Implementation of Light Weight Cryptography(LWC) algorithm Ascon

Surya Raghav.B CS21B2042

I. AIM

This project aims to implement the Ascon lightweight cryptography algorithm in Python and conduct a comprehensive analysis of its functionality, performance, and suitability for deployment in low-powered IoT devices. By exploring its cryptographic properties, resource efficiency, and resistance to various security threats, the project seeks to provide insights into Ascon's potential as a security solution for resource-constrained IoT environments. Through benchmarking, experimentation, and comparative analysis with other cryptographic algorithms, the project aims to elucidate Ascon's strengths and limitations, thereby informing developers and stakeholders about its applicability and effectiveness in securing IoT devices with limited computational resources.

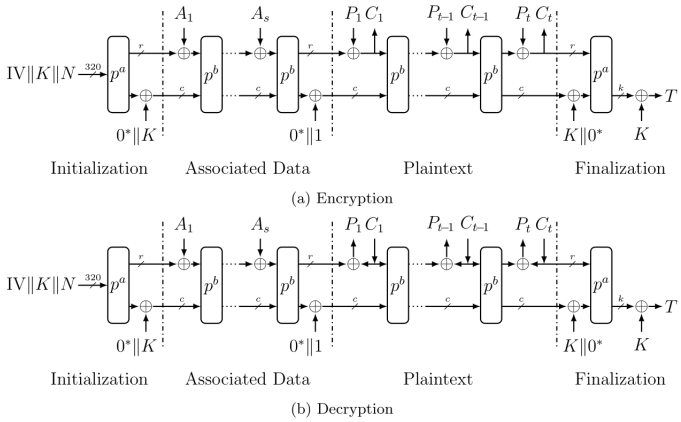
II. Apparatus/Software Required

- Language: Python 3.11.8 is used to implement and benchmark the algorithm.
- Libraries: python libraries such as PyCryptodome(native crypto library), timeit(measuring the run-time of the algorithm), benchmark, memory_profiler and line_profiler, cProfile(to analyze the algorithm).
- Algorithm Specification: The native Ascon algorithm implementation submitted to NIST LWC category used as software reference.
- Development Environment: Visual Studio Code Integrated Development environment.

III. Procedure

A. Introduction

In today's interconnected world, securing data transmitted by IoT devices is paramount, especially considering the proliferation of low-powered devices in various applications. This project aims to explore the feasibility and effectiveness of utilizing the Ascon lightweight cryptography algorithm to secure communication in low-powered IoT environments.



(a) Encryption

(b) Decryption

Fig. 1. Flow chart of the Ascon-128 algorithm

Objective: The primary objective of this project is to implement the Ascon cryptography algorithm in Python and conduct a comprehensive analysis of its functionality, performance, and suitability for

deployment in low-powered IoT devices. Through this analysis, we aim to provide valuable insights into the strengths, weaknesses, and practical considerations of using Ascon in resource-constrained IoT environments.

B. Methodology/Problem Statement

Traditional cryptographic algorithms, while effective, may not be suitable for low-powered IoT devices due to their high computational overhead and memory requirements. Therefore, there is a need to explore lightweight cryptography solutions that can provide robust security while minimizing resource utilization.

The problem addressed by this project is to investigate the feasibility and effectiveness of using the Ascon lightweight cryptography algorithm for securing communication in low-powered IoT environments. The project aims to evaluate Ascon's performance, security properties, and practical use cases in IoT devices to provide insights into its suitability for deployment in real-world scenarios.

Methodology:

- **Literature Review:** Conduct a comprehensive review of existing literature on lightweight cryptography algorithms, IoT security, and Ascon algorithm specifications. Gain insights into the challenges of securing communication in low-powered IoT enironments and identify potential solutions.
- **Ascon Algorithm Implementation:** Implement the Ascon cryptography algorithm in Python based on the specifications provided by the Ascon design team. Develop modules for encryption, decryption, key generation, and other essential cryptographic operations.
- **Performance Analysis:** Perform rigorous performance testing of the Ascon implementation on low-powered hardware platforms commonly used in IoT devices. Measure factors such as execution time, memory usage, and energy consumption. Benchmark Ascon's performance against other lightweight cryptographic algorithms used in IoT applications.
- **Security Evaluation:** Analyze Ascon's security properties, including its resistance to common cryptographic attacks such as differential and linear cryptanalysis. Evaluate the algorithm's strength against various threat models relevant to IoT environments.
- **Use Case Evaluation:** Explore practical use cases of Ascon in low-powered IoT devices, considering factors such as computational overhead, memory requirements, and energy efficiency. Assess the trade-offs between security and performance in different deployment scenarios.
- **Documentation and Reporting:** Document the implementation details, analysis findings, and conclusions of the project. Prepare a comprehensive report summarizing the methodology, results, and recommendations for deploying Ascon in low-powered IoT environments.
- **Validation and Verification:** Validate the implementation and analysis results through peer review, testing, and validation against established cryptographic standards and best practices. Ensure the reliability and reproducibility of the findings.

