CYBER-PHYSICAL SYSTEMS PROGRAMMING

Riccardo Gaspari Giulio Vignati Matteo Franchi

SCOPO DEL PROGETTO

Lo scopo del progetto è quello di realizzare ex novo 4 algoritmi crittografici sul cluster PULP:

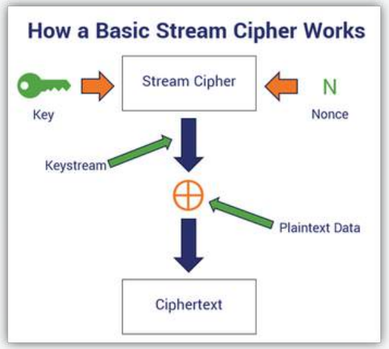
- AES-256-CTR,
- AES-256-GCM,

- ChaCha20,
- ChaCha20-Poly1305,

implementarne una versione parallelizzata sfruttando i core disponibili su GAP8 e valutare il guadagno raggiunto rispetto alla versione sequenziale.

AES CTR:

- 'Algoritmo di crittografia simmetrica con una chiave utilizzata per crittografare gli stessi dati;
- Stream cipher;

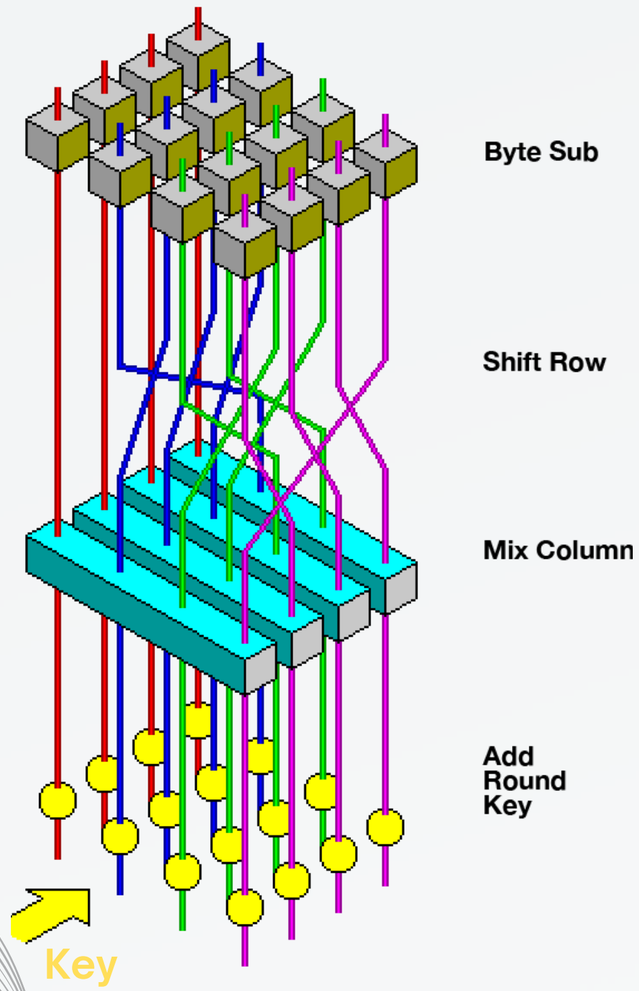


Ogni blocco è organizzato in una matrice 4x4 e viene ripetuto, per 14 round, un set di operazioni.

variante GCM:

Implementa anche l’autenticazione dei dati con un TAG di autenticazione (processo GHASH). Esso è utilizzato per garantire che i dati non siano stati modificati in modo non autorizzato.

CRIPTAZIONE AES



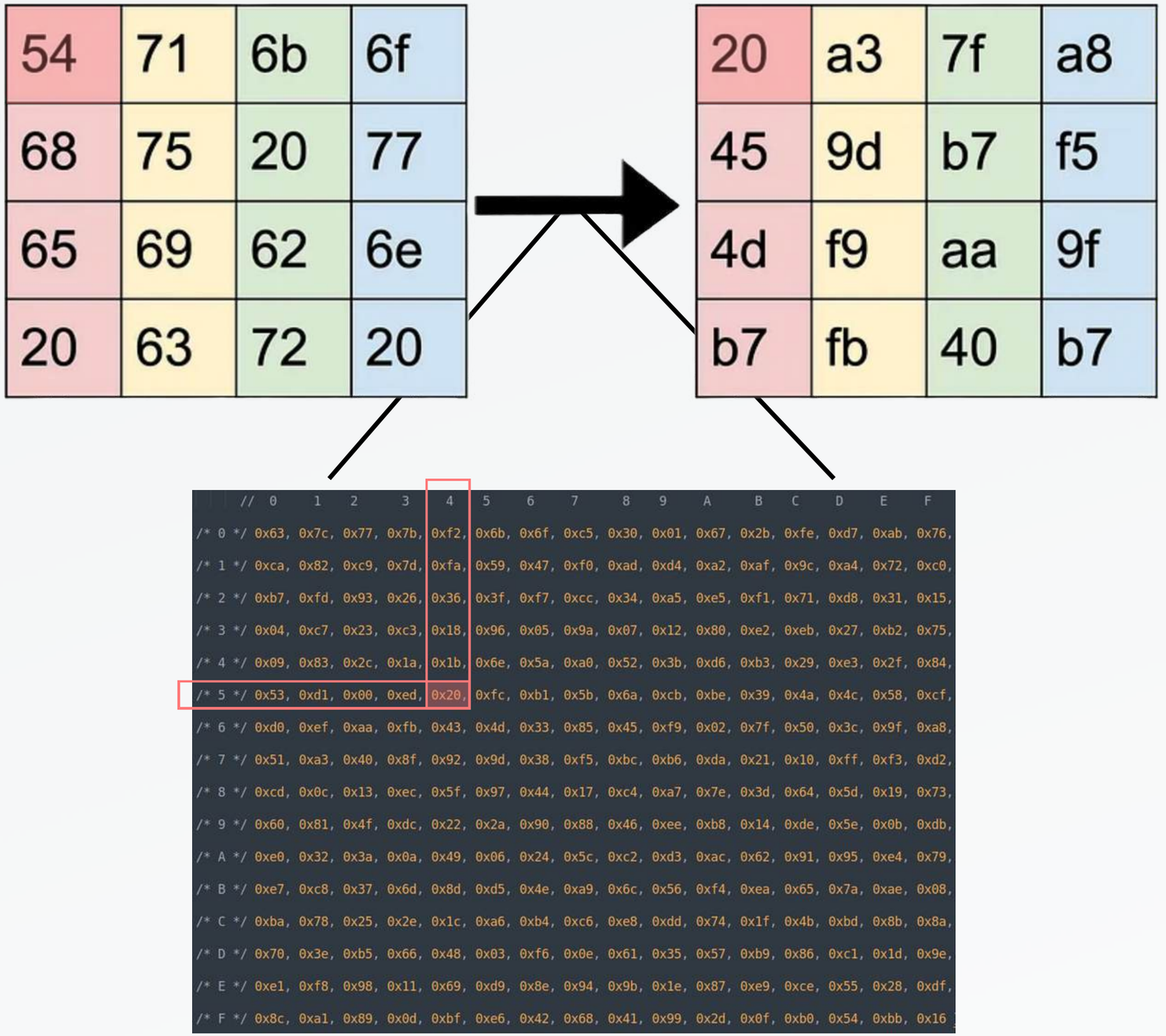
Ci sono 4 operazioni su matrici 4x4 che andremo ad esegurire:

- subBytes()
- shiftRows()
- mixColumns()
- addRoundKey()

.

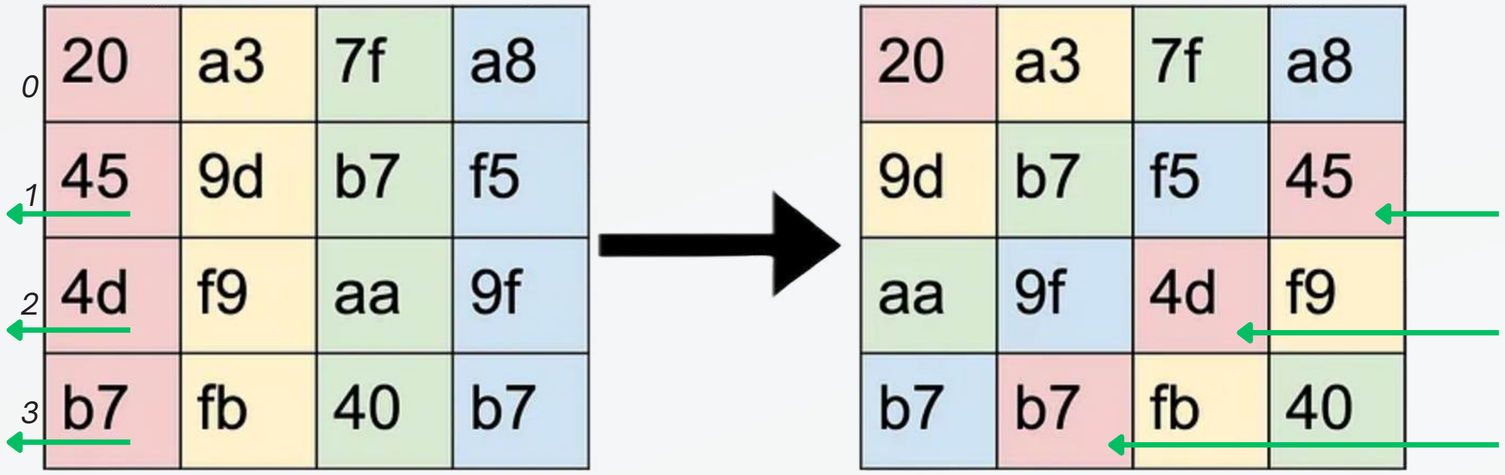
CRIPTAZIONE AES

- **subBytes();** sostituisco ogni elemento della matrice con il corrispettivo della matrice sub().



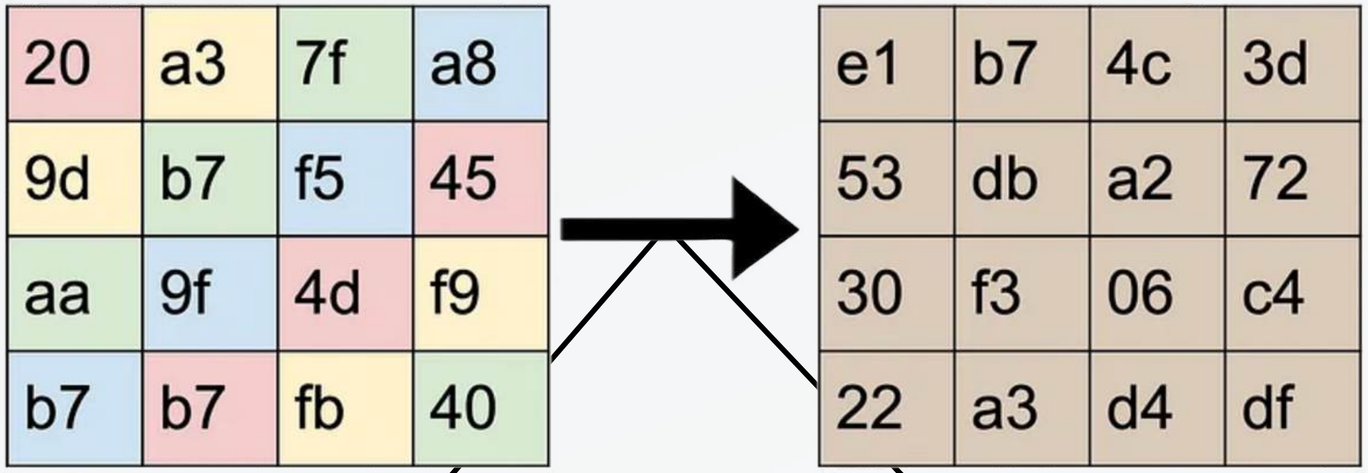
CRIPTAZIONE AES

- **shiftRows();** spostiamo ogni riga verso sinistra di un numero di spazi corrispondente al *numero* di riga



CRIPTAZIONE AES

- **mixColumns();** Ora moltiplichiamo ogni colonna per la seguente matrice usando l'aritmetica GF(2^8) (gmul() e XOR invece delle normali addizioni e moltiplicazioni):



$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} \cdot \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix}$$

$$d_0 = 2 \cdot b_o \oplus 3 \cdot b_1 \oplus 1 \cdot b_2 \oplus 1 \cdot b_3$$
$$d_1 = 1 \cdot b_o \oplus 2 \cdot b_1 \oplus 3 \cdot b_2 \oplus 1 \cdot b_3$$
$$d_2 = 1 \cdot b_o \oplus 1 \cdot b_1 \oplus 2 \cdot b_2 \oplus 3 \cdot b_3$$
$$d_3 = 3 \cdot b_o \oplus 1 \cdot b_1 \oplus 1 \cdot b_2 \oplus 2 \cdot b_3$$

Vantaggi dell'operazione nel Galois Field:

- **velocità computazionale, semplici operazioni bitwise (XOR e shift);**
- **operazioni chiuse, associative, commutative e distributive;**
- **risultati uniformi e prevedibili senza overflow;**

CRIPTAZIONE AES

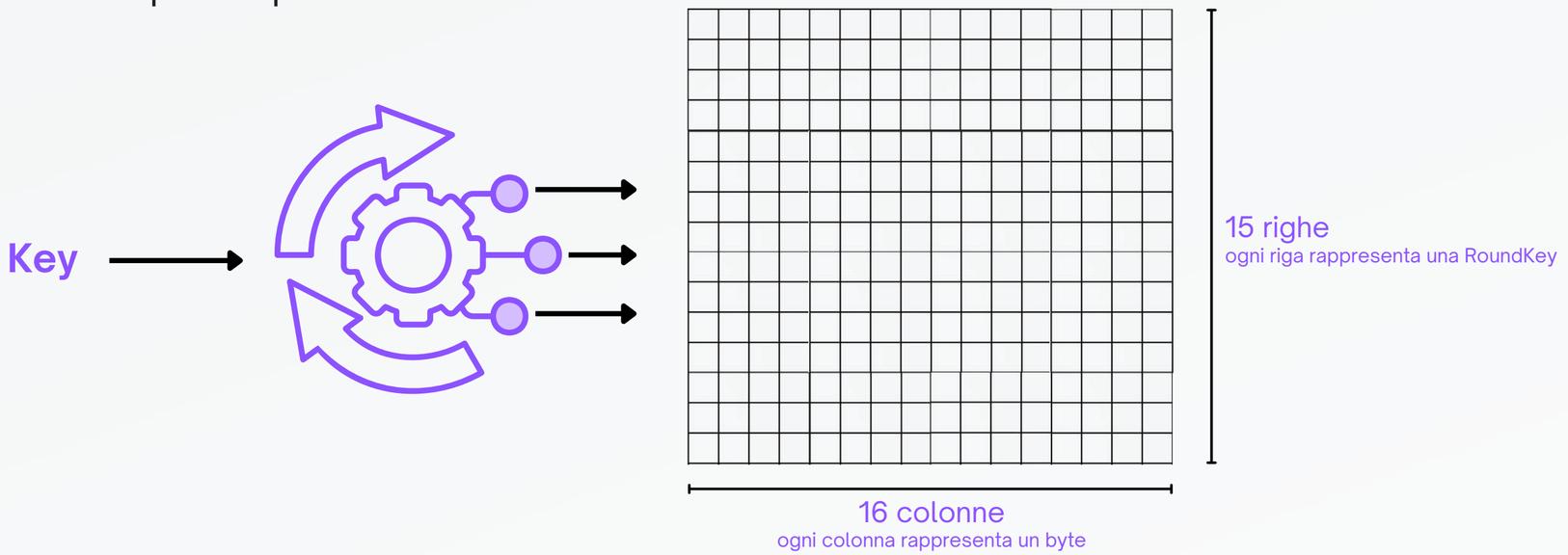
- **addRoundKey();** XOR ogni byte nel blocco con il rispettivo byte della RoundKey.

e1	b7	4c	3d
53	db	a2	72
30	f3	06	c4
22	a3	d4	df

→

80	d2	25	50
31	bd	c8	1c
53	94	d6	ab
46	cb	b8	af

RoundKey è un set di chiavi di round generate a valle di una serie di trasformazioni non lineari e permutazioni, partendo dalla chiave principale.

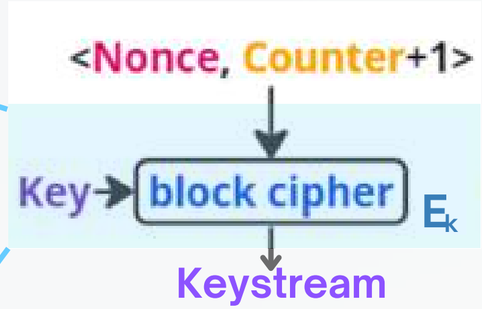


CRIPTAZIONE AES-CTR (COUNTER MODE)

- Generazione di un Counter, incrementato per ogni blocco tale da essere differente per ognuno di essi;
- Nonce/Initialization Vector, elemento arbitrario per inizializzare la criptazione;

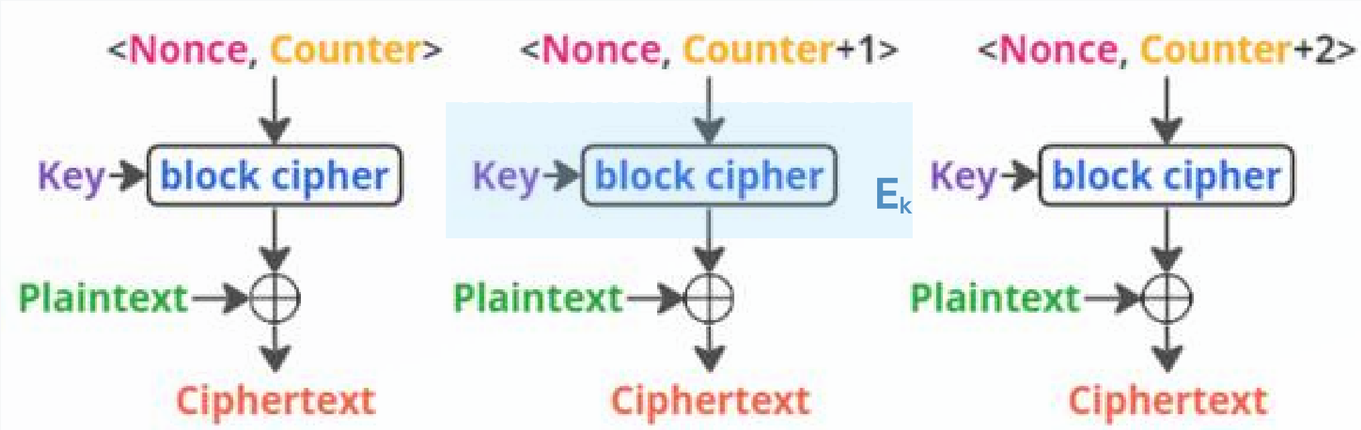
- I due elementi vengono criptati tramite algoritmo AES. Dalla key viene internamente ottenuta la RoundKey usata in AES;

```
// Cipher is the main function for AES encryption
void Cipher(state_t* state, const uint8_t* RoundKey)
{
  uint8_t round = 0;

  // Add the First round key to the state before starting the rounds.
  AddRoundKey(0, state, RoundKey);

  // There will be Nr rounds.
  // The first Nr-1 rounds are identical.
  // These Nr rounds are executed in the loop below.
  // Last one without MixColumns()
  for (round = 1; ; ++round)
  {
    SubBytes(state);
    ShiftRows(state);
    if (round == Nr) {
      break;
    }
    MixColumns(state);
    AddRoundKey(round, state, RoundKey);
  }
  // Add round key to last round
  AddRoundKey(Nr, state, RoundKey);
}
```



CRIPTAZIONE AES-CTR

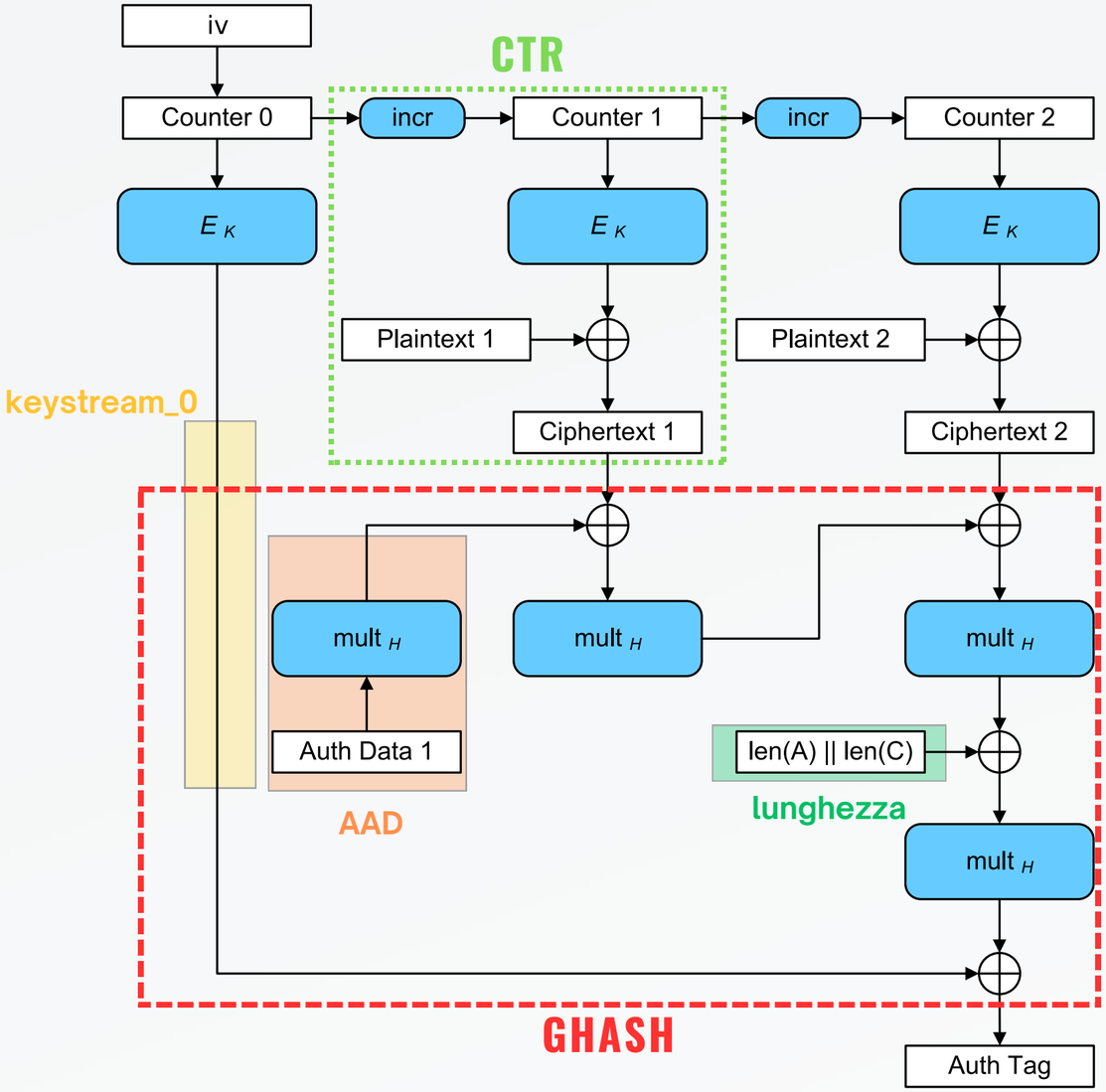
- Per ogni risultato di tale criptazione (**Keystream**) viene svolta l'operazione di XOR con il blocco di **Plaintext** corrispettivo per completare la criptazione



