. Otherwise, the receiver sends an error to the encryptor.

VI. DISCUSSIONS

Given below are some properties of ASCON, as stated in the original paper [3]:

1. ‘Single pass’ [3]: Allows message encryption as well as tag generation in a single pass over the data. ASCON similarly allows decryption and tag re-creation in a single pass over the data. [3]
2. “Online”, “Lightweight” and fast in “Software and hardware” [3] [5]
3. ‘Inverse Free’ [3]: Since both encryption and decryption are in one direction [3], there is no need to perform expensive inverse for decryption.
4. Side channel protections [3]: No lookup tables are used, protecting this algorithm from the naïve differential power side channel attacks.

VI-A DISCUSSIONS ON PERFORMANCE COMPARISON

The authors of this paper conducted a comparison of speed (time) on ASCON 128 (custom implementation in C) v/s AES 128 in GCM mode (using openSSL) for the authenticated encryption process using the same IV, Key and the plaintext as seen in section “III-A: Parameter choice and state declaration” (sample run for implementation of ASCON in C) on one of our laptops.

The specifications of the (Windows) Laptop used were:

- Processor: 11th Gen Intel(R) Core(TM) i7-1195G7 @ 2.90GHz 2.92 GHz
- RAM: 16.0 GB (15.8 GB usable)
- System type: 64-bit operating system, x64-based processor

Using openSSL, the following shell script (.ps1) script was run on Windows Powershell to average the time taken for authenticated encryption using AES 128 in GCM mode, of the same plaintext, with the same IV and key (to produce a 8 byte tag) used during sample run of implementation of ASCON in C (section III-A):

```
$number = 100
$TotTime = 0

for ($i = 0; $i -lt $number; $i++) {
    $measure = Measure-Command { openssl enc -aes-128-gcm -in plaintext.txt -out ciphertext.bin `
    -iv 80400c0600000000 -K ece2cafb8397c3c7075b889de2e32b69 -nopad -plaintext -a -taglen 8 }
    $TotTime += $measure.TotalMilliseconds
}

$averageTime = $TotTime / $number
Write-Output "Average time of authenticated encryption process (AES 128 GCM): $averageTime miliseconds"
```

Figure 52: Shell command to run openSSL 100 times to average the time taken for authentication encryption

The plaintext.txt contains the same plaintext as was used during sample run of implementation of ASCON in C (section III-A):

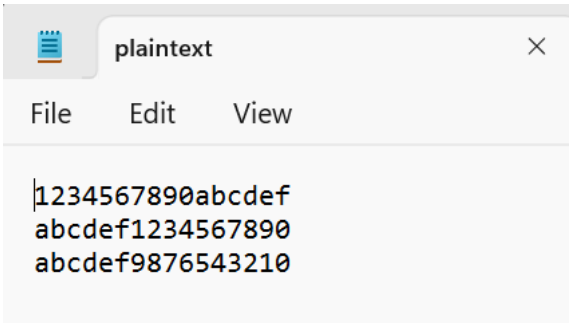

Figure 53: Plaintext (plaintext.txt) used in Figure 52

The result of the averaged results on our Laptop using openSSL was as follows:

```
Average time of authenticated encryption process (AES 128 GCM): 30.694167 miliseconds
```

Figure 54: Average time for authenticated encryption (AES 128 GCM) using openSSL

Next, using our custom implementation of ASCON in C, we measured on the same laptop the average time taken (averaged over 100 runs) for authenticated encryption process (not decryption and tag creation: compare the equivalent operation performed on openSSL for AES 128 GCM). The following code was used to measure execution time:

```c
//Main loop
void main() {


    double total_time = 0.0;
     for (int i = 0; i < 100; i++)
    {
        clock_t start_time = clock();
        // Perform the operation to be timed here
        b64 IV=0x80400c0600000000;
        b64 key[2]={0xece2cafb8397c3c7,0x075b889de2e32b69};
        b64 nonce[2]={0xe85bd7b5eca7924e,0x1d2691e5bf4c40c3};
        b64 plain_text[]={0x1234567890abcdef,0xabcdef1234567890,0xabcdef9876543210};
        b64 cipher_text[3]={0};
        //setting up initial internal state
        internal_state[0]=IV;
        internal_state[1]=key[0];
        internal_state[2]=key[1];
        internal_state[3]=nonce[0];
        internal_state[4]=nonce[1];
        initialization(internal_state,key);
        encryption(internal_state, 3, plain_text,cipher_text);
        finalization_phase(internal_state,key);
        usleep(2000); // Sleep for 2 milliseconds
        clock_t end_time = clock();
        // Calculate overall duration in milliseconds and output it (subtract the 2 ms sleep)
        double duration = ((double)(end_time - start_time) / (CLOCKS_PER_SEC / 1000))-2 ;
        total_time += duration;

     }
     double average_time = total_time / 100;
    printf("Average time of authenticated encryption process (ASCON 128) %f milliseconds\n", average_time);
}
```