AES GCM (GALOIS/COUNTER MODE)

Processo di autenticazione: verifica basata su:

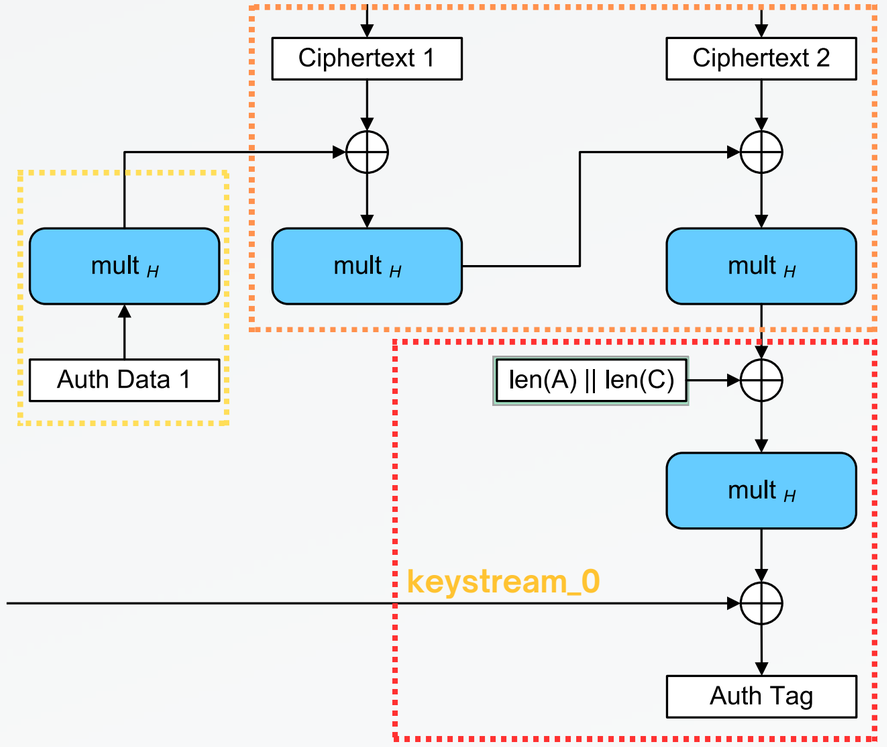
- Lunghezza del messaggio trasmesso (len(A));
- Dati di autenticazione addizionali (AAD);
- Match del processo di criptazione del Ciphertext;



TAG GENERATION STRUCTURE:

La generazione del TAG è stata suddivisa in:

- INIT gestione del primo blocco di AAD;
- UPDATE con ulteriori blocchi di AAD e Ciphertext;
- FINISH contributo di len(A) e keystream_0;



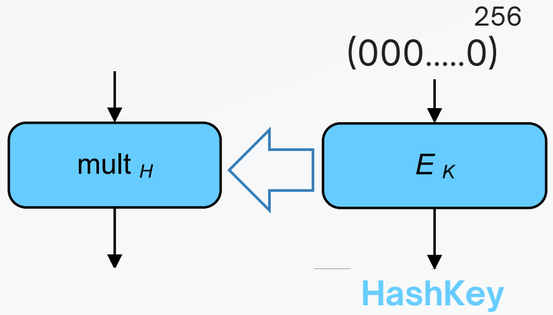
MOLTIPLICAZIONE IN GF(2^{128})

La hash_key è utilizzata durante l'operazione di moltiplicazione in GF(2^128), nella fase di autenticazione.

Moltiplicazione in GF(2^128):

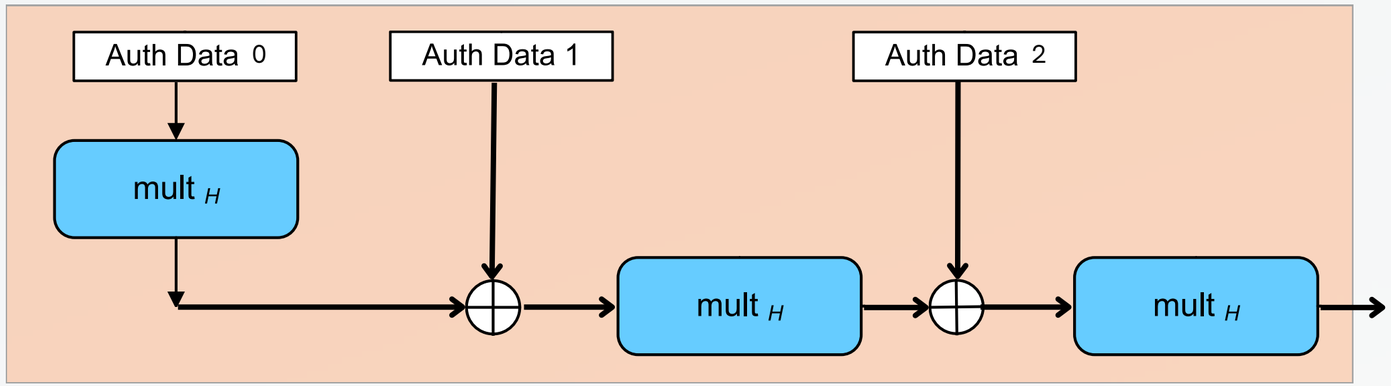
1. *Rappresentazione*: Ogni elemento è un polinomio binario di grado < 128.
2. *Moltiplicazione*: Moltiplica i polinomi come normali polinomi binari.
3. *Riduzione Modulo*: Riduci il risultato modulo in un polinomio irriducibile $X^{128}+X^{7}+X^{2}+X+1$
4. XOR per somma e shift per moltiplicazione

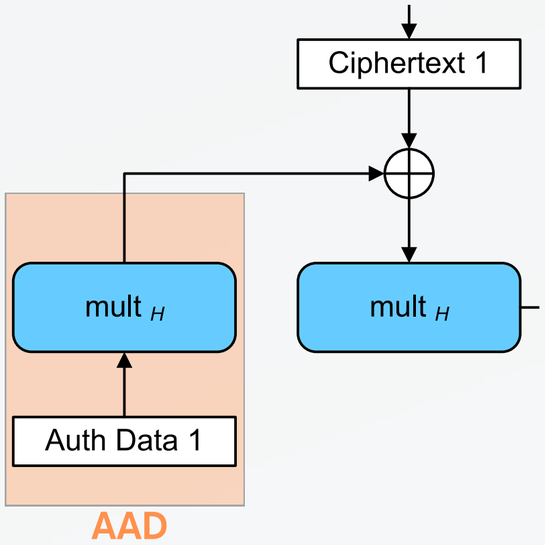
```c
// Define the irreducible polynomial R for GF(2^128)
#define R 0xE1 // R = 11100001

// Function to perform right shift
void right_shift(uint8_t V[16]) {
    for (int i = 15; i > 0; i--) {
        V[i] = (V[i] >> 1) | ((V[i - 1] & 0x01) << 7);
    }
    V[0] >>= 1;
}


//funzione della MUL_H
void gf128_mul(uint8_t AAD[16], uint8_t HashKey[16], uint8_t OUT[16]) {
    uint8_t V[16];

    // Copy AAD to V
    for (int i = 0; i < 16; i++) {
        V[i] = AAD[i];
    }

    // Initialize Z to 0
    for (int i = 0; i < 16; i++) {
        OUT[i] = 0;
    }

    // Perform the multiplication
    for (int i = 0; i < 128; i++) {
        if (HashKey[i / 8] & (1 << (7 - i % 8))) {
            // Yi = 1, perform Z ^ V
            for (int j = 0; j < 16; j++) {
                OUY[j] ^= V[j];
            }
        }
        if (V[15] & 0x01) {
            // V127 = 1, perform rightshift(V) ^ R
            right_shift(V);
            V[0] ^= R;
        } else {
            // V127 = 0, perform rightshift(V)
            right_shift(V);
        }
    }
}
```



ADDITIONAL AUTHENTICATED DATA

Gli AAD sono a loro volta divisi in blocchi e seguono il seguente processo di criptazione:

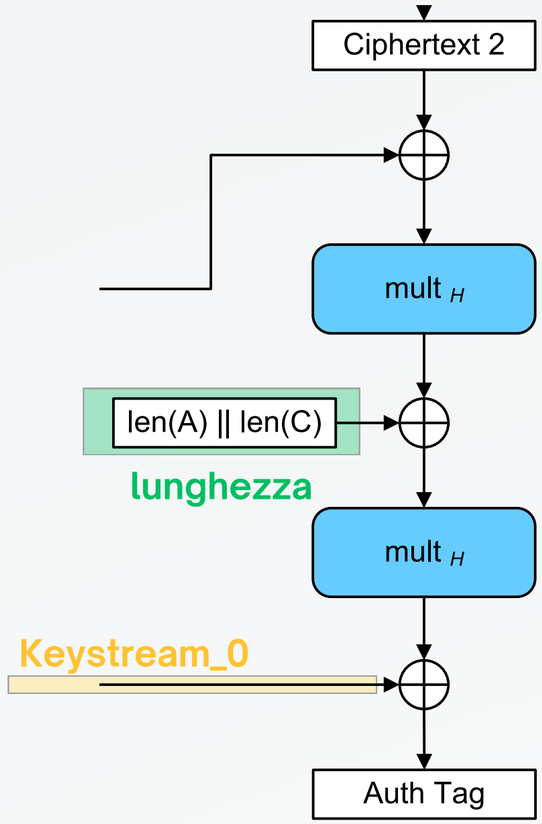




LUNGHEZZA E KEYSTREAM_0

Al termine delle operazioni tra AAD e Ciphertext, viene effettuata l'operazione di XOR con la **lunghezza** e infine con il **Keystream_0** .

Termina così il processo di autenticazione **GHASH** da cui si ottiene il Tag di autenticazione.



CHACHA20:

TIPO DI ALGORITMO

Approccio di crittografia simmetrico che crittografa e decrittografa i dati con la stessa chiave a 256 bit.

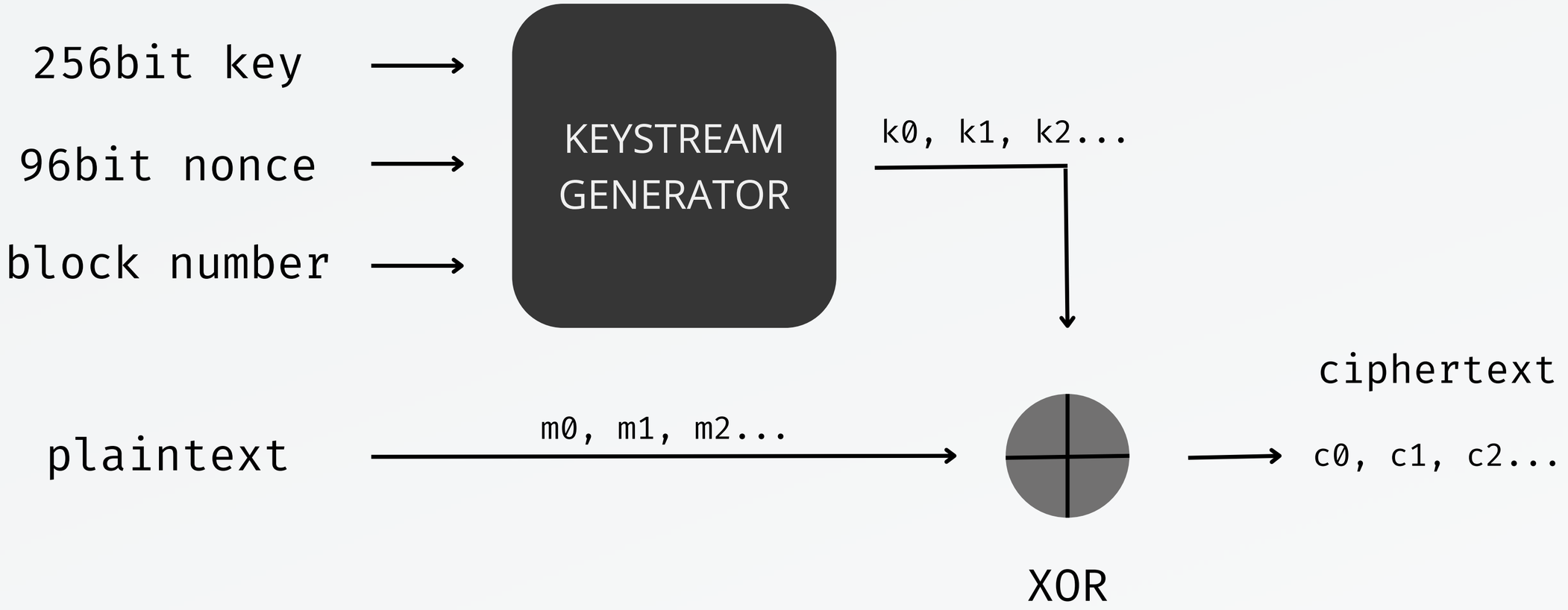
ChaCha20 è uno “stream cipher” (codice a flusso) che crittografa i dati in flussi continui anziché in blocchi di dimensione fissa.

Permette di avere un buon rapporto velocità-sicurezza e permette facilmente implementazioni in parallelo.

STREAM CIPHER WORKFLOW:

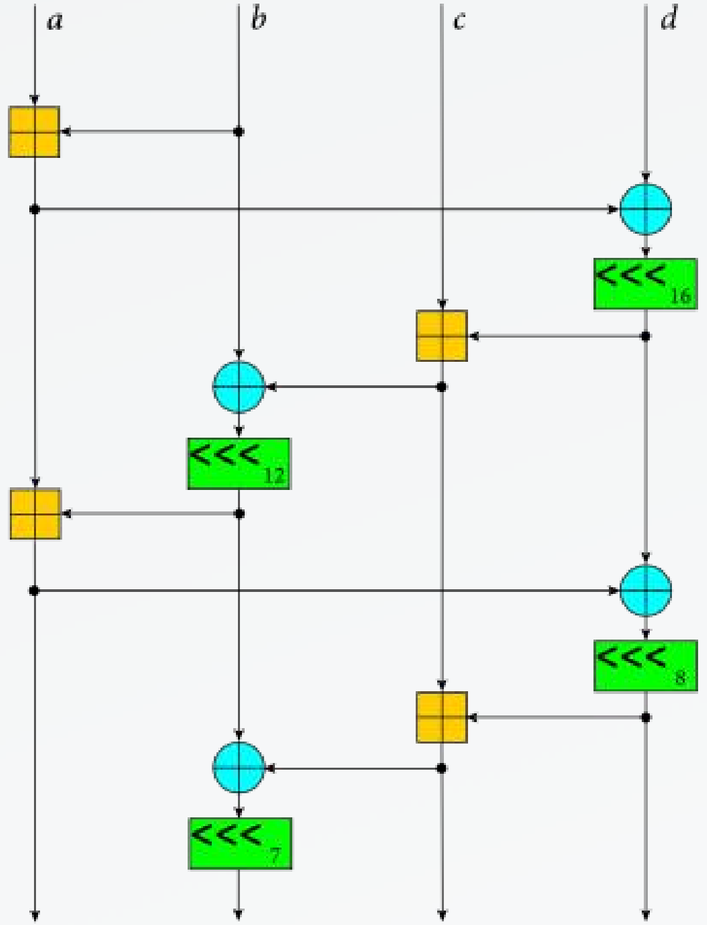
- Cifrario simmetrico
- Bit che codificano plaintext indipendenti l’uno dall’altro
- Trasformazione dei simboli successivi varia con il procedere della cifratura
- Ogni simbolo dipende da uno stato corrente
- Esecuzione più veloce rispetto alla struttura cifrari a blocchi
- Richiesta computazionale minore implica necessità di HW meno complesso

CHACHA20 BLOCK DIAGRAM:



QUARTER ROUND OPERATION:

```
#define QUARTERROUND(x, a, b, c, d) \
  x[a] += x[b]; x[d] = ROTL32(x[d] ^ x[a], 16); \
  x[c] += x[d]; x[b] = ROTL32(x[b] ^ x[c], 12); \
  x[a] += x[b]; x[d] = ROTL32(x[d] ^ x[a], 8); \
  x[c] += x[d]; x[b] = ROTL32(x[b] ^ x[c], 7);
```



Opera su integer 32bit

"+" indica una somma tra interi modulo 2^32

"^" indica un bitwise Exclusive OR (XOR)

"<<< n" indica n-bit rotazione verso sinistra
(possibile grazie alla funzione ROTL32)

CHACHA20 INITIAL STATE:

Lo stato di C20 è rappresentabile attraverso una matrice 4x4 con ogni cella da 32bit.

cost	cost	cost	cost
key[0]	key[1]	key[2]	key[3]
key[4]	key[5]	key[6]	key[7]
count	nonce[0]	nonce[1]	nonce[2]

La key viene importata in blocchi da 4 byte in little endian.

counter garantisce output diverso per ogni blocco

counter da 32bit può tenere traccia di fino a 2^32 blocchi di dati prima di esaurire lo spazio, un blocco in C20 è di 64 byte, un contatore a 32 bit è sufficiente per cifrare fino a 256GB di dati.

CHACHA20 KEYSTREAM GENERATION:

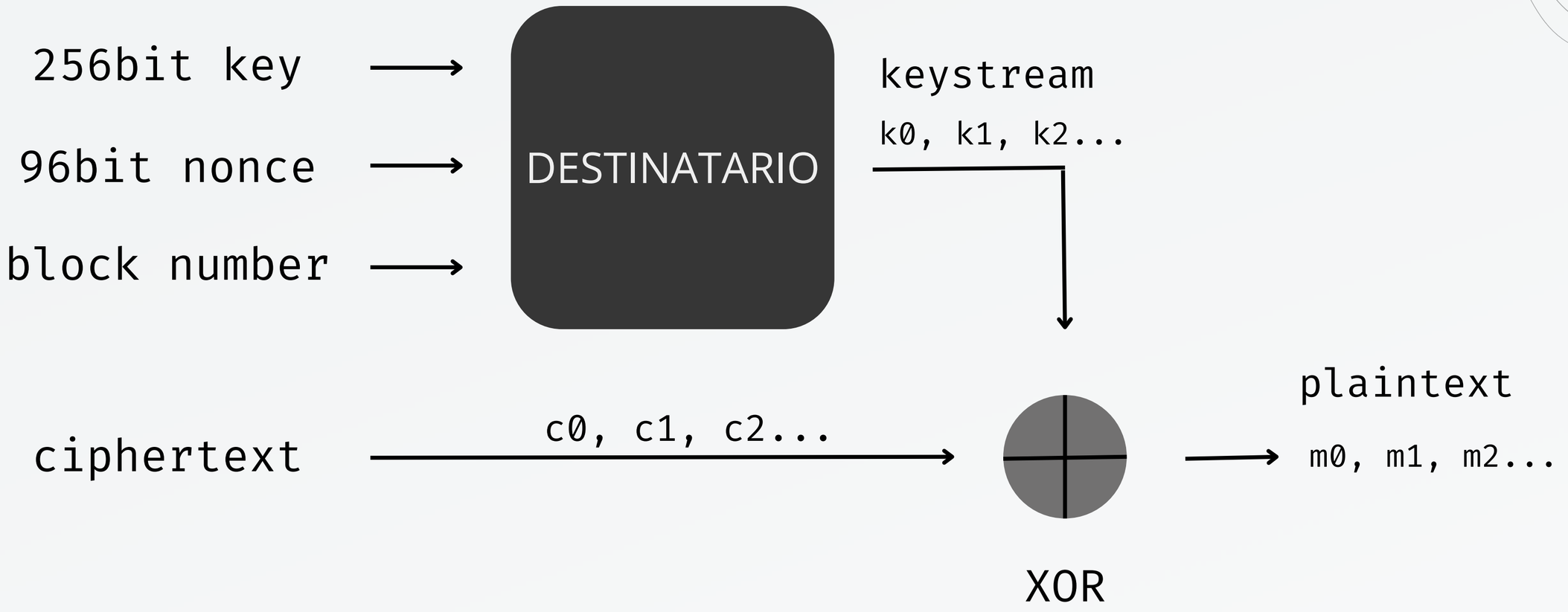
ChaCha20 esegue 20 rounds, alternando tra "column rounds" e "diagonal rounds" ognuno consiste di 4 quarter-rounds.

```
for (int i = 0; i < 20; i += 2) {
    QUARTERROUND(state, 0, 4, 8, 12)
    QUARTERROUND(state, 1, 5, 9, 13)
    QUARTERROUND(state, 2, 6, 10, 14)
    QUARTERROUND(state, 3, 7, 11, 15)
    QUARTERROUND(state, 0, 5, 10, 15)
    QUARTERROUND(state, 1, 6, 11, 12)
    QUARTERROUND(state, 2, 7, 8, 13)
    QUARTERROUND(state, 3, 4, 9, 14)
}

for (int i = 0; i < 16; ++i)
    out[i] += state[i];
```

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

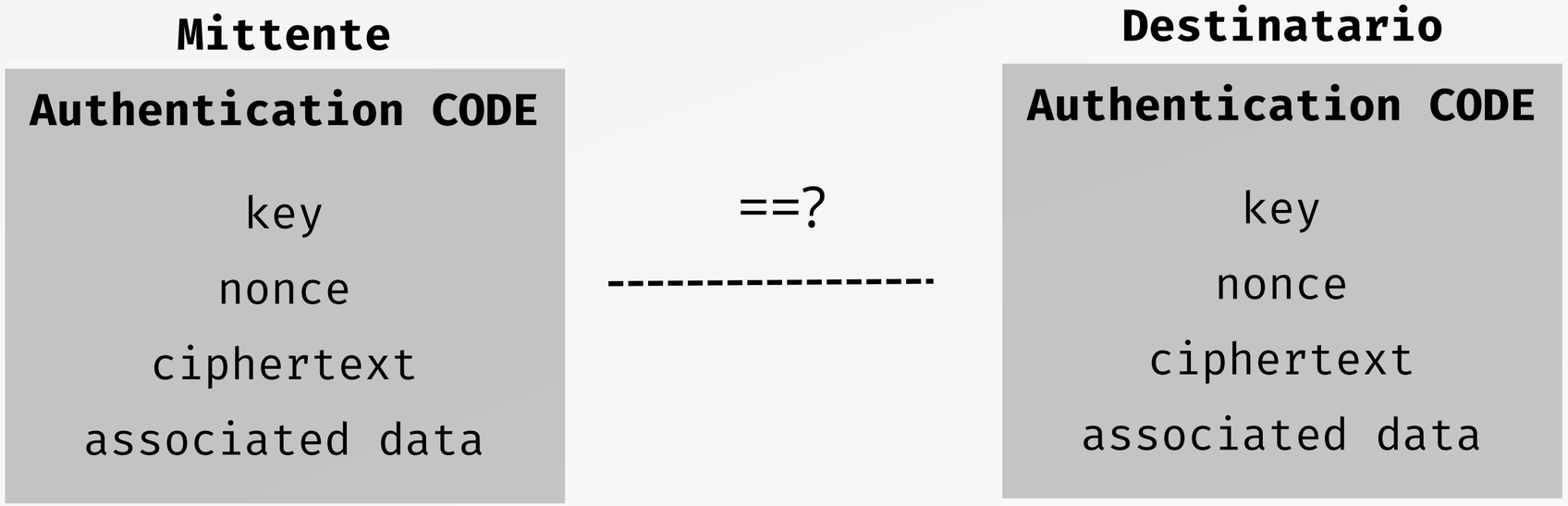
Finita questa operazione viene generato il keystream da mettere in xor con il plaintext.

CHACHA20 DECRYPTION:

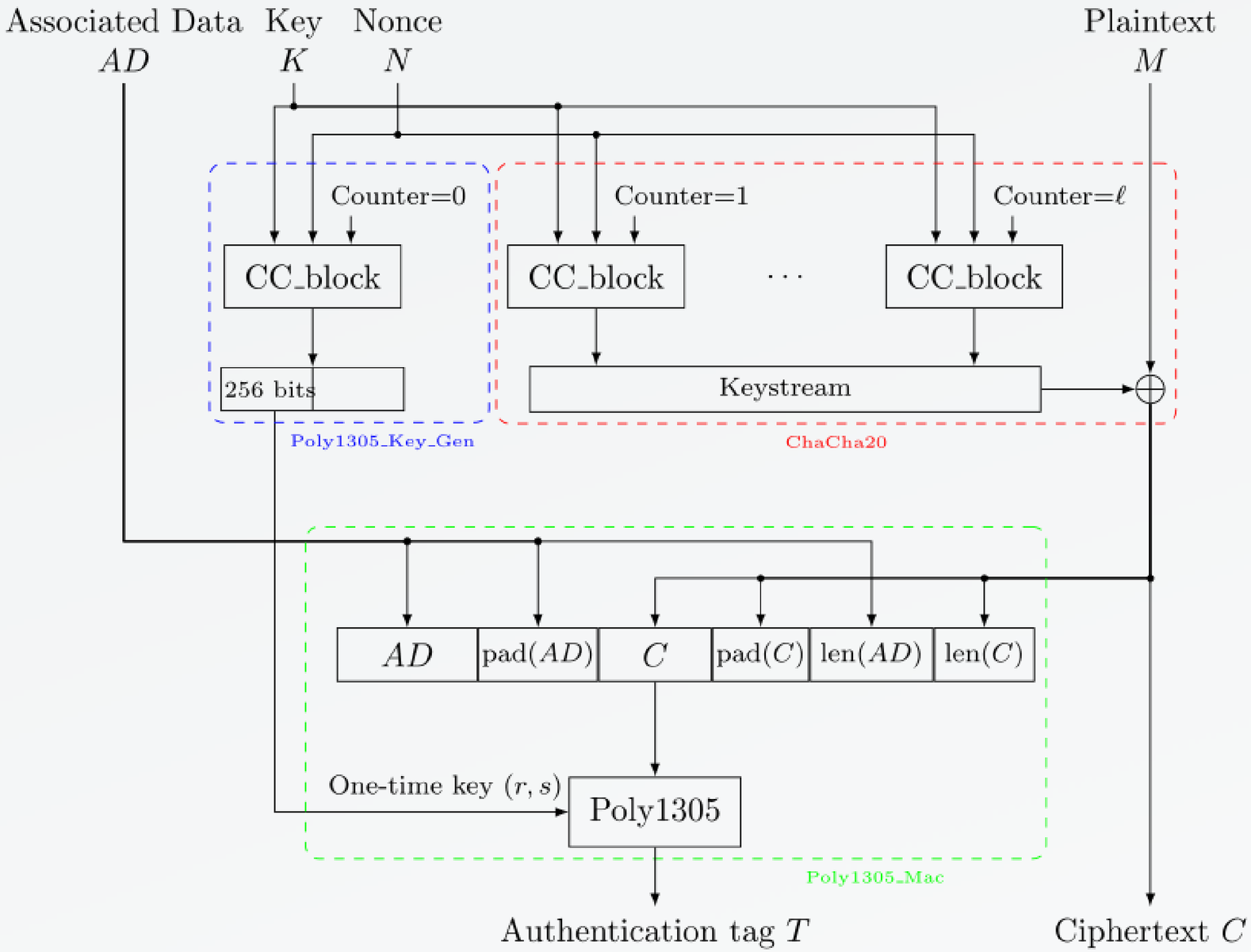


CHACHA20-POLY1305:

- Combina algoritmo Chacha20 con autenticazione Poly1305
- Garantisce maggiore sicurezza
- L’autenticazione verifica l’integrità dei dati



C20P1305 STRUCTURE:



C20P1305 STRUCTURE:

poly1305_context

```
uint32_t r[5];
uint32_t h[5];
uint32_t pad[4];
size_t leftover;
unsigned char buffer[16];
unsigned char final;
```

Secret key Poly1305 ⟶ array of five 32-bit integers

Current State ⟶ array of five 32-bit integers

Padding ⟶ array of four 32-bit integers

Conta bit non processati nel blocco precedente

Buffer dati in ingresso

Flag che indica se blocco è stato processato

C20P1305 STRUCTURE:

`init`

Inizializza algoritmo con key e nonce

`update`

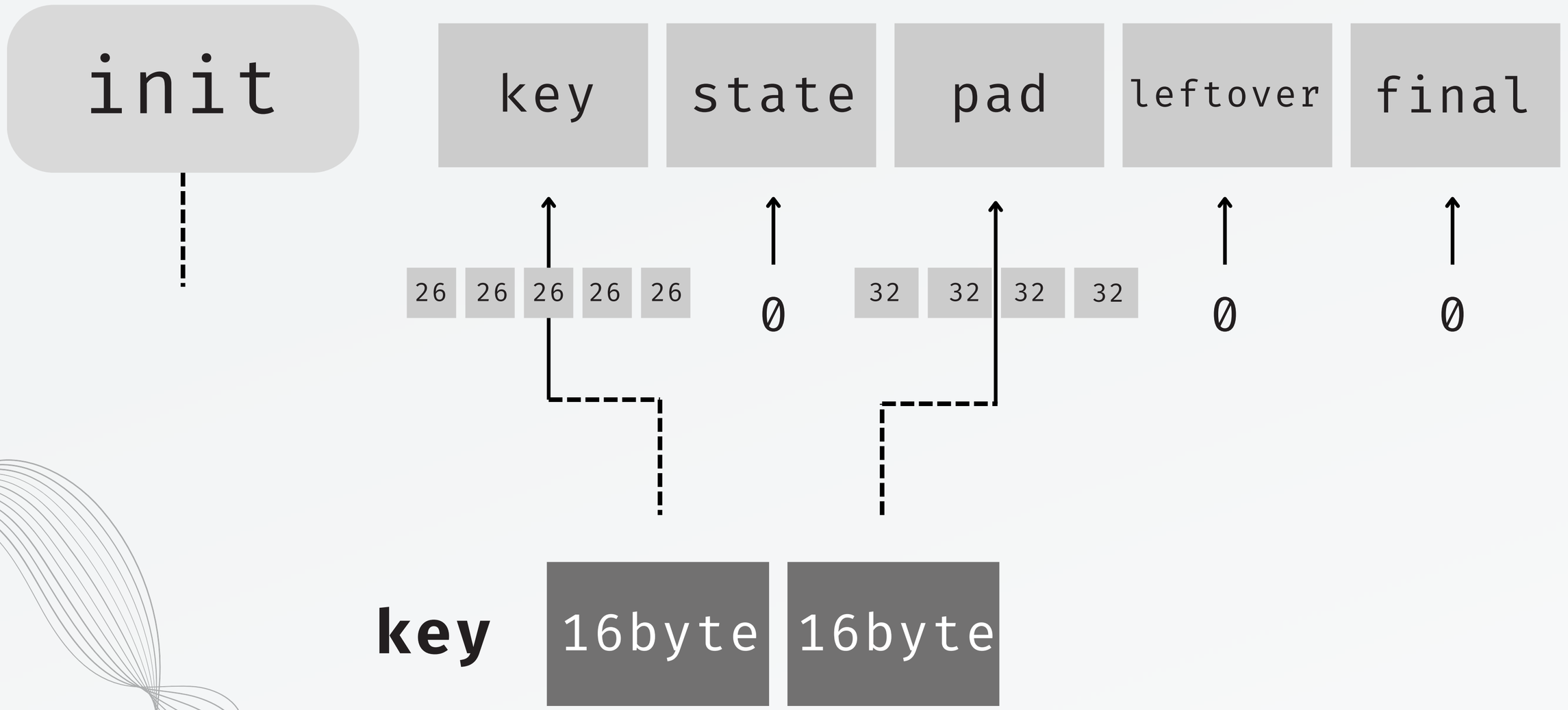
Aggiornamento hash:
chiamata ogni volta che si deve cifrare nuovo blocco

`finish`

Finalizzazione hash:

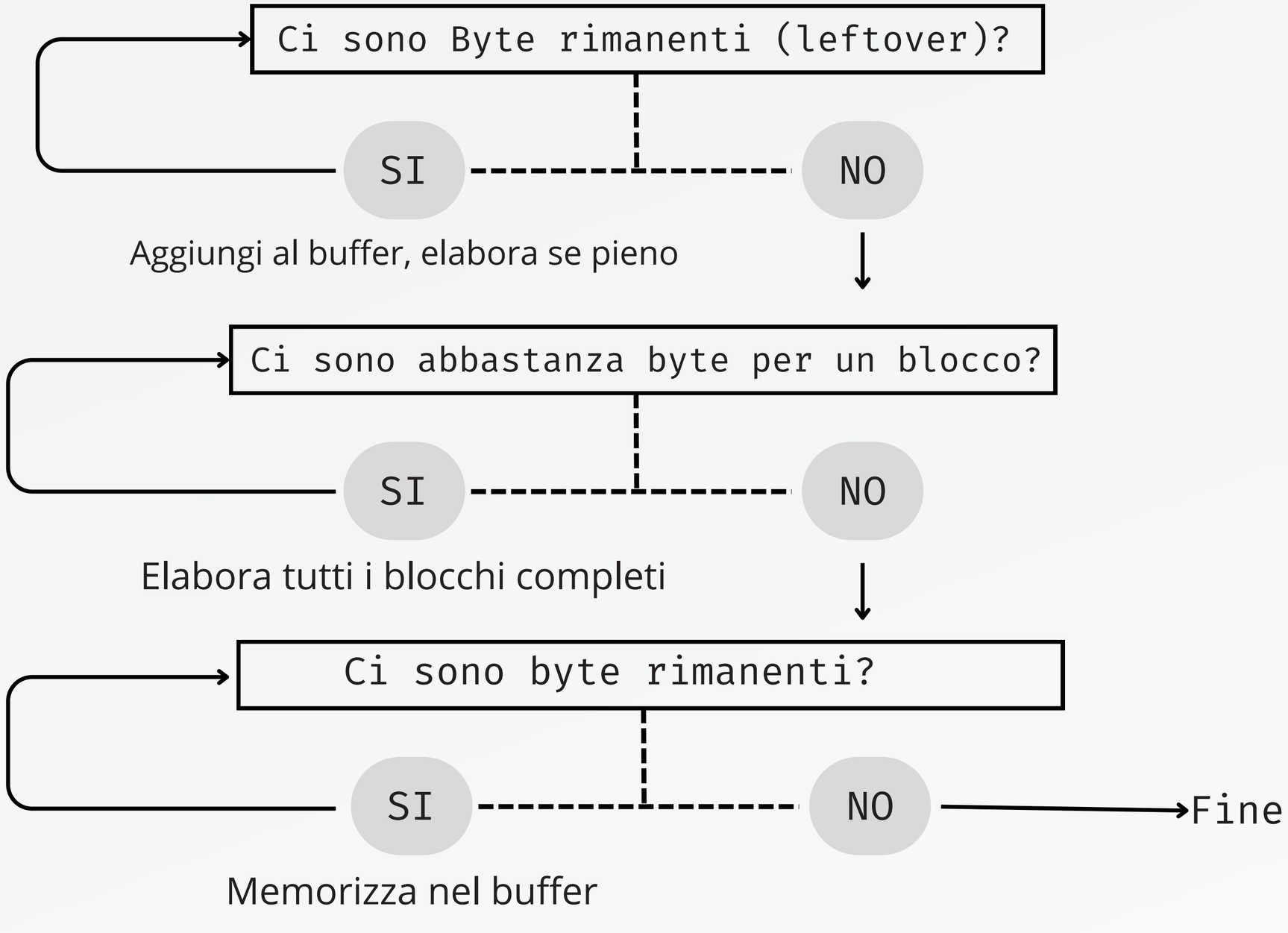
restituisce hash finale utilizzato per la verifica

C20P1305 INIT FUNCTION:

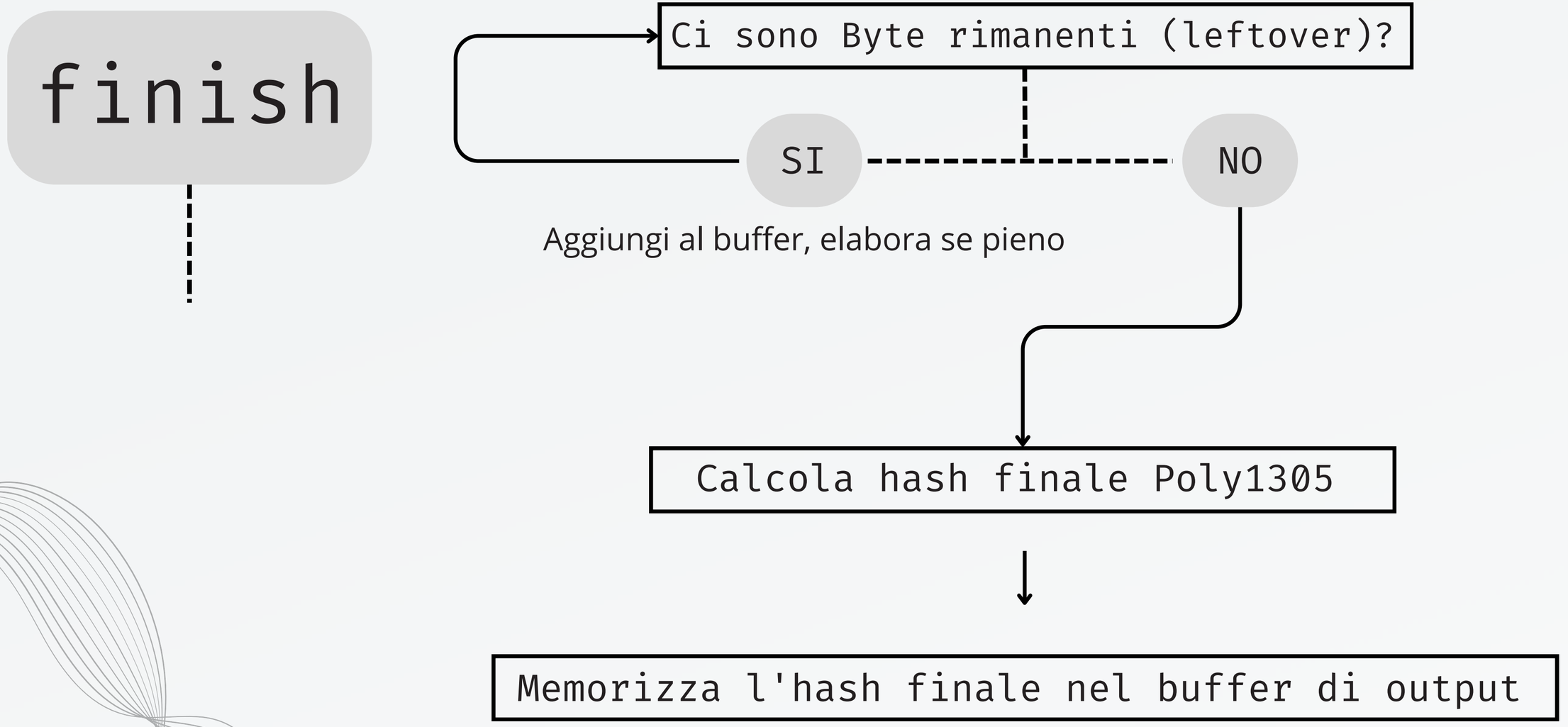


C20P1305 UPDATE FUNCTION:

update



C20P1305 FINISH FUNCTION:



C20P1305 FINISH FUNCTION:

finish

Calcola hash finale Poly1305

↓

Memorizza l'hash finale nel buffer di output

Aggiornamento dello stato corrente

stato

26	26	26	26	26
h0	h1	h2	h3	h4

Prevenzione Overflow

```
c = h1 >> 26
h1 = h1 & 0x3ffffff
h2 += c
```

Calcolo di g = h+ (-p)

g abbastanza grande da diventare negativo?