Figure 55: Code to measure Average time for authenticated encryption (ASCON 128) using C

As seen above, the decryption and tag re-creation steps are omitted from the original code, and the code (main function) is slightly modified to run 100 times to measure average time taken for execution of those functions. The necessary headers are included in the code. The output time for authenticated encryption process of ASCON in C over 100 runs, for our laptop was as follows:

```
Average time of authenticated encryption process (ASCON 128) 13.410000 milliseconds
```

Figure 56: Average time for authenticated encryption (ASCON 128) using custom C implementation

Note: It is easy to see that the same IV, key and plaintext blocks are used for the comparison between AES 128 GCM mode (openSSL) and ASCON 128 (in C). This "average time of execution for authenticated encryption" version of the main function in C as in Figure 55 is also added in the code that is included in the Appendix. This version of main is commented out and can be used instead of the original main function (used for encryption, decryption, and authentication) to reproduce the results (average time) as shown in Figure 56.

It is interesting to note that as opposed to 30.69 milliseconds (using AES 128 in GCM mode), ASCON 128 takes 13.41 milliseconds for the authenticated encryption (encryption and tag creation) process. **In our simulation experiment, there is a increase in speed of over 2.28 times by using ASCON 128 over AES 128 in GCM mode (for authenticated encryption),** notwithstanding the fact that more efficient

equivalent instruction sets were used in openSSL for AES 128, since it is quite established and benchmarked already. Compared to ASCON, AES is a clear win in terms of speed!

VI-B DISCUSSIONS ON PERFORMANCE COMPARISON (ARM Cortex M3 (results taken from [7])):

On running tests on different lightweight cryptography algorithms on ARM Cortex M3 (Results taken from https://rweather.github.io [7]) running at 84MHz], a comparison was mode on the speed of each algorithm as compared to ChaChaPoly (Authenticated Encryption with Associated Data) [7]. Among the top 10 finalists of NIST Lightweight cryptography contest, ASCON-128 (ASCON-128a) stood third, following Sparkle and Xoodyak. Sparkle provided a 100% improvement in speed and both ASCON and Xoodyak provide ~60% increase in speed when measured against ChaChaPoly stream ciphers [7].

Algorithm	Key Bits	Nonce Bits	Tag Bits	Encrypt 128 bytes	Decrypt 128 bytes	Encrypt 16 bytes	Decrypt 16 bytes	Average
COMET-128_CHAM-128/128[2] (*)	128	128	128	1.57	1.56	2.91	2.69	2.05
Schwaemm128-128 (SPARKLE) (*)	128	128	128	1.60	1.58	2.84	2.39	2.01
COMET-64_SPECK-64/128 (*)	128	120	128	1.42	1.43	2.86	2.75	1.94
Schwaemm256-128 (SPARKLE) (*)	128	256	128	1.74	1.63	1.90	1.93	1.80
ASCON-128a (*)	128	128	128	1.86	1.70	1.80	1.78	1.78
SATURNIN-Short[1]	256	128	256			1.82	1.66	1.73
Schwaemm192-192 (SPARKLE) (*)	192	192	192	1.47	1.50	1.98	1.81	1.68
Xoodyak (*)	128	128	128	1.66	1.51	1.73	1.60	1.62
ASCON-128 (*)	128	128	128	1.54	1.44	1.78	1.68	1.61
ASCON-80pq (*)	160	128	128	1.52	1.43	1.71	1.65	1.57