SI

Uso **h**

NO

Uso **g**

C20P1305 FINISH FUNCTION:

finish

Calcola hash finale Poly1305

↓

Memorizza l'hash finale nel buffer di output

“**h**” o “**g**”

Riduzione di valore scelto

26	26	26	26	26
§0	§1	§2	§3	§4

Calcolo TAG:

#0 = Resto(§0 **or** §1)

#1 = Resto(§1 **or** §2)

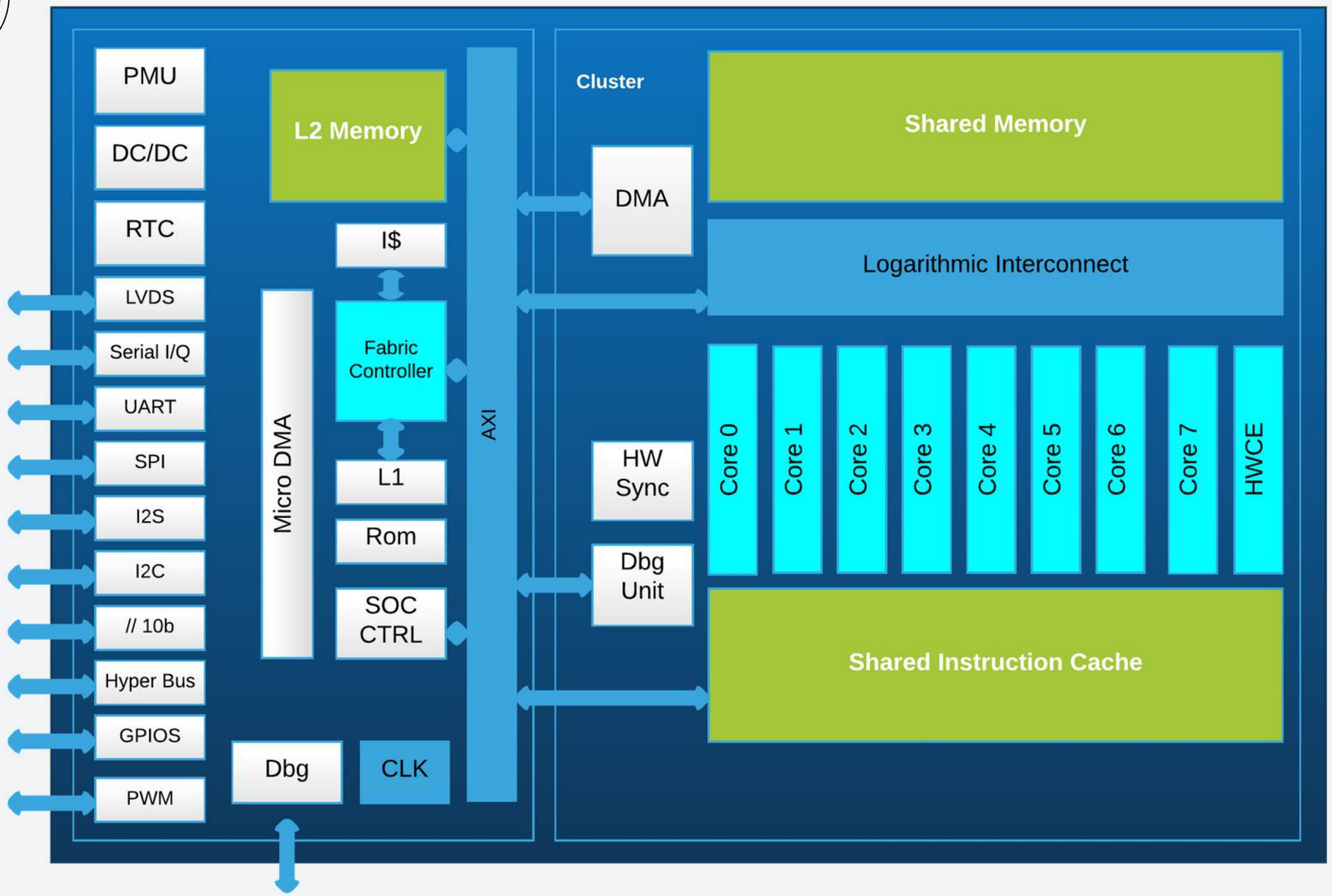
⋮

tag = (§ + pad) % (2^128)

Ogni elemento del vettore <2^32

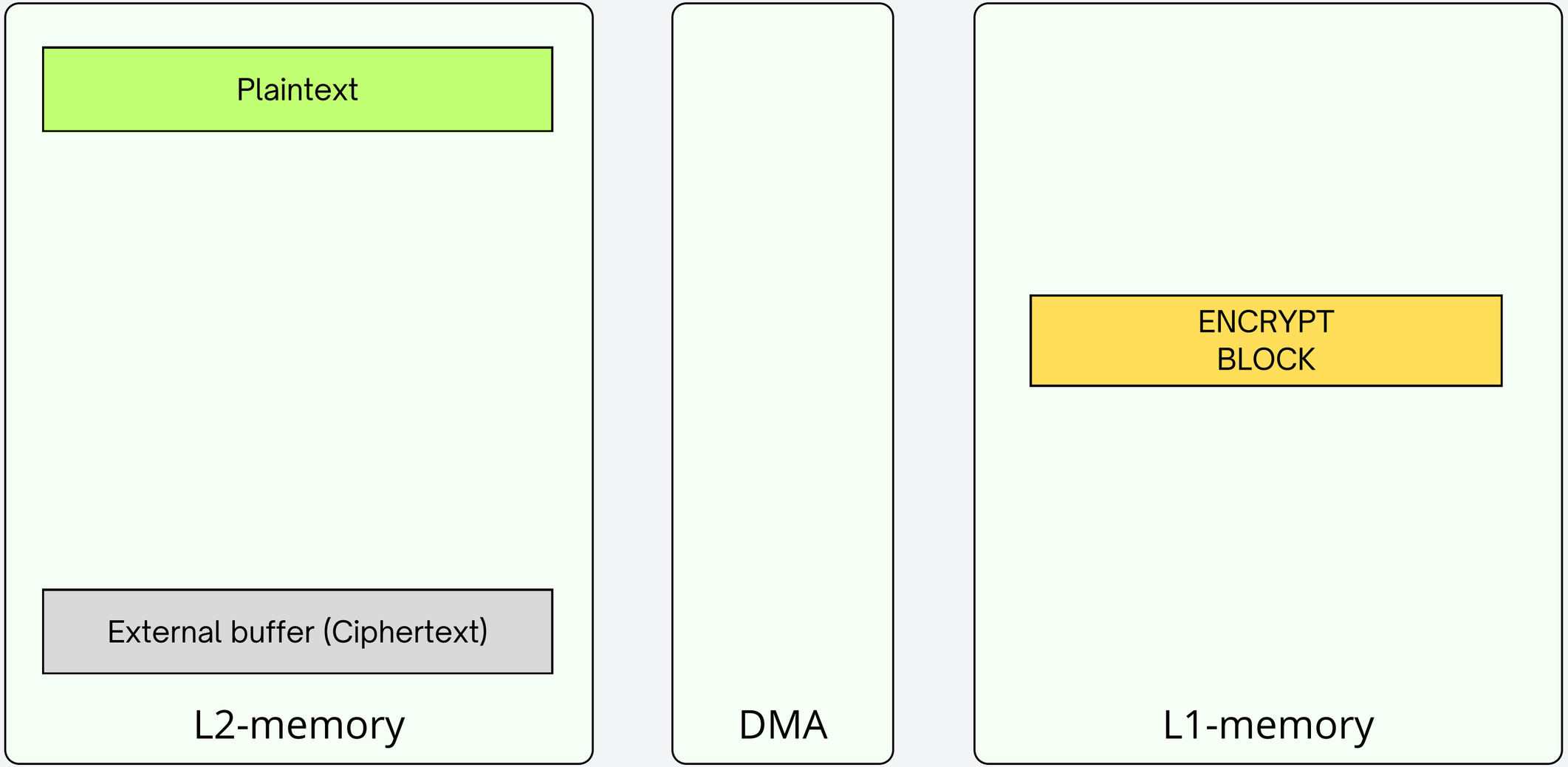
IMPLEMENTAZIONE SU GVSOC

GAP8 MICRO-ARCHITECTURE



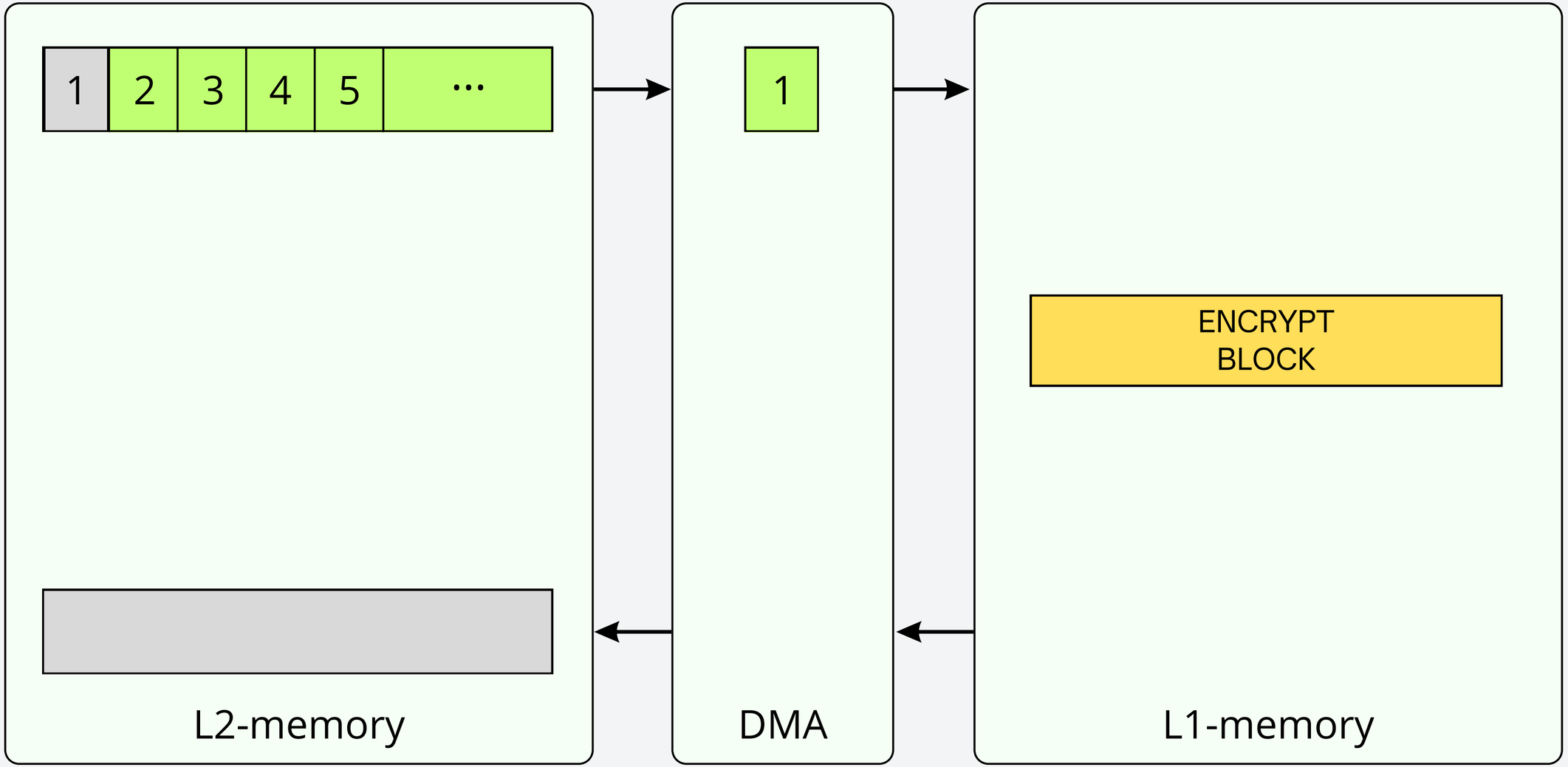
- Fabric controller core (for control, communications and security functions);
- 8 cores cluster;
- Global L2 memory;
- L1 memory shared by the 8 cores;
- Multi-channel cluster-DMA (for the transactions between the L2 Memory and L1 Memory).

DMA TRANSACTION CONTROL



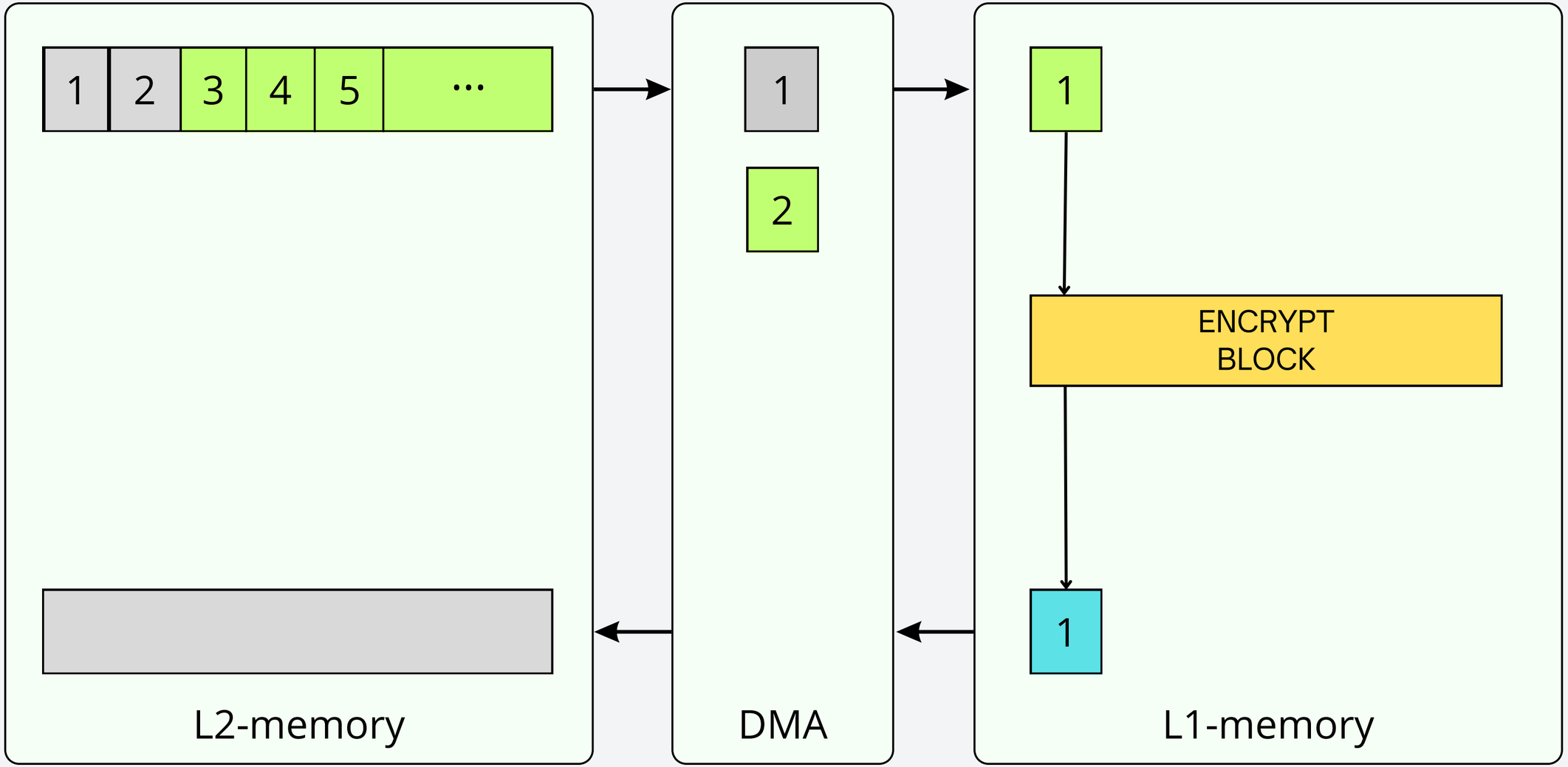
DMA TRANSACTION CONTROL

Iteration 0



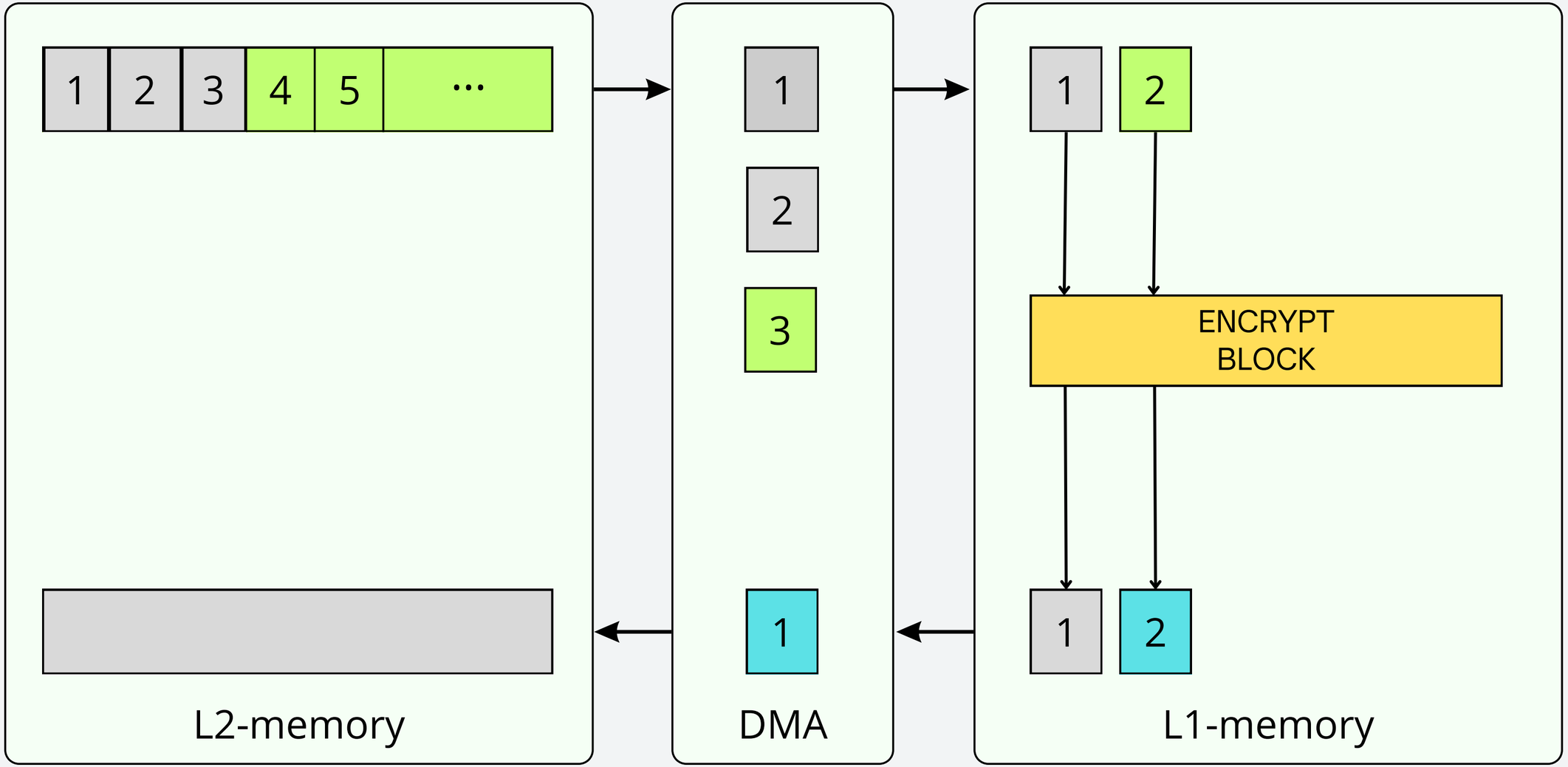
DMA TRANSACTION CONTROL

Iteration 1



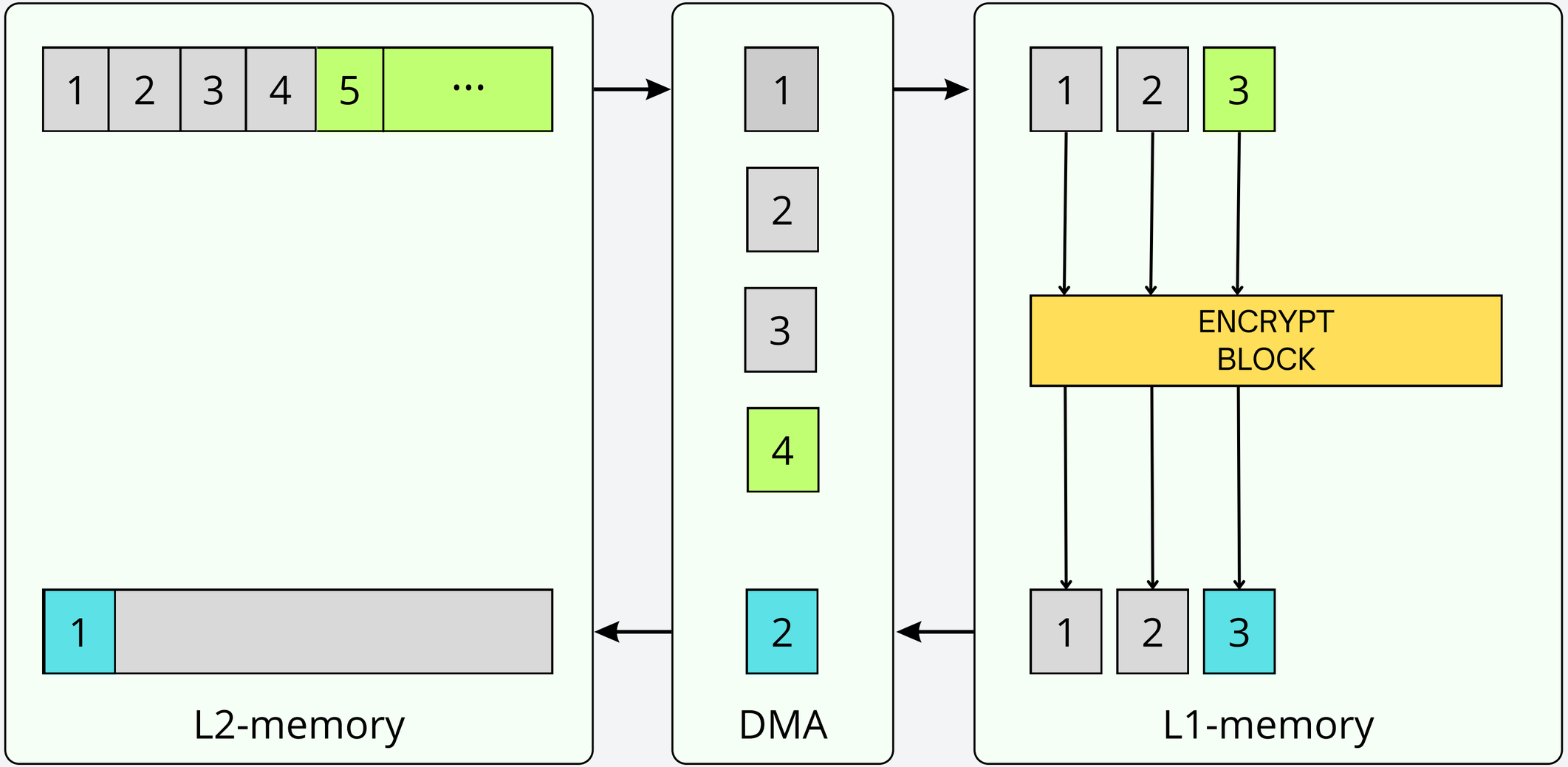
DMA TRANSACTION CONTROL

Iteration 2



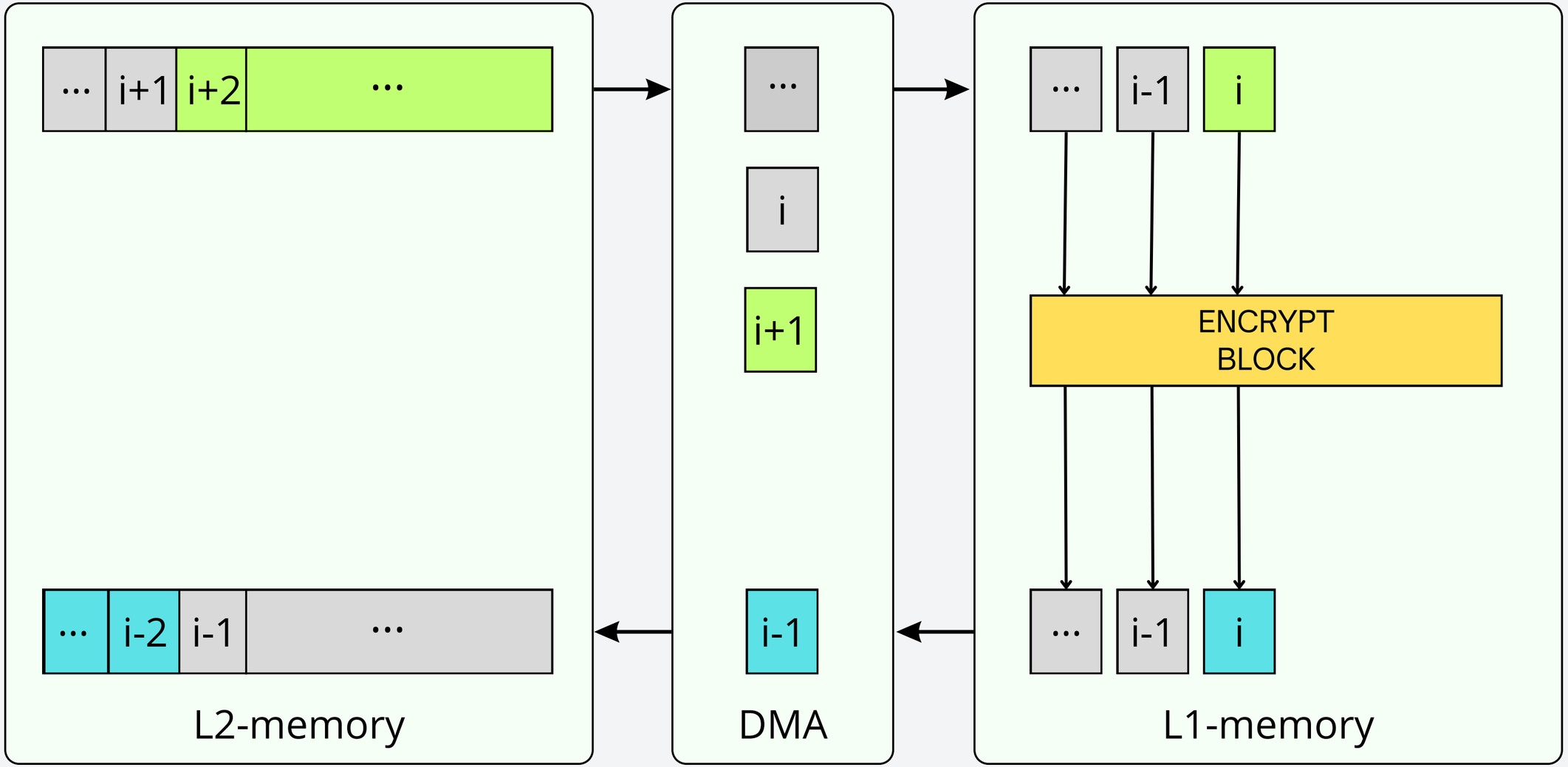
DMA TRANSACTION CONTROL

Iteration 3

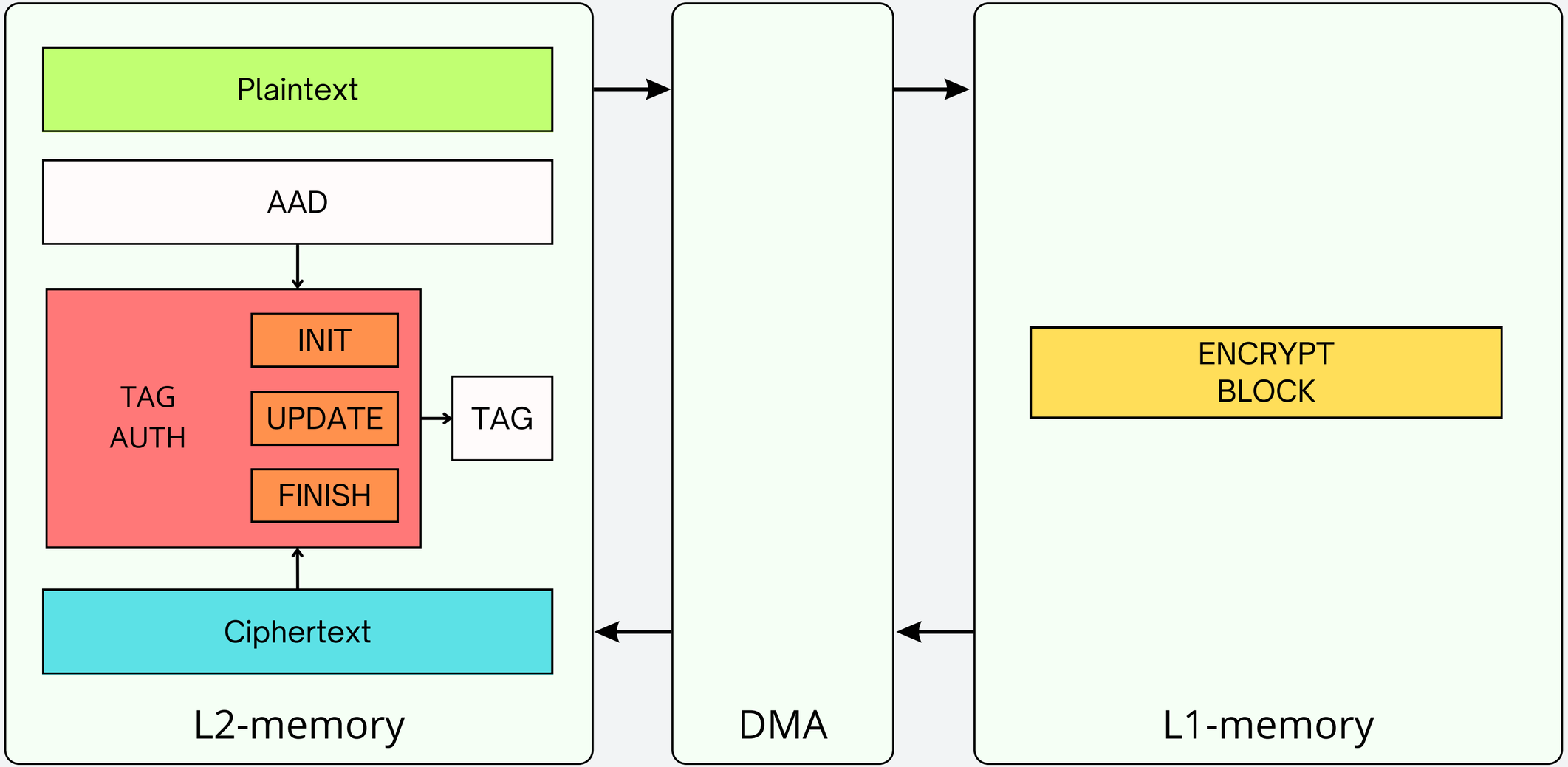


DMA TRANSACTION CONTROL

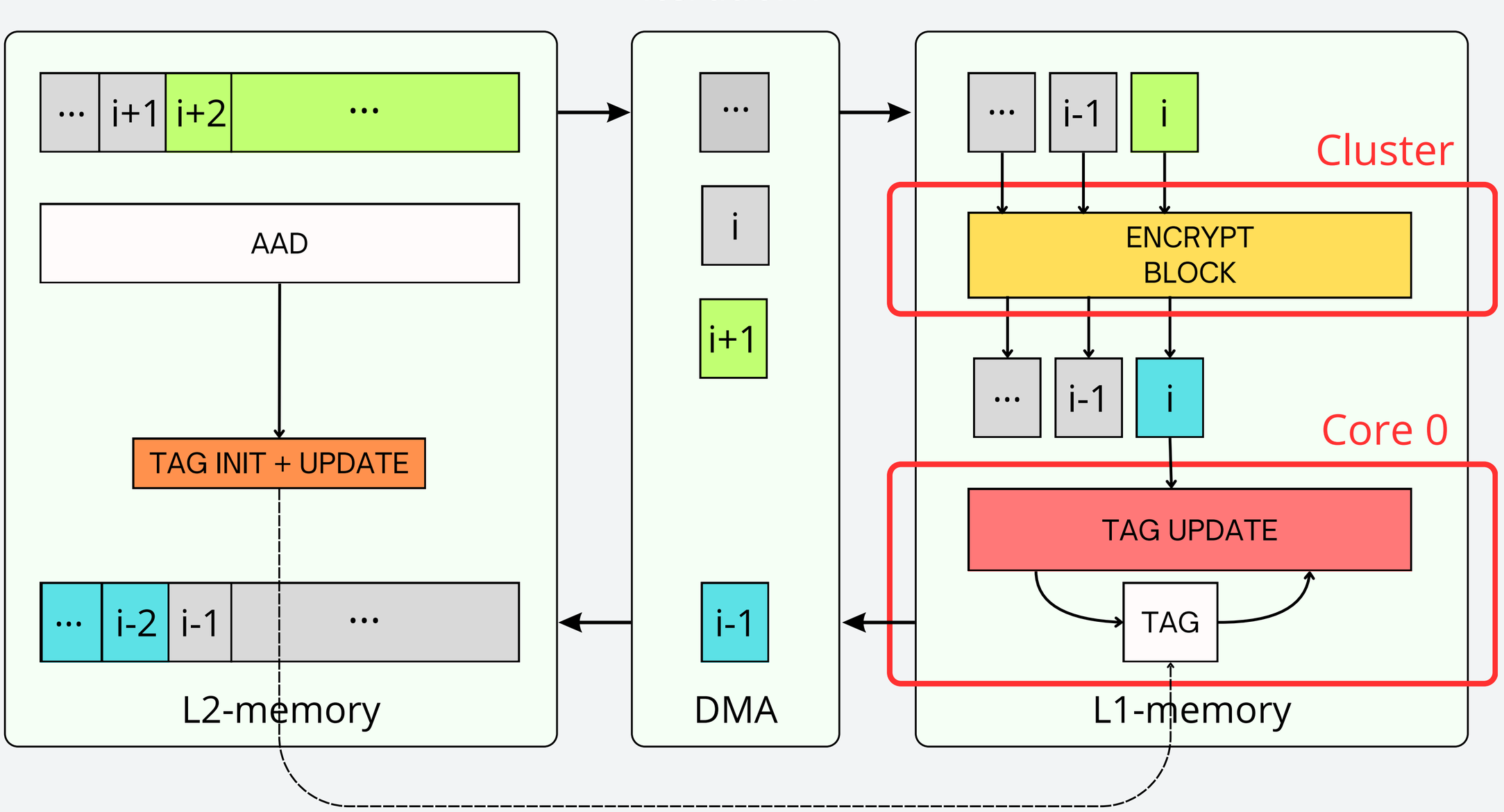
Iteration i



AUTHENTICATION



ENCRYPTION + AUTHENTICATION



ENCRYPTION + AUTHENTICATION

- Initialization tag computation with aad
- Update every cycle with the encrypted cyphertext’s block

Pros:

- Update of the digest done with variables all in L1 memory (faster access)

Cons:

- Update intrinsically sequential for both the algorithms, only 1 of the 8 cores are computing while the others are waiting

But:

- That would have also been the case computing the update of the ciphertext after moving it into L2 memory

ENCRYPTION + AUTHENTICATION: FUTURE STEPS

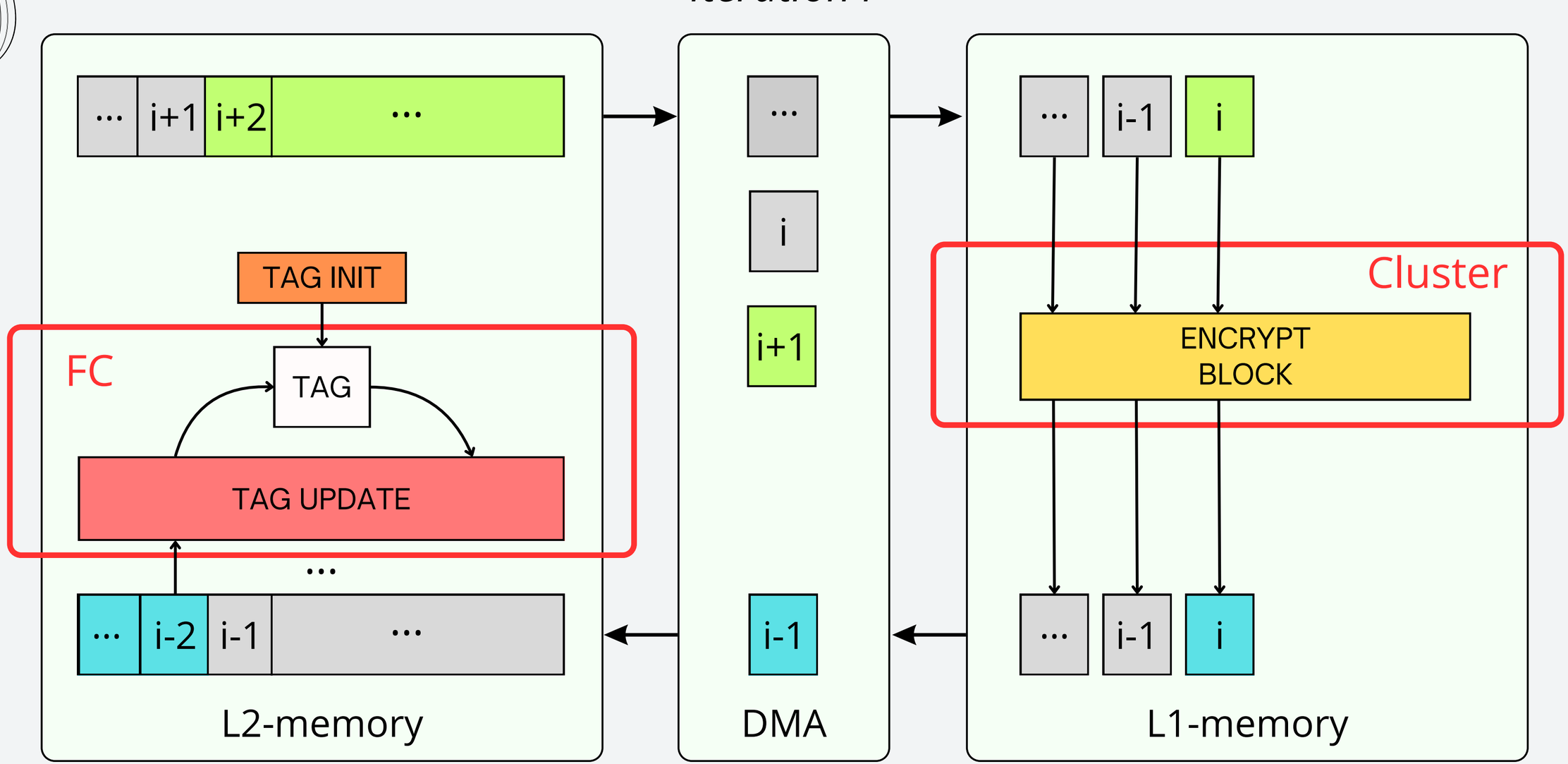
There might be the possibility to maximize the efficiency (and the performances overall) by synchronizing the FC and the cluster:

- CLUSTER: encryption block;
- FC: authentication

The synchronization would be implemented via team synchronization structures.

However, the solution has been attempted with sparse results, so it hasn’t been explored.

DMA TRANSACTION CONTROL



PERFORMANCE EVALUATION

PERFORMANCE ASSESSED

- Clock Cycles
- Instruction Count
- CPI / IPC
- Cycles per Byte
- Speedup
- Efficiency

Every measure has been performed with different payload sizes, from 512 bytes up to 32 Kbytes, in order to better analyze the results obtained.

Finally, we'll point to the memory-transfer/execution tradeoff, evaluating the cycles "wasted" for the dma control.

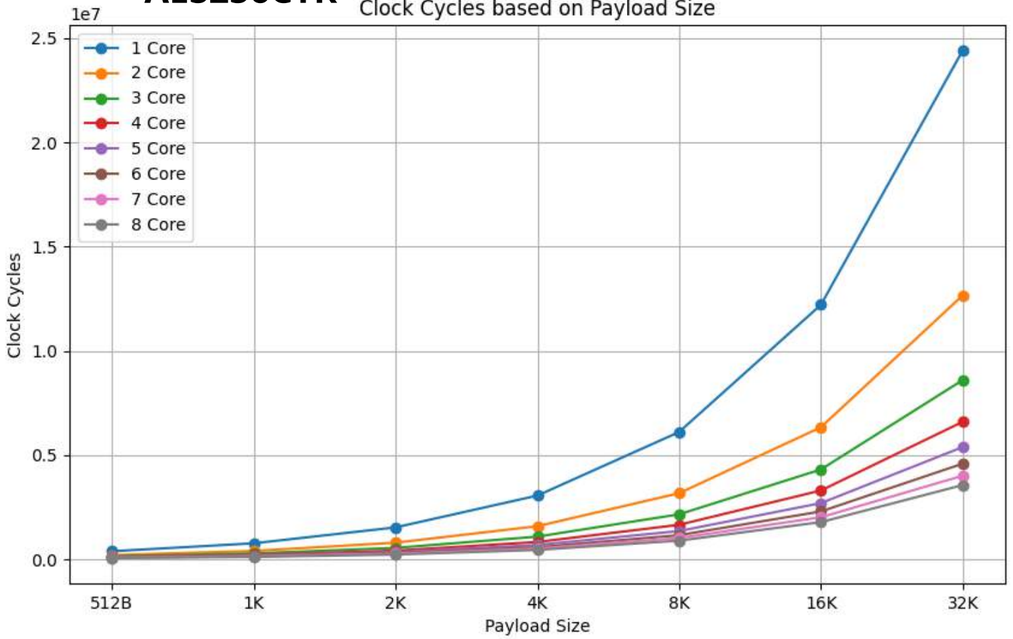
Generation of random TV

Generation of trace files for each (NUM_CORES, MEM_SIZE) configuration

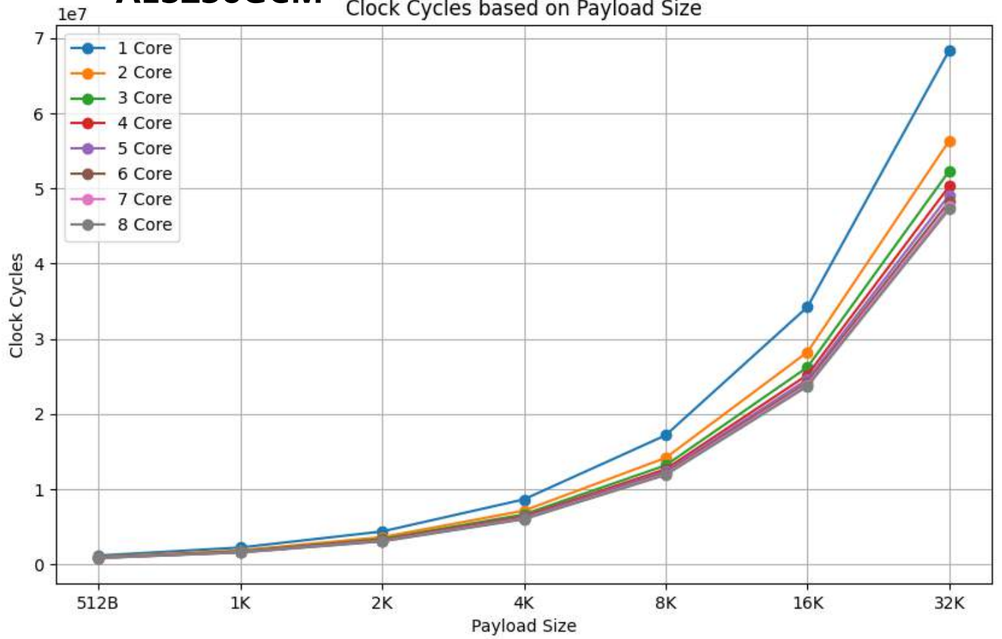
```bash
#!/bin/bash

# Directory to store the output files
OUTPUT_DIR="output_files"
mkdir -p "$OUTPUT_DIR"

# Array of MEM_SIZE values
MEM_SIZES=(512 1 2 4 8 16 32)

# Loop over NUM_CORES from 1 to 8
for NUM_CORES in {1..8}; do
    # Loop over each MEM_SIZE value
    for MEM_SIZE in "${MEM_SIZES[@]}"; do
        # Run the make command and redirect the output to a file
        OUTPUT_FILE="${OUTPUT_DIR}/${MEM_SIZE}_${NUM_CORES}cores.txt"
        echo "Running with NUM_CORES=$NUM_CORES and MEM_SIZE=${MEM_SIZE}"
        make clean all run USE_CLUSTER=1 NUM_CORES=$NUM_CORES MEM_SIZE=$MEM_
    done
done
```