Figure 57: Comparison of different lightweight crypto algorithms against ChaCha20 on ARM Cortex: Results taken from https://rweather.github.io [7]

Next, Hashing is compared against BLAKE2 (Results taken from https://rweather.github.io [7]) over different lightweight cryptography algorithms:

It can be seen from Figure 58 below that among the top 10 Finalists of Lightweight Cryptography contest, ASCON stood third after Sparkle and Xoodyak. While Sparkle and Xoodyak performed hashing in approximately the same time as BLAKE2, ASCON is not very slow either, with around 50% speed as compared to BLAKE2 [7]:

Algorithm	Hash Bits	1024 bytes	128 bytes	16 bytes	Average
Esch256 (SPARKLE) (*)	256	0.89	0.78	1.50	1.06
Xoodyak (*)	256	0.71	0.65	1.43	0.93
GIMLI-24-HASH (*)	256	0.54	0.47	0.86	0.62
ASCON-HASH (*)	256	0.51	0.41	0.63	0.52
DryGASCON128-HASH (*)	256	0.29	0.29	0.88	0.48
Esch384 (SPARKLE) (*)	384	0.45	0.37	1.50	0.47
SATURNIN-Hash	256	0.28	0.23	0.57	0.36
ACE-HASH	256	0.10	0.09	0.15	0.11
DryGASCON256-HASH	512	0.06	0.05	0.11	0.08
KNOT-HASH-256-384	256	0.05	0.04	0.07	0.05

Figure 58: Comparison of hashing speed of several lightweight cryptography algorithms against BLAKE2. Results taken from https://rweather.github.io [7]

VI-B DISCUSSIONS ON THE SECURITY OF ASCON:

Side channel attacks (Fault injection and power analysis (differential) attacks):

In the paper by Keyvan Ramezanpour et. Al[8], the authors used a “voltage glitch on ASCON implemented on a FPGA”, “injected a fault into a pair of S-box computations” and recovered 2 bits of the secret key [8]. The Ascon cipher is susceptible to biased fault attacks because of the “XORing of the secret key after finalization phase for tag generation”. [8] This attack is performed on the “bit-slice implementation of the S-box" [8]. Additionally, using the “secret key to initialize the cipher state” leaves the algorithm open to power analysis attacks [8]. The authors also demonstrate a “deep learning approach” to a power analysis attack during S Box computations, to recover the key [8].

A Note from authors of this paper regarding another possible side channel attack:

ASCON claims in its original submission to be free of lookup tables.[3] However, it is to be noted that during round constant addition step in the permutation operations, a lookup table is directly created to store the 12 constants to be referenced on each round, as seen below:

```
b64 constants[12]={0xf0,0xe1,0xd2,0xc3,0xb4,0xa5,0x96,0x87,0x78,0x69,0x5a,0x4b};
```

Figure 54: Constants lookup table during round constant addition

This lookup table can be exploited using differential power consumption or side channel information leaked during the permutation operation [4]. However, there are a few problems in launching such an attack. Firstly, the lookup table is small, consisting of only 12 constants. As a result, a successful attack would be more challenging because there are fewer values that an attacker can guess in order ascertain some portion of the key [4]. Secondly, the 8-bit constants are only XOR’d with the third row in each round. As a result, side-channel information like power consumption that an attacker could possibly gather during this process would only give them knowledge about a single row of the state than the entire state [4].

VII. FUTURE WORK

Although ASCON is currently the state of the art for lightweight cryptography, providing “lightweight”, “fast” and “single pass” “inverse free” authenticated encryptions (and Hashing) [3], it is important to note that logical operations (bit slice implementation) to perform S-box are also vulnerable to fault injection (side channel) attacks [7], Additionally, power consumption might be monitored during ciphertext execution to partially recover the key[4]. ASCON should continue to extend to include additional “masking” or “binding” techniques [4] to prevent such attacks.