Performance evaluation

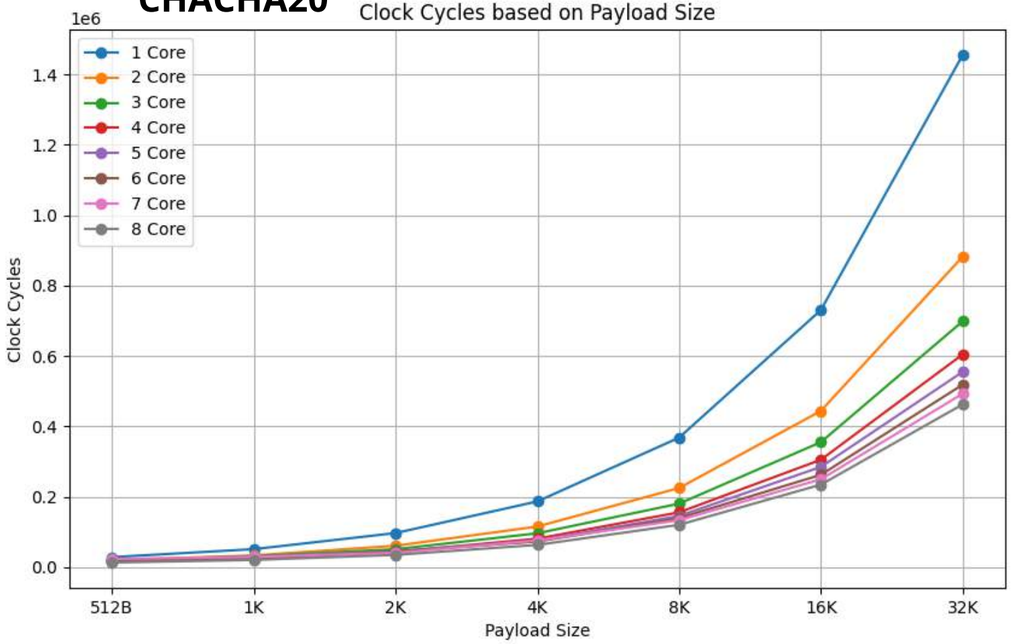
AES256CTR



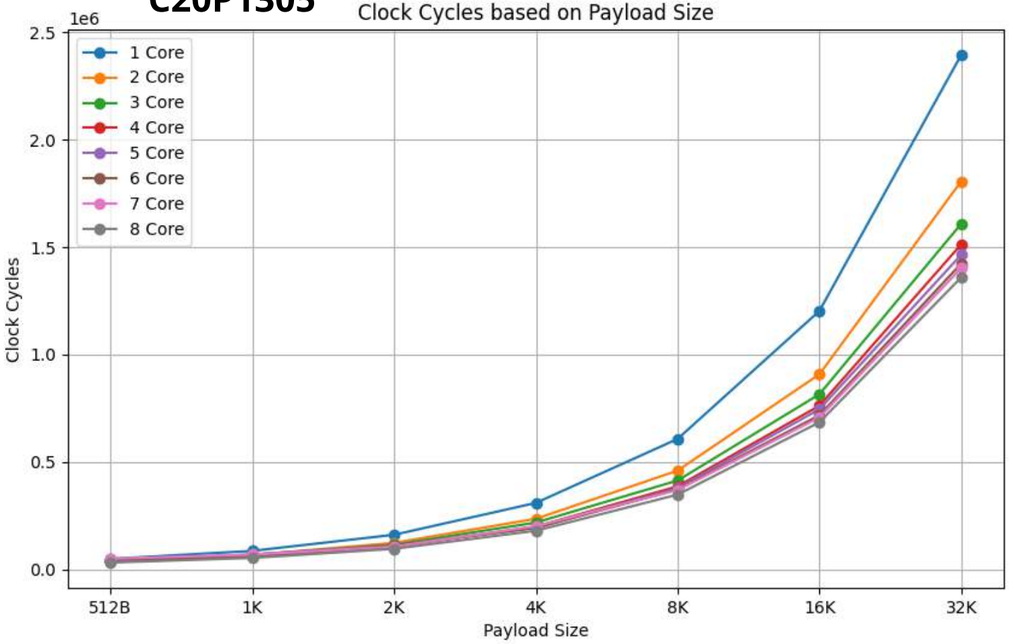
AES256GCM



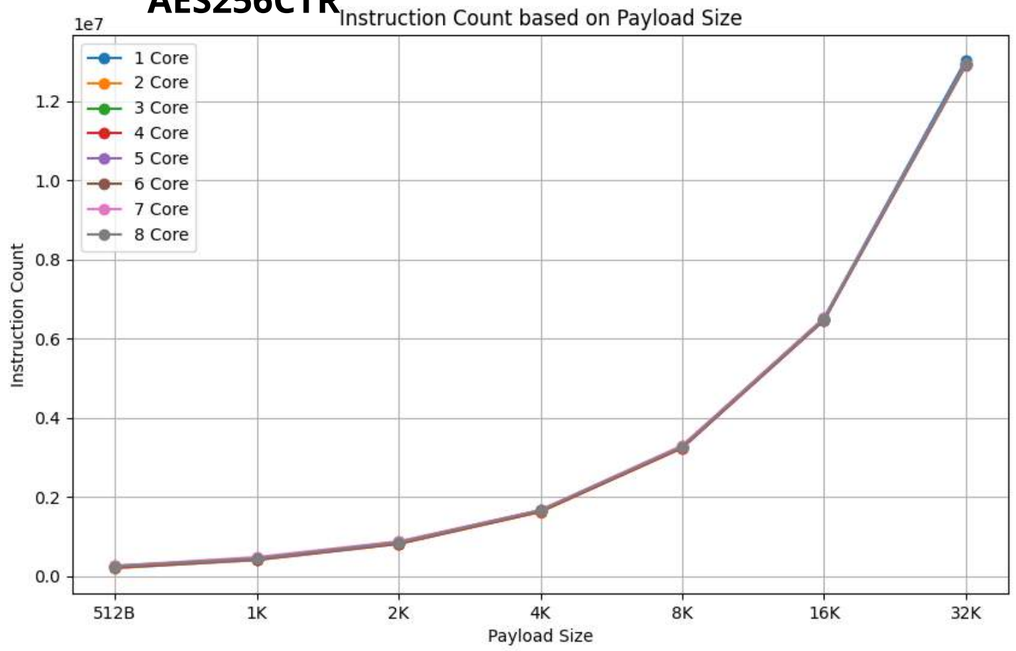
CHACHA20



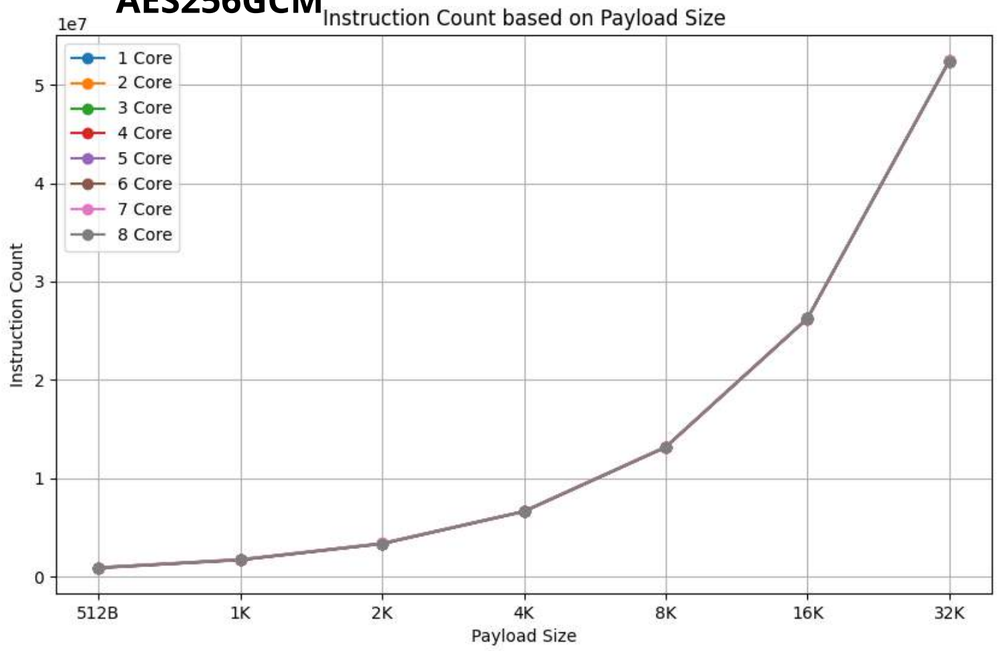
C20P1305



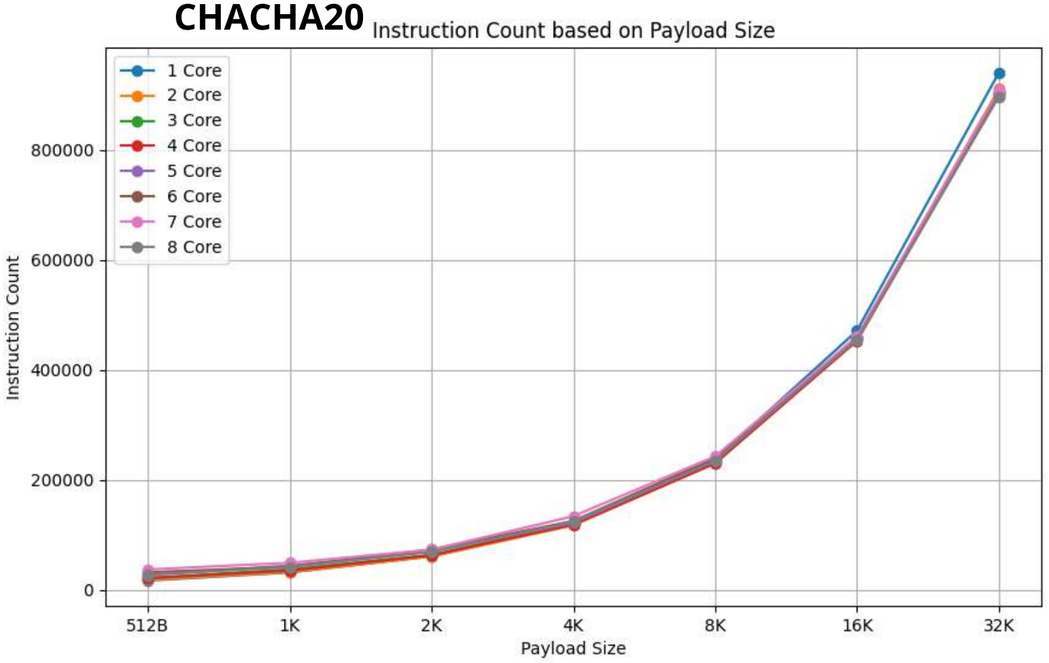
AES256CTR



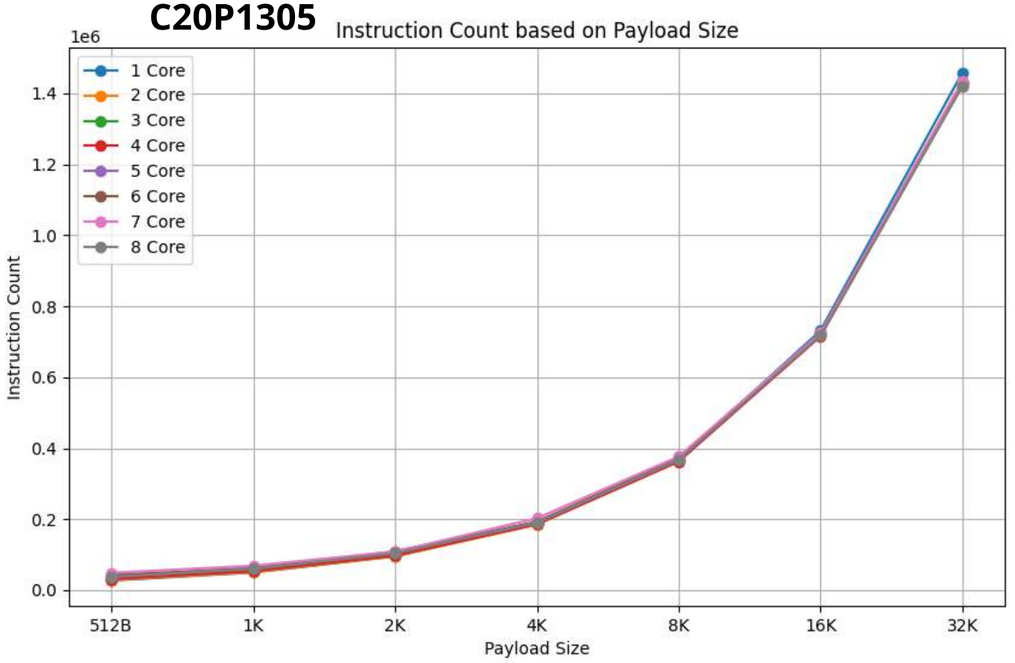
AES256GCM



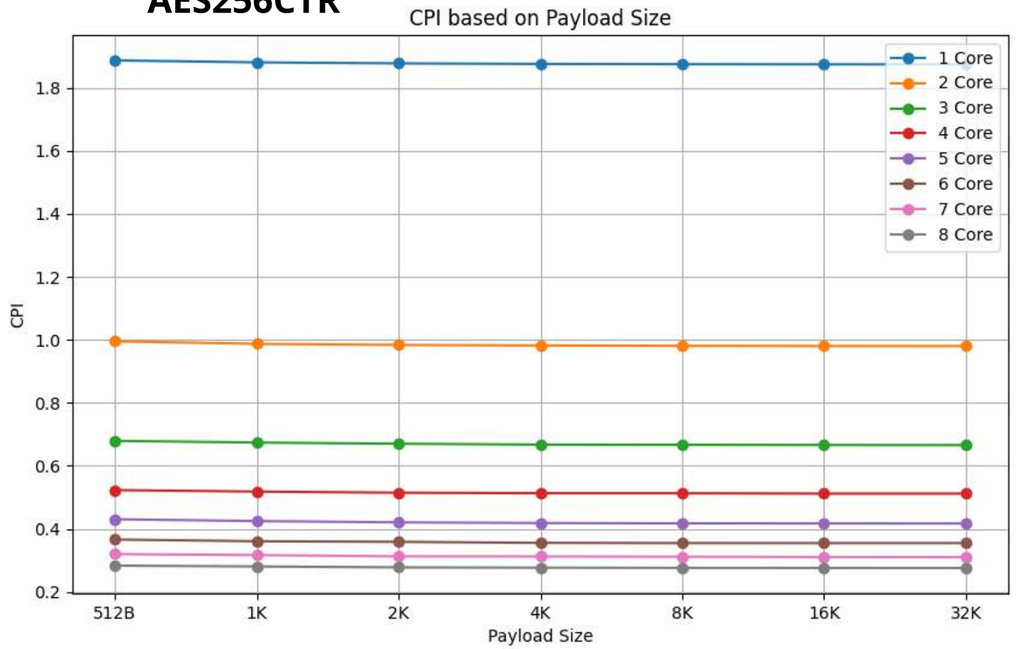
CHACHA20



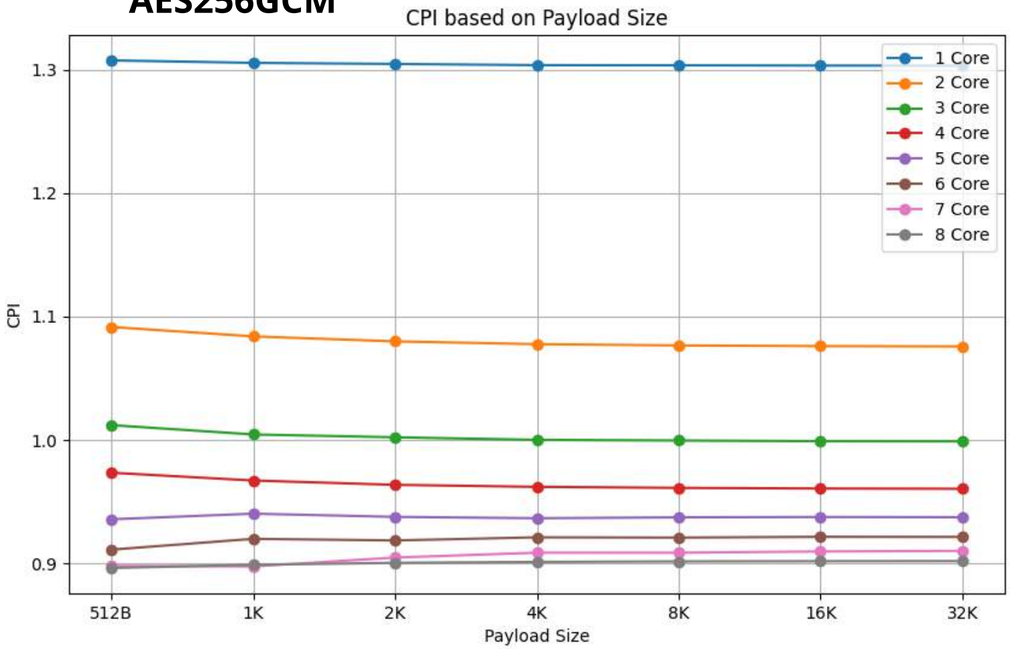
C20P1305



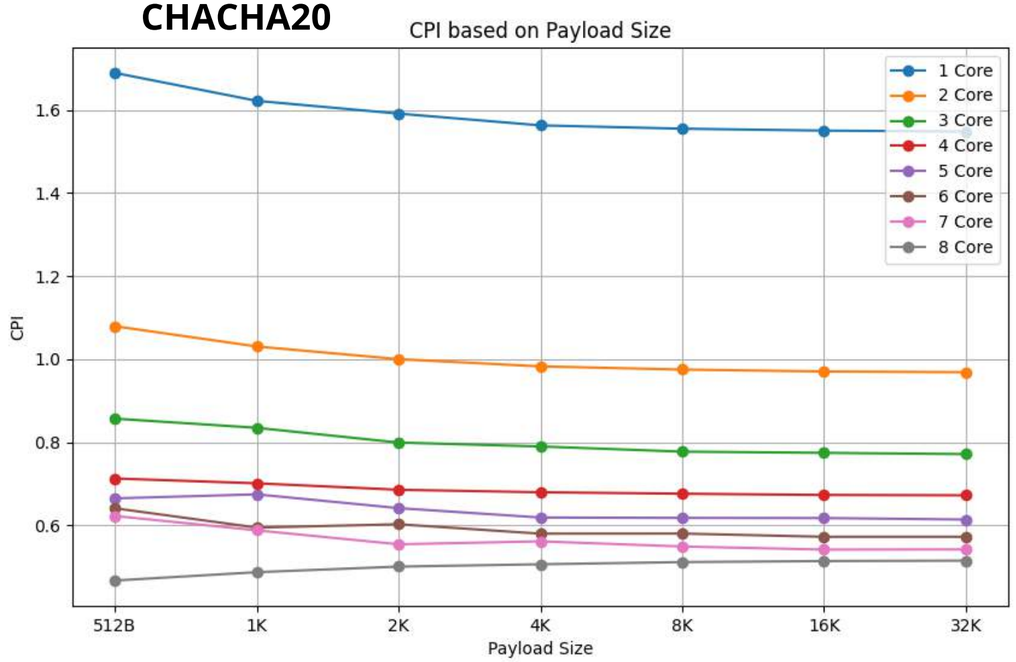
AES256CTR



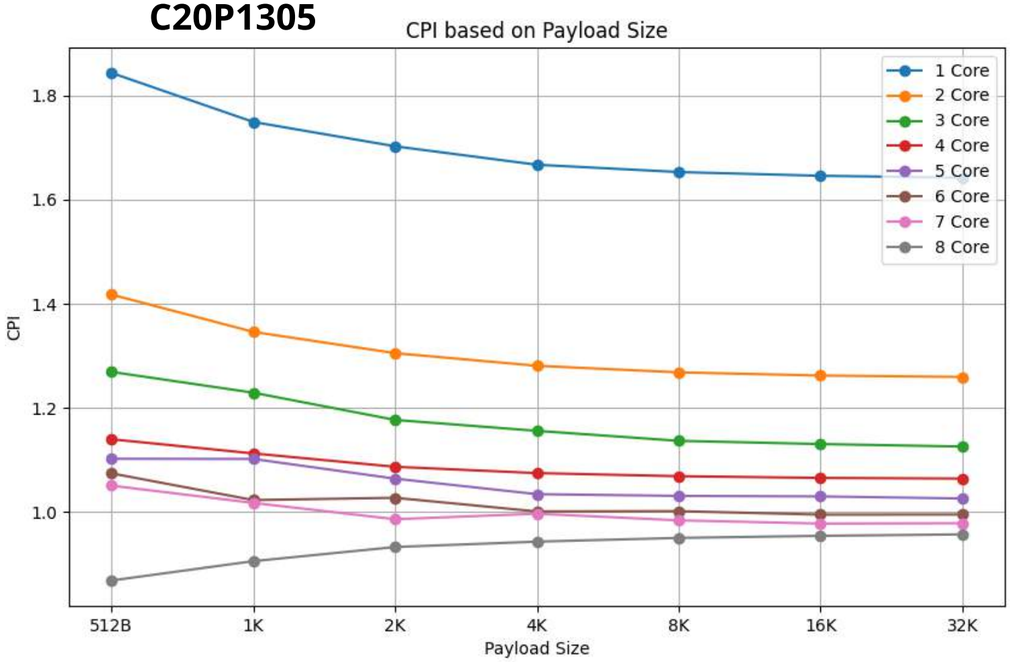
AES256GCM



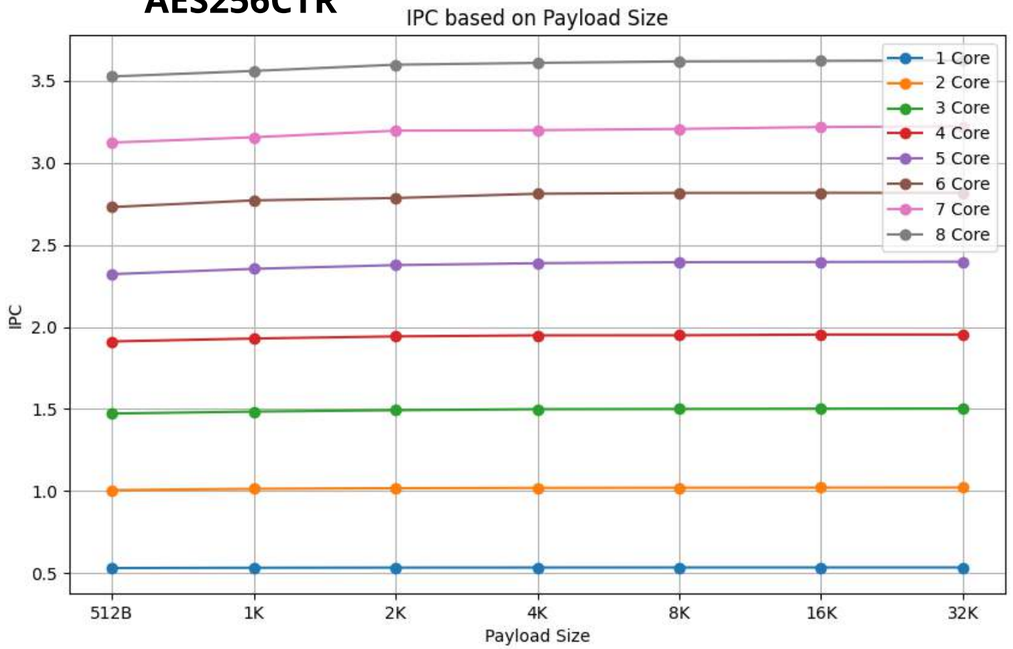
CHACHA20



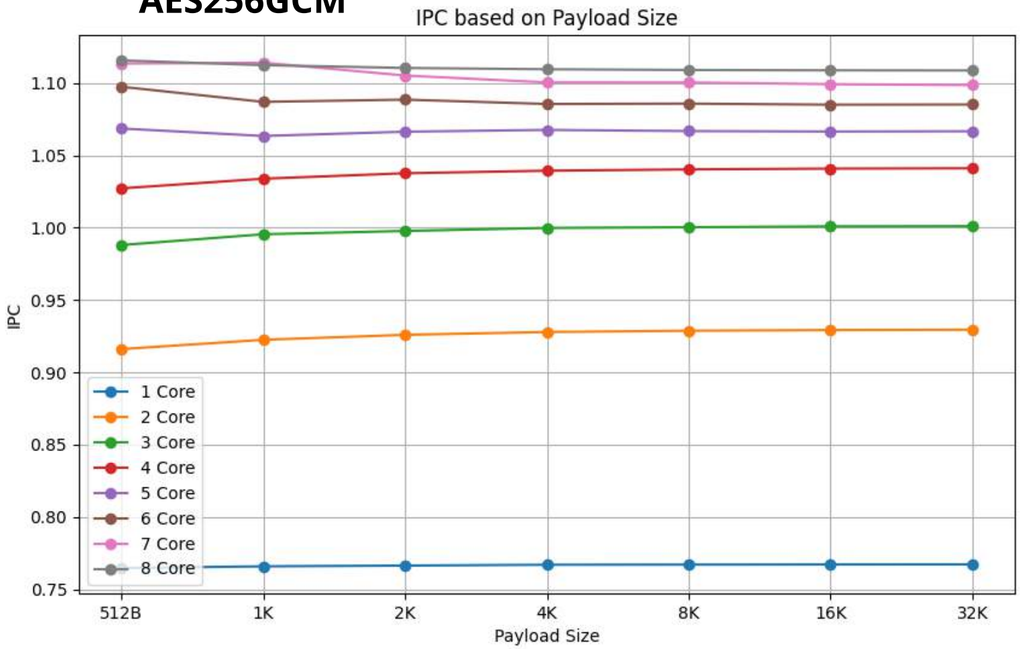
C20P1305



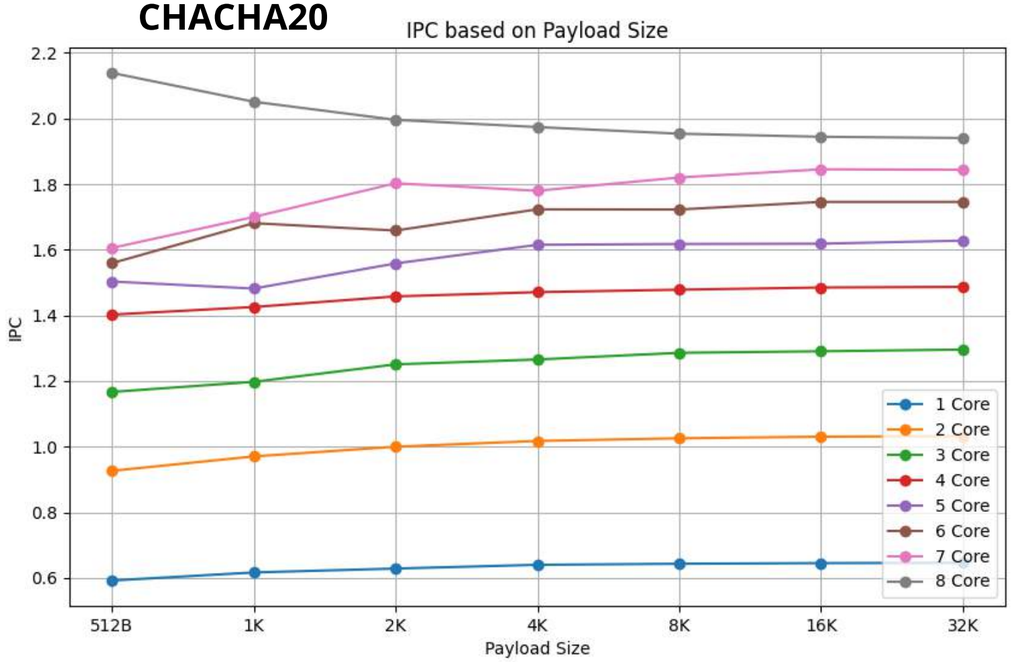
AES256CTR



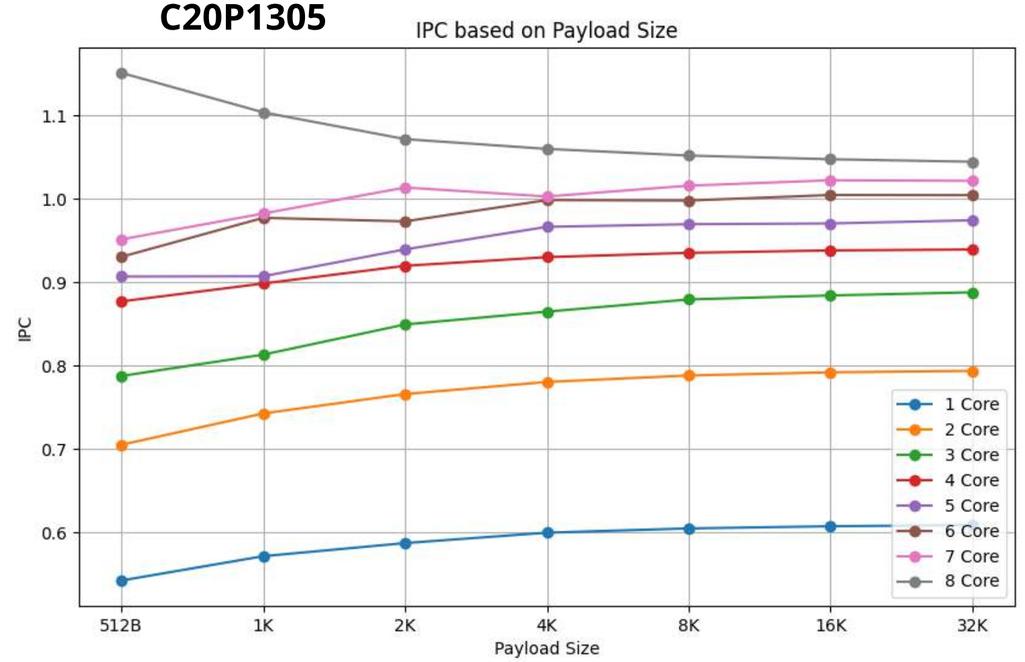
AES256GCM



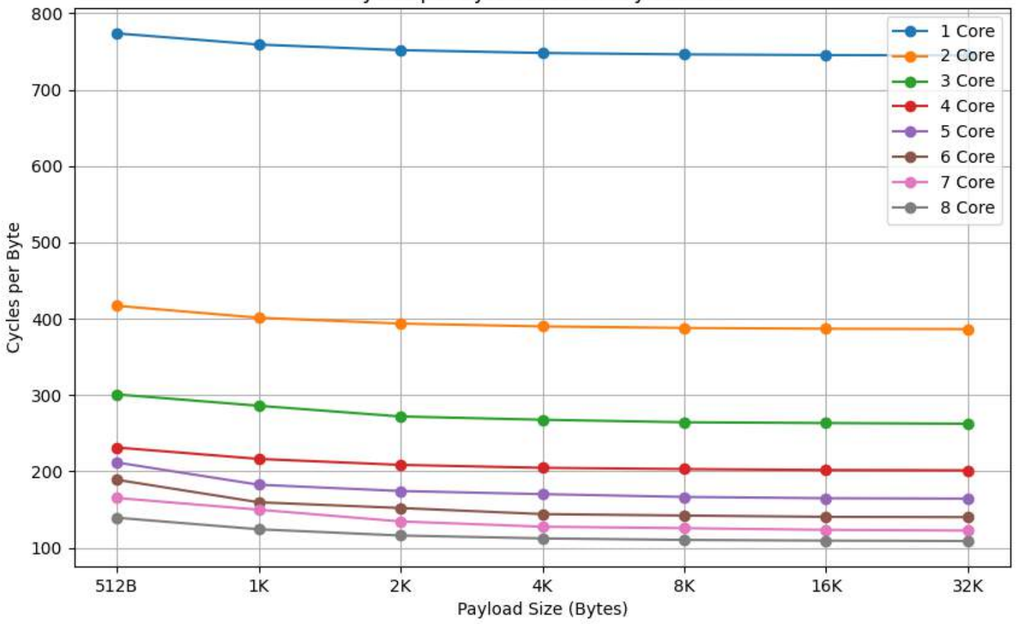
CHACHA20



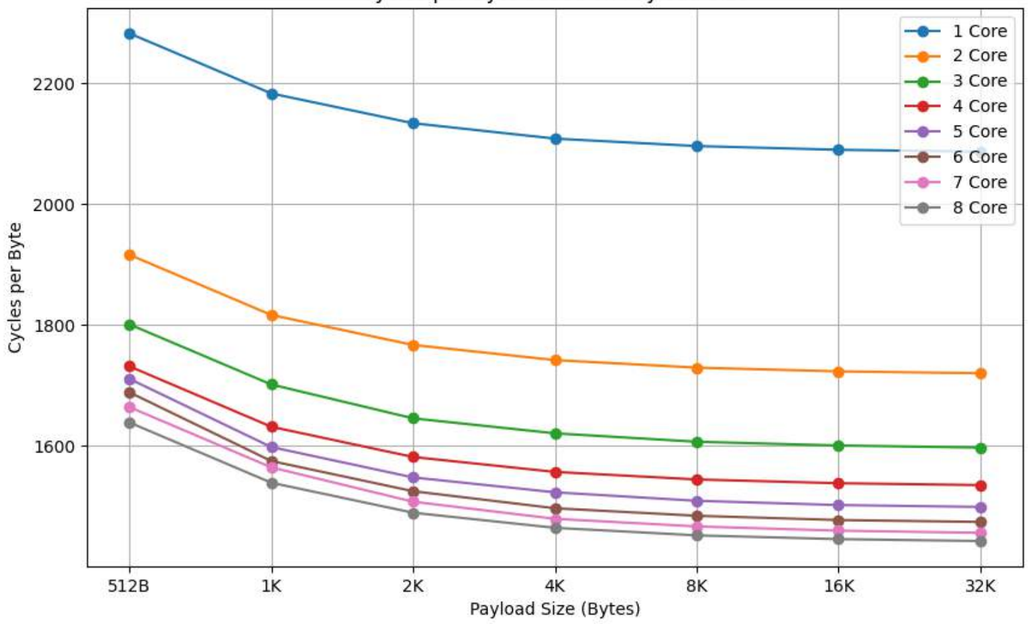
C20P1305



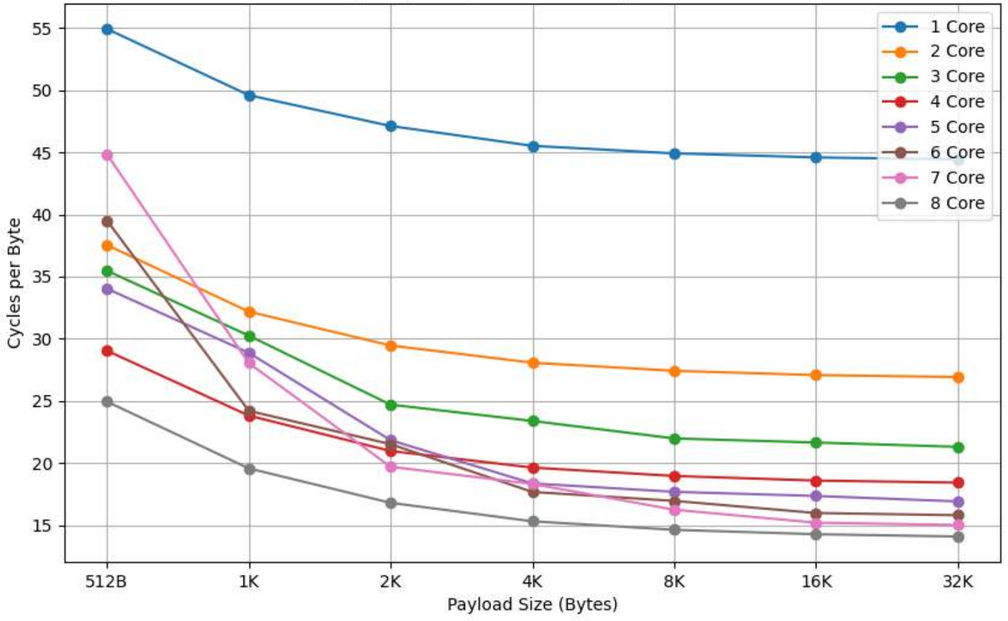
AES256CTR



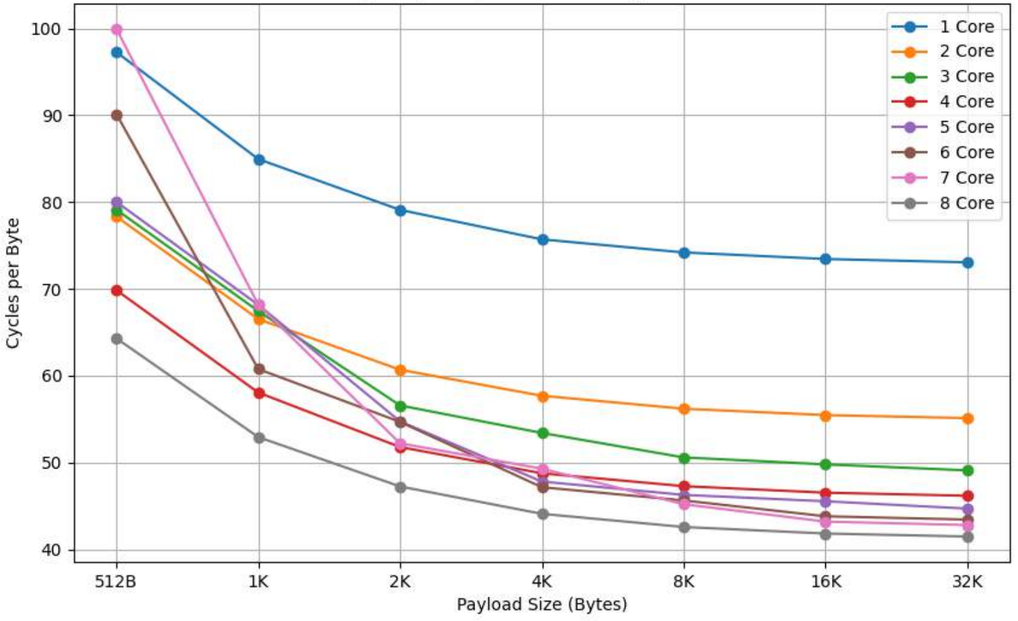
AES256GCM



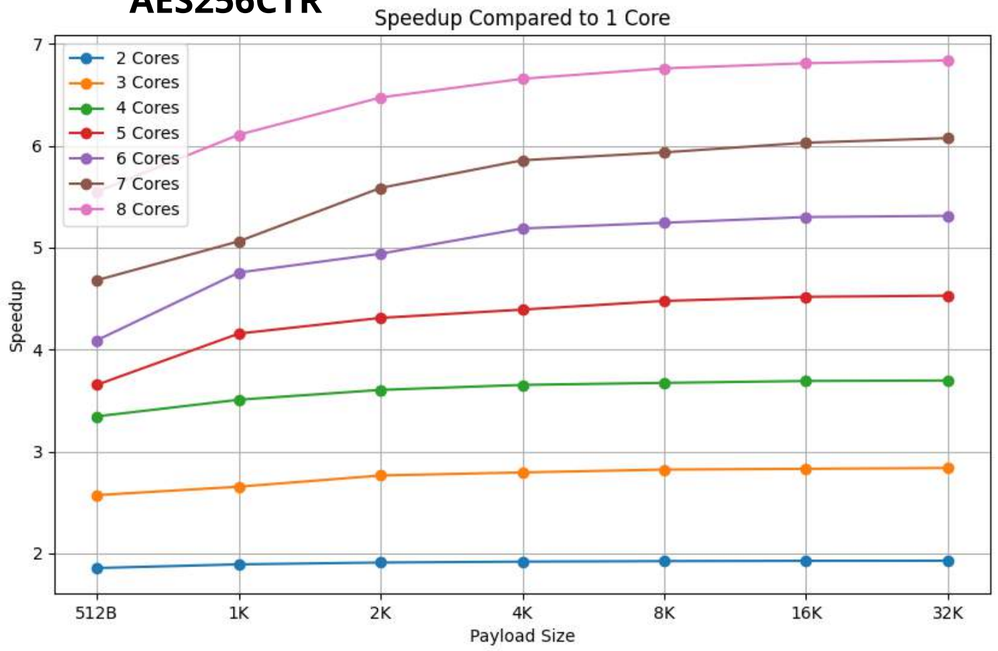
CHACHA20



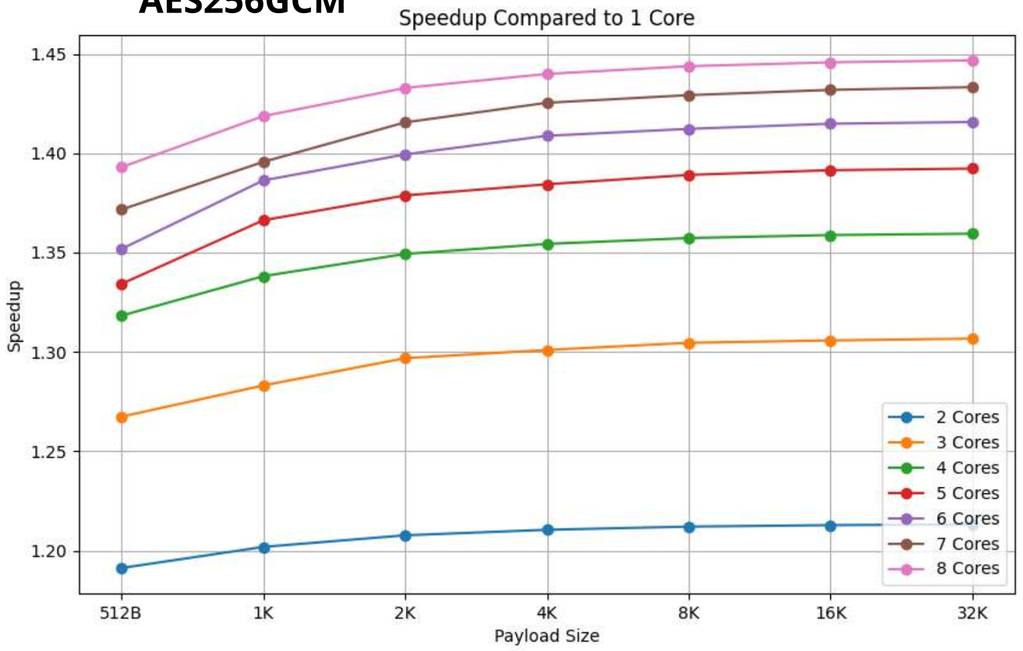
C20P1305



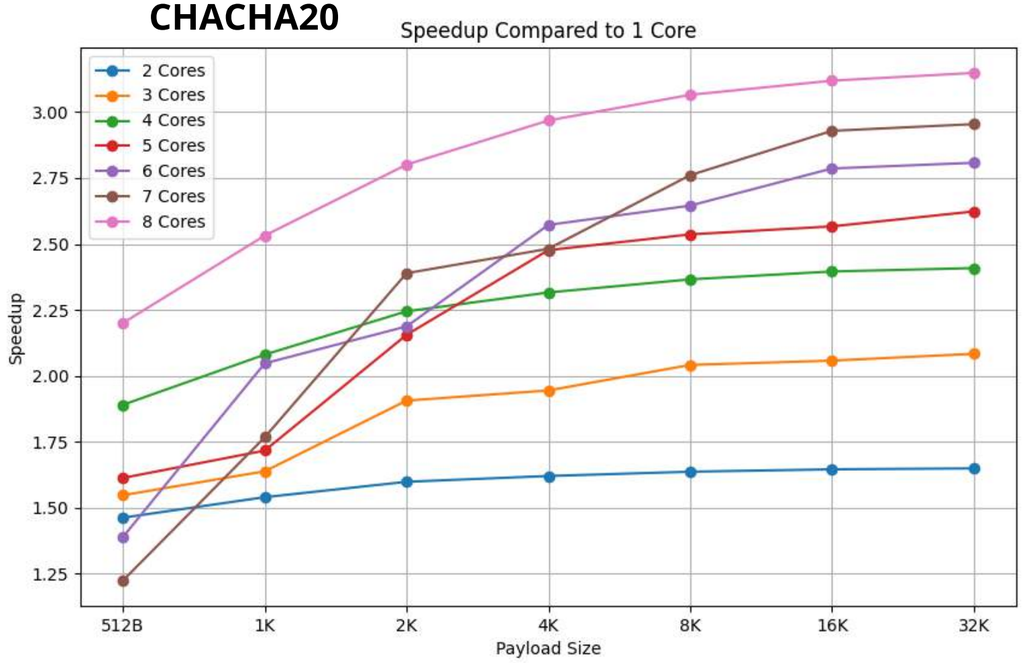
AES256CTR



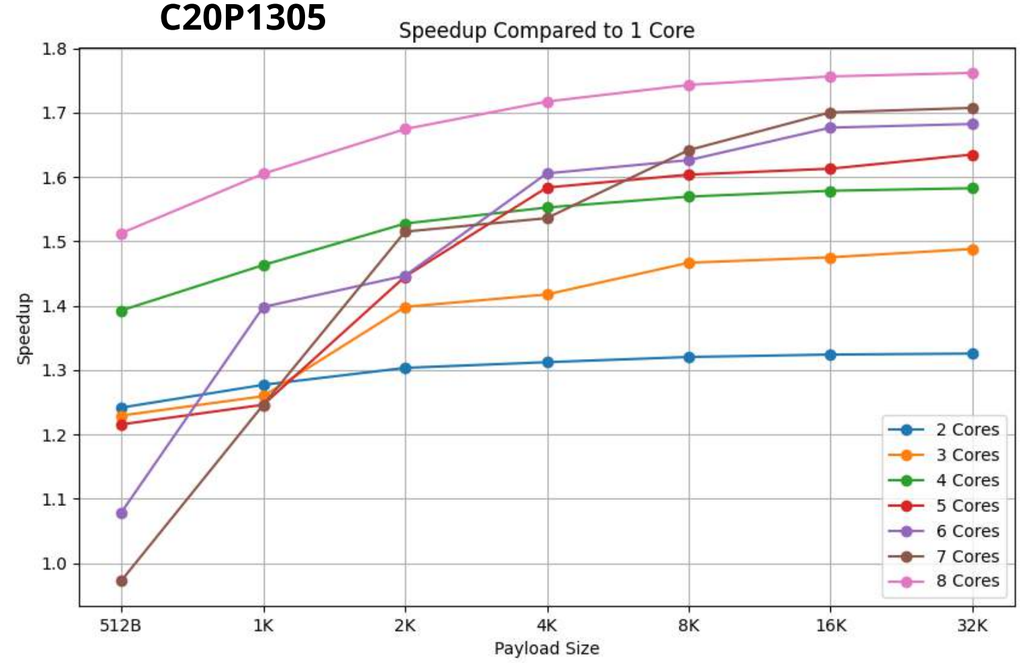
AES256GCM



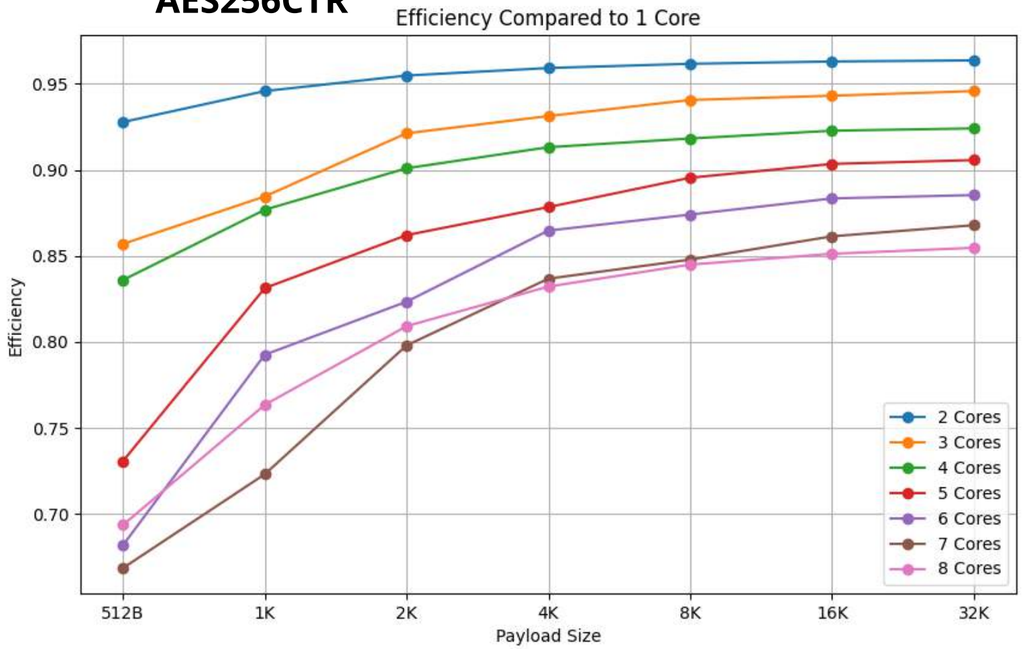
CHACHA20



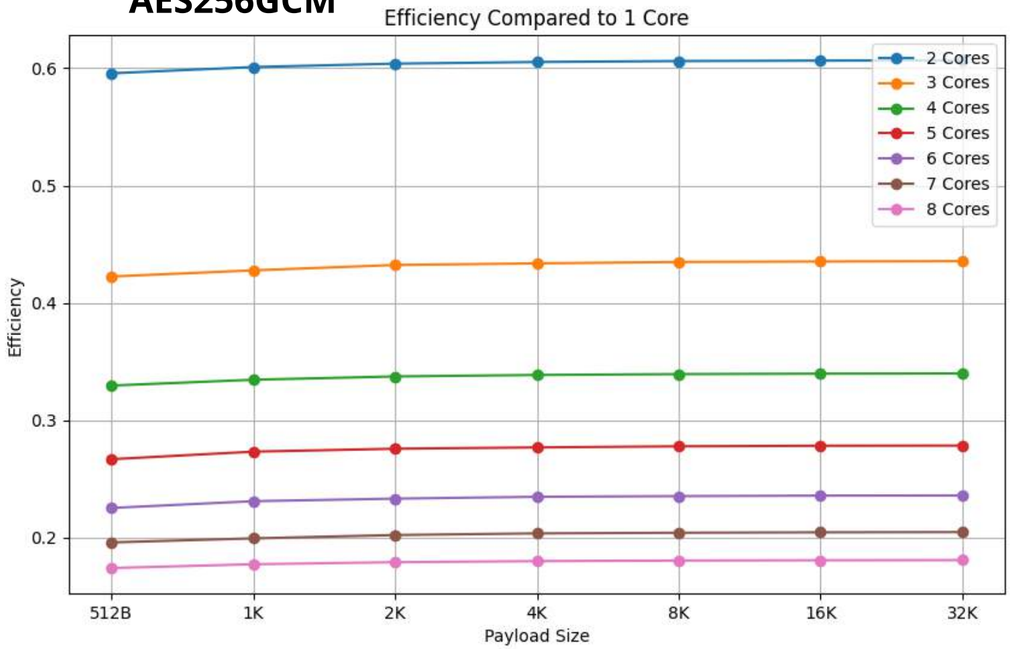
C20P1305



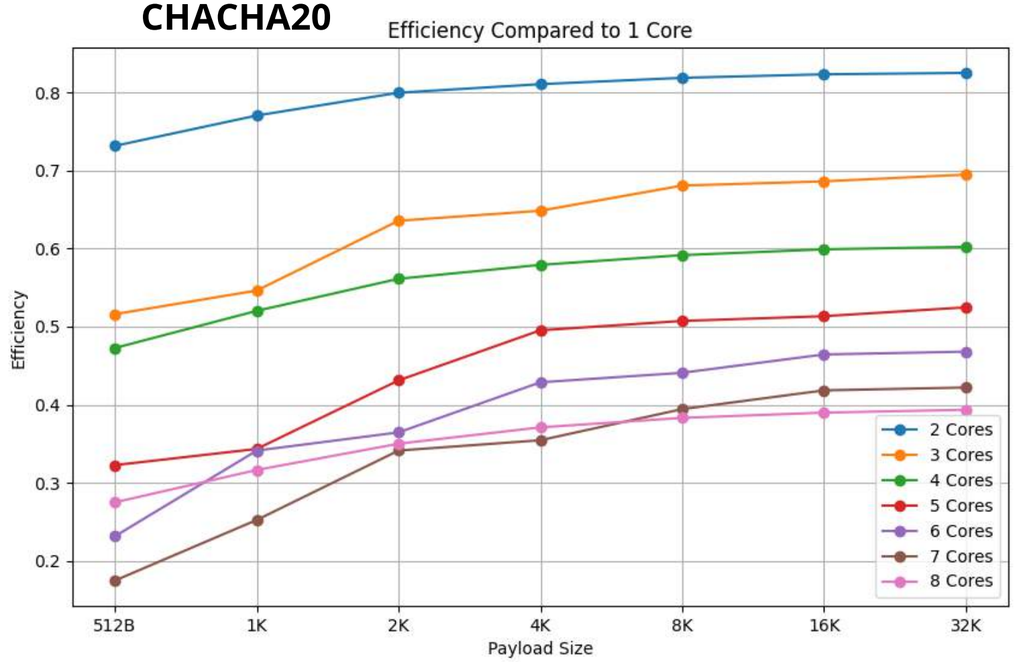
AES256CTR



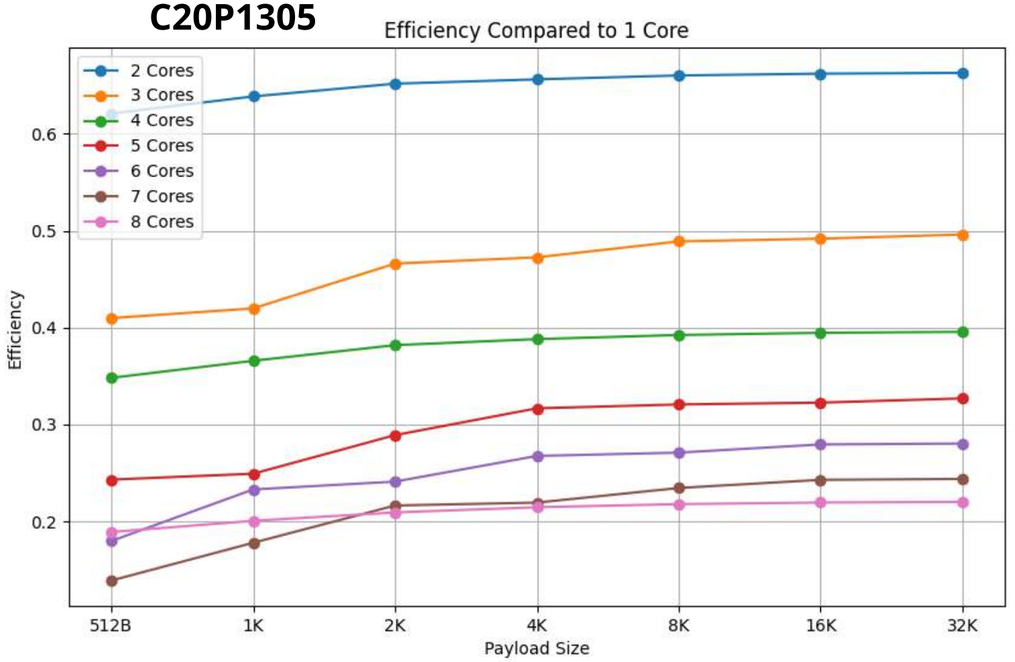
AES256GCM



CHACHA20



C20P1305



NOTE: TRADEOFF EXECUTION/DMA

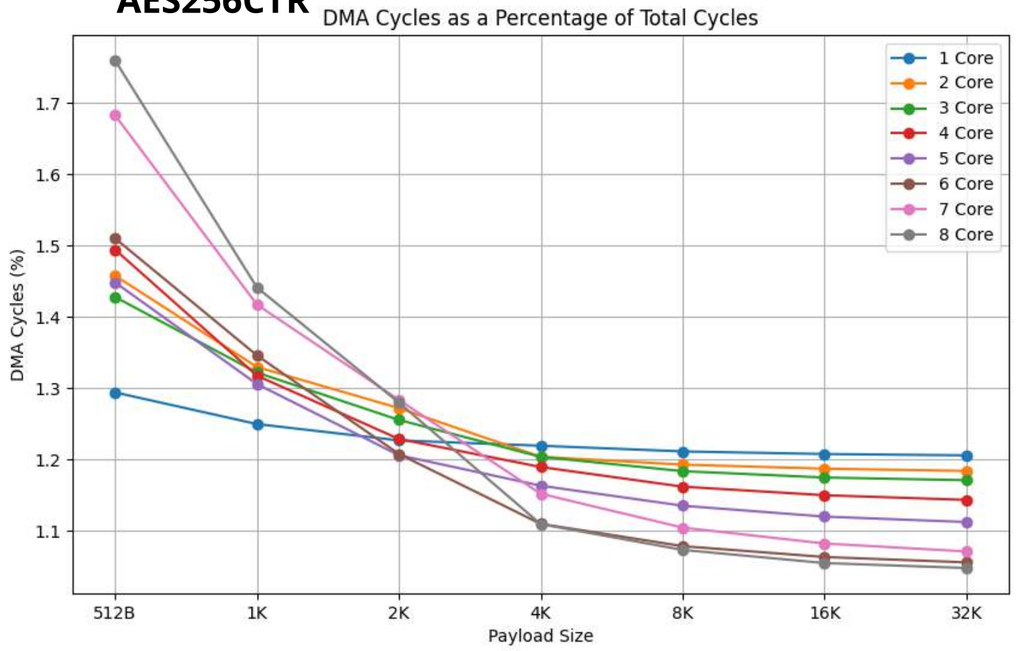
To understand wether the bottleneck of the implementation is the memory management (DMA transactions) or the execution we need to analyze the trace file comparing the sequence of instructions with the “pi_cl_dma_wait” function, which stalls the execution until the memory transfer is completed.

```
static inline void pi_cl_dma_cmd_wait(pi_cl_dma_cmd_t *cmd)
{
  __cl_dma_wait((pi_cl_dma_cmd_t *)cmd);
}

static inline void __cl_dma_wait(pi_cl_dma_cmd_t *copy)
{
  int counter = copy->id;

  eu_mutex_lock_from_id(0);

  while(DMA_READ(MCHAN_STATUS_OFFSET) & (1 << counter)) {
    eu_mutex_unlock_from_id(0);