VIII. FINAL CONCLUSION

ASCON is a lightweight cryptography, providing “lightweight”, “fast” and “single pass” “inverse free” authenticated encryptions (and Hashing) [3]. This algorithm can be used particularly on IOT and mobile devices [4], and lightweight environments like automotive systems [4] and provides “high cryptanalytic safety” [3], with the main feature being the absence of large lookup tables, preventing a very naïve differential power attack performed on such large lookup tables. This is indeed the group’s first go at ASCON. From this exercise, the group learnt that ASCON is very fast and performs authentication and encryption in one go, as opposed to heavy operations and separate encryption and tag generation algorithms for GCM mode using AES. The group also noted more than 2.28X increase in speed by using

ASCON 128 as compared to AES in GCM mode. Based on these results, and based on the ease of implementation in C, the group feels that ASCON is a clear winner as compared to AES!

ACKNOWLEDGEMENT

The authors of this paper would like to thank Cihangir Tezcan (Middle East Technical University) for his course's (CSEC 508: Applied Cryptanalysis) YouTube page. This helped us a great deal in implementing ASCON from scratch in C. It is to be noted that 3 mistakes were discovered in the course video and pointed to Chihangir Tezcan on his YouTube video in the comments section [5]. The authors also would like to acknowledge the original submission of ASCON v1.2 to NIST. This paper was referenced in the paper throughout for details about ASCON methodology and working. It is to be noted that the authors of this paper do not claim ownership of ASCON. The intent of the paper is to provide an understandable overview, implementation, features, performance analysis,comparison and security analysis of ASCON 128v1.2 as it was originally submitted [3]. Additionally, the authors of this paper would like to thank authors of ASCON for the GitHub repository (for implementation in C). It must be noted that the git was referenced and not forked. [5]

References:

1. https://csrc.nist.gov/Projects/lightweight-cryptography
2. https://www.researchgate.net/figure/The-encryption-of-Ascon_fig1_292962529
3. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/ascon-spec-round2.pdf
4. https://chat.openai.com/

Note: ChatGPT was asked specific questions in the sections referenced as [4].

5. Implementation of ASCON in C
6. https://github.com/ascon/ascon-c
7. https://rweather.github.io/lightweight-crypto/performance.html
8. https://csrc.nist.gov/CSRC/media/Events/lightweight-cryptography-workshop-2020/documents/papers/active-passive-recovery-attacks-ascon-lwc2020.pdf

Appendix:

Code from scratch of ASCON 128 v1.2 [5][6]:

```
/*ASCON implementation
Authors: SRIVASTAVA SHIVANG, WAN YONG AND HENG HIAN HEE
*/

//Include basic stdio.h
#include <stdio.h>
#include <time.h>
#include <sys/time.h>
#include <unistd.h>
//Ensure to use 64 bit values
typedef unsigned __int64 b64;
//Define the internal state
```

```
b64 internal_state[5]={0}, t[5]={0};
//Define constants to be added in permutation operations
b64 constants[12]={0xf0,0xe1,0xd2,0xc3,0xb4,0xa5,0x96,0x87,0x78,0x69,0x5a,0x4b};

//print the current state
b64 print_current_state(b64 internal_state[5])
{
   for(int i=0;i<5;i++)
   {
      //Show the 64 hexademical state. adding 016 ensures we also print 0s on the screen
      printf("%016I64x\n",internal_state[i]);
   }
}

//Now we work on the 3 permutation operations

//Add constants to row 2
void round_constant(b64 internal_state[5],int i,int x)
{
   //add to row 2 the constant depending on the current round and #rounds
   internal_state[2]=internal_state[2]^constants[12-x+i];

}
//S box implemtation
void s_box(b64 x[5])
{
//single pass applied to 64 different columns
//modification of 5 bit states taken from the official document directly
x[0] ^= x[4]; x[4] ^= x[3]; x[2] ^= x[1];
t[0] = x[0]; t[1] = x[1]; t[2] = x[2]; t[3] = x[3]; t[4] = x[4];
t[0] =~ t[0]; t[1] =~ t[1]; t[2] =~ t[2]; t[3] =~ t[3]; t[4] =~ t[4];
t[0] &= x[1]; t[1] &= x[2]; t[2] &= x[3]; t[3] &= x[4]; t[4] &= x[0];
x[0] ^= t[1]; x[1] ^= t[2]; x[2] ^= t[3]; x[3] ^= t[4]; x[4] ^= t[0];
x[1] ^= x[0]; x[0] ^= x[4]; x[3] ^= x[2]; x[2] =~ x[2];
}
//Rotation operation function for the linear diffusion layer (rotate a bits)
b64 rotate_bits( b64 x,int a)
{
   b64 rotated_a_bits;
   /*Provide correct rotate operation
   make sure that left most a bits correct, so we use left shift XORd
   */
   rotated_a_bits=(x>>a) ^ (x<<(64-a));
```

```
    return rotated_a_bits;
}
//This is the linear diffusion layer
void linear_diffusion(b64 state[5])
{
    //Modification of row 1:  Σ0(x0) = x0 ⊕ (x0 ≫ 19) ⊕ (x0 ≫ 28)
    b64 t1,t2;
    t1=rotate_bits(internal_state[0],19);
    t2=rotate_bits(internal_state[0],28);
    internal_state[0]^=t1^t2;

    //Modification of row 2:Σ1(x1) = x1 ⊕ (x1 ≫ 61) ⊕ (x1 ≫ 39)
    b64 t3,t4;
    t3=rotate_bits(internal_state[1],61);
    t4=rotate_bits(internal_state[1],39);
    internal_state[1]^=t3^t4;

    //Modification of row 3 Σ2(x2) = x2 ⊕ (x2 ≫ 1) ⊕ (x2 ≫ 6)
    b64 t5,t6;
    t5=rotate_bits(internal_state[2],1);
    t6=rotate_bits(internal_state[2],6);
    internal_state[2]^=t5^t6;

    //Modification of row 4 Σ3(x3) = x3 ⊕ (x3 ≫ 10) ⊕ (x3 ≫ 17)
    b64 t7,t8;
    t7=rotate_bits(internal_state[3],10);
    t8=rotate_bits(internal_state[3],17);
    internal_state[3]^=t7^t8;

    //Modification of row 5 Σ4(x4) = x4 ⊕ (x4 ≫ 7) ⊕ (x4 ≫ 41)
    b64 t9,t10;
    t9=rotate_bits(internal_state[4],7);
    t10=rotate_bits(internal_state[4],41);
    internal_state[4]^=t9^t10;
}

/*
Permutation function operations run in a loop x times
x defines how many times the permutation opeartion
is to be performed (12 or 6)
*/
void permutations(b64 internal_state[5], int x)
{
```

```
    for(int i=0;i<x;i++)
    {
        round_constant(internal_state,i,x);
        s_box(internal_state);
        linear_diffusion(internal_state);
    }
}
//Next we work on the initilation of internal state (initialization phase)
void initialization(b64 internal_state[5],b64 key[2])
{
    //perform the first 12 rounds of permutation
    permutations(internal_state,12);
    //XOR the key with the last 128 bits of the initial state
    internal_state[3]^=key[0];
    internal_state[4]^=key[1];
}
//length is number of blocks needed to be encrypted t
void encryption(b64 internal_state[5],int length, b64 plain_text[],b64 cipher_text[])
{
    /*first ciphertext is the first row of initialized state XOR'd
    with the first plaintext block*/
    cipher_text[0]=plain_text[0]^internal_state[0];
    internal_state[0]=cipher_text[0];
    for (int i=1;i<length;i++)
    {
        //perform 6 rounds of permutation from the previous state
        permutations(internal_state,6);
        //get ith block of ciphertext
        cipher_text[i]=plain_text[i]^internal_state[0];
        //modify state to be fed to each 6 round permutations
        internal_state[0]=cipher_text[i];
    }
}
void decryption(b64 internal_state[5],int length,b64 cipher_text[],b64 plain_text_decrypted[])
{
    //first ciphertext is the first row of initialized state XOR'd with the first plaintext block
    plain_text_decrypted[0]=cipher_text[0]^internal_state[0];
    internal_state[0]=cipher_text[0];
    for (int i=1;i<length;i++)
    {
        //perform 6 rounds of permutation from the previous state
        permutations(internal_state,6);
        //get ith block of ciphertext
```

```
        plain_text_decrypted[i]=cipher_text[i]^internal_state[0];
        //modify state to be fed to each 6 round permutations
        internal_state[0]=cipher_text[i];
    }
}
//perform finalization phase to get the key
void finalization_phase(b64 internal_state[5], b64 key[2])
{
    //XOR the first 2 rows of c (2nd and 3rd row of the state) with the key
    internal_state[1]^=key[0];
    internal_state[2]^=key[1];
    permutations(internal_state,12);
    //at the end, the last 2 rows of the state are XOR'd with the key to produce the tag
    internal_state[3]^=key[0];
    internal_state[4]^=key[1];
}
//Main loop
void main() {
    // Write C code here
    //kllprintf("Hello world\n");
    //define nonce, key and IV
    b64 IV=0x80400c0600000000;
    b64 key[2]={0xece2cafb8397c3c7,0x075b889de2e32b69};
    b64 nonce[2]={0xe85bd7b5eca7924e,0x1d2691e5bf4c40c3};
    b64 plain_text[]={0x1234567890abcdef,0xabcdef1234567890,0xabcdef9876543210};
    b64 cipher_text[3]={0};
    //setting up initial internal state
    internal_state[0]=IV;
    internal_state[1]=key[0];
    internal_state[2]=key[1];
    internal_state[3]=nonce[0];
    internal_state[4]=nonce[1];
    initialization(internal_state,key);
    printf("\n\nEncryption: \n");
    printf("Encryption initialized state: \n");
    print_current_state(internal_state);
    encryption(internal_state, 3, plain_text,cipher_text);
    printf("Ciphertext: %016I64x %016I64x %016I64x\n",cipher_text[0],cipher_text[1],cipher_text[2]);
    printf("State after plaintext (encryption) phase \n");
    print_current_state(internal_state);
    finalization_phase(internal_state,key);
    printf("Generated Tag: %016I64x %016I64x\n",internal_state[3],internal_state[4]);
    printf("Encryption Final state: \n");
```

```
    print_current_state(internal_state);

    //decryption
    //Parameters provided by encryptor

    b64 plain_text_decrypted[3]={0};
    internal_state[0]=IV;
    internal_state[1]=key[0];
    internal_state[2]=key[1];
    internal_state[3]=nonce[0];
    internal_state[4]=nonce[1];
    initialization(internal_state,key);
    printf("\nDecryption: \n");
    printf("Decryption inititalized state \n");
    print_current_state(internal_state);
    decryption(internal_state, 3,cipher_text, plain_text_decrypted);
                        printf("Decrypted plaintext: %016I64x %016I64x
%016I64x\n",plain_text_decrypted[0],plain_text_decrypted[1],plain_text_decrypted[2]);
    printf("State after decryption phase \n");
    print_current_state(internal_state);
    finalization_phase(internal_state,key);
    printf("Re computed Tag: %016I64x %016I64x\n",internal_state[3],internal_state[4]);
    printf("Decryption final state: \n");
    print_current_state(internal_state);
}


/*
//Use this version of main for calculating average speed of execution for authenticated encryption process
over 100 times
//Main loop
void main() {


    double total_time = 0.0;
     for (int i = 0; i < 100; i++)
    {
        clock_t start_time = clock();
        // Perform the operation to be timed here
        b64 IV=0x80400c0600000000;
        b64 key[2]={0xece2cafb8397c3c7,0x075b889de2e32b69};
        b64 nonce[2]={0xe85bd7b5eca7924e,0x1d2691e5bf4c40c3};
        b64 plain_text[]={0x1234567890abcdef,0xabcdef1234567890,0xabcdef9876543210};
```

```
    b64 cipher_text[3]={0};
    //setting up initial internal state
    internal_state[0]=IV;
    internal_state[1]=key[0];
    internal_state[2]=key[1];
    internal_state[3]=nonce[0];
    internal_state[4]=nonce[1];
    initialization(internal_state,key);
    encryption(internal_state, 3, plain_text,cipher_text);
    finalization_phase(internal_state,key);
    usleep(2000); // Sleep for 2 milliseconds
    clock_t end_time = clock();
    // Calculate overall duration in milliseconds and output it (subtract the 2 ms sleep)
    double duration = ((double)(end_time - start_time) / (CLOCKS_PER_SEC / 1000))-2 ;
    total_time += duration;

  }
  double average_time = total_time / 100;
   printf("Average time of authenticated encryption process (ASCON 128) %f milliseconds\n", average_time);
}
*/
```