    eu_evt_maskWaitAndClr(1<<ARCHI_CL_EVT_DMA0);
    eu_mutex_lock_from_id(0);
  }

  plp_dma_counter_free(counter);

  eu_mutex_unlock_from_id(0);
}
```

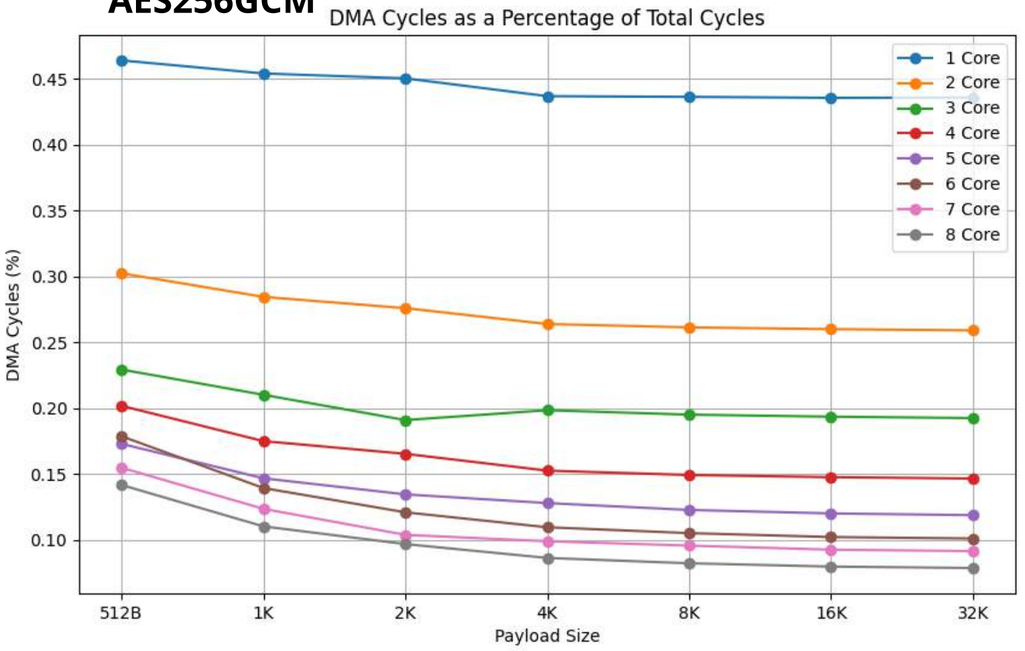
Wait cycles

As expected, by looking at the trace files we never experienced the fulfilment of the while conditions.

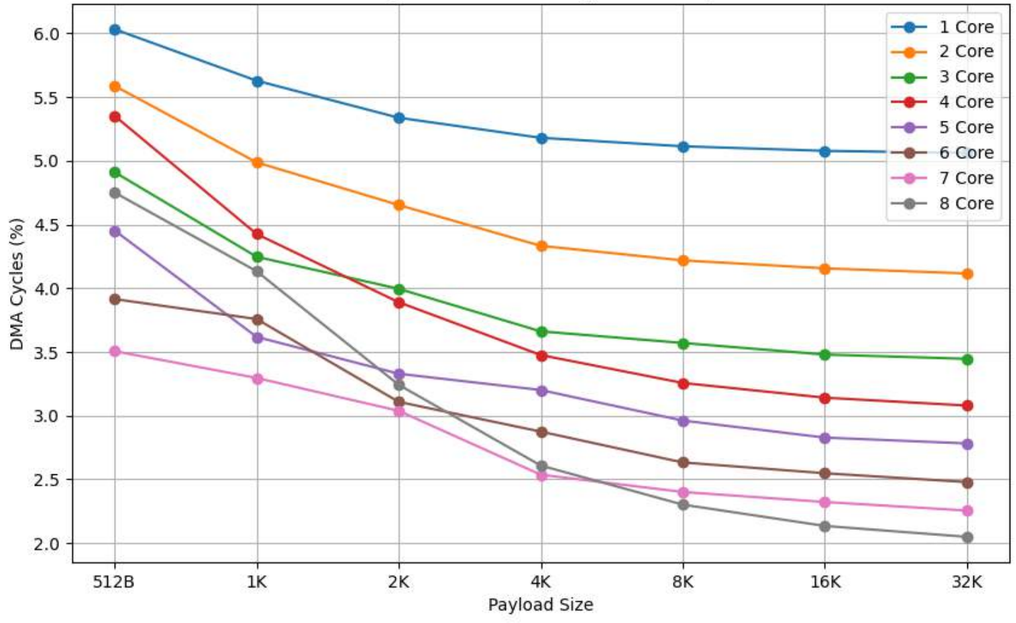
AES256CTR



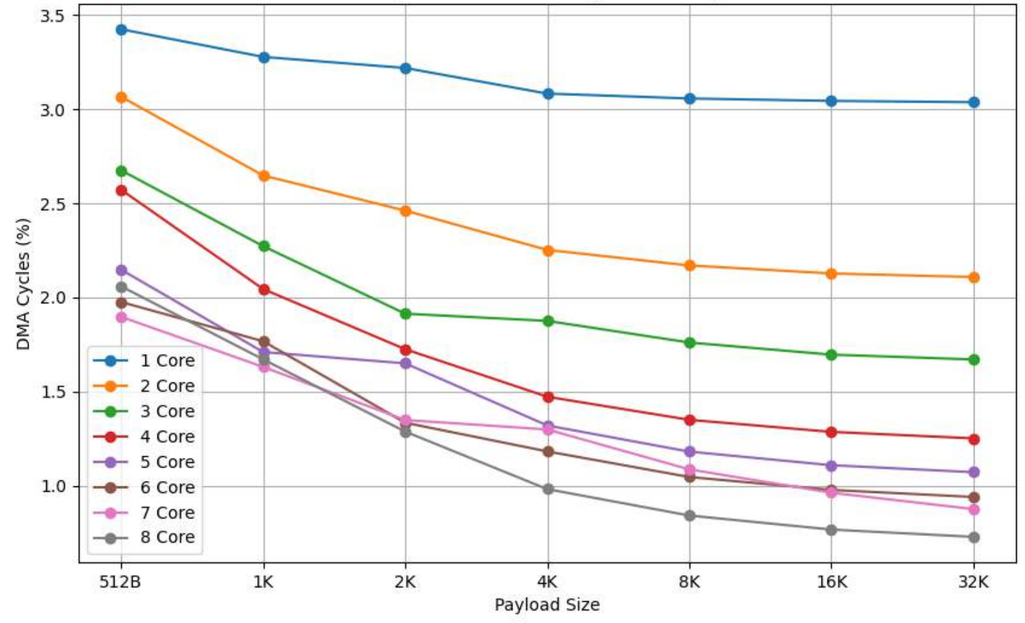
AES256GCM



CHACHA20



C20P1305



CONCLUSION

GAP8 architecture has showed as a suitable platform for stream cipher encryption algorithms, exploiting its 8-cores cluster and multi-channel DMA to maximize performances.

Possible future steps will consider FC-Cluster synchronization to achieve maximum efficiency even in case of Authenticated Encryption algorithms.

THANKS FOR YOUR ATTENTION!

More information in our repo:
https://gitlab.com/ecs-lab/projects/cps-crypto-on-pulp/-/tree/final_version

	Matteo Franchi	Riccardo Gaspari	Giulio Vignati
AES CTR	✓	✓	
AES GCM	✓	✓	
CHACHA20			✓
CHACHA20-POLY1305		✓	✓
PARALLELIZZAZIONE		✓	
SCRIPT PERFORMANCE		✓	✓