C. Implementation

1) Hardware Design:

- OS: Arch Linux x86_64
- Host: RedmiBook 15 Pro
- Kernel: 6.8.2-arch2-1
- Shell: bash 5.2.26
- WM: sway
- CPU: 11th Gen Intel i5-11300H (8) @ 4.400GHz
- GPU: Intel TigerLake-LP GT2 [Iris Xe Graphics]
- Memory: 7729MiB

2) Executable Code: The python implemention of the Ascon-128 is given below

```python
# ASCON-128
# Surya Raghav B
# CS21B2042

def ascon_encrypt(key, nonce, associateddata, plaintext):
    S = [0, 0, 0, 0, 0]
    k = len(key) * 8
    a = 12
    b = 6
    rate = 8

    ascon_initialize(S, k, rate, a, b, key, nonce)
    ascon_process_associated_data(S, b, rate, associateddata)
    ciphertext = ascon_process_plaintext(S, b, rate, plaintext)
    tag = ascon_finalize(S, rate, a, key)
    return ciphertext + tag


def ascon_decrypt(key, nonce, associateddata, ciphertext):
    S = [0, 0, 0, 0, 0]
    k = len(key) * 8
    a = 12
    b = 6
    rate = 8

    ascon_initialize(S, k, rate, a, b, key, nonce)
    ascon_process_associated_data(S, b, rate, associateddata)
    plaintext = ascon_process_ciphertext(S, b, rate, ciphertext[:-16])
    tag = ascon_finalize(S, rate, a, key)
    if tag == ciphertext[-16:]:
        return plaintext
    else:
        return None


def ascon_initialize(S, k, rate, a, b, key, nonce):
    iv_zero_key_nonce = (
        to_bytes([k, rate * 8, a, b]) + zero_bytes(20 - len(key)) + key + nonce
    )
    S[0], S[1], S[2], S[3], S[4] = bytes_to_state(iv_zero_key_nonce)

    ascon_permutation(S, a)

    zero_key = bytes_to_state(zero_bytes(40 - len(key)) + key)
    S[0] ^= zero_key[0]
    S[1] ^= zero_key[1]
    S[2] ^= zero_key[2]
    S[3] ^= zero_key[3]
    S[4] ^= zero_key[4]


def ascon_process_associated_data(S, b, rate, associateddata):
    if len(associateddata) > 0:
        a_padding = to_bytes([0x80]) + zero_bytes(
            rate - (len(associateddata) % rate) - 1
        )
        a_padded = associateddata + a_padding

        for block in range(0, len(a_padded), rate):
            S[0] ^= bytes_to_int(a_padded[block : block + 8])
            if rate == 16:
                S[1] ^= bytes_to_int(a_padded[block + 8 : block + 16])

            ascon_permutation(S, b)
```

```
    S[4] ^= 1


def ascon_process_plaintext(S, b, rate, plaintext):
    p_lastlen = len(plaintext) % rate
    p_padding = to_bytes([0x80]) + zero_bytes(rate - p_lastlen - 1)
    p_padded = plaintext + p_padding

    # first t-1 blocks
    ciphertext = to_bytes([])
    for block in range(0, len(p_padded) - rate, rate):
        if rate == 8:
            S[0] ^= bytes_to_int(p_padded[block : block + 8])
            ciphertext += int_to_bytes(S[0], 8)
        elif rate == 16:
            S[0] ^= bytes_to_int(p_padded[block : block + 8])
            S[1] ^= bytes_to_int(p_padded[block + 8 : block + 16])
            ciphertext += int_to_bytes(S[0], 8) + int_to_bytes(S[1], 8)

        ascon_permutation(S, b)

    # last block t
    block = len(p_padded) - rate
    if rate == 8:
        S[0] ^= bytes_to_int(p_padded[block : block + 8])
        ciphertext += int_to_bytes(S[0], 8)[:p_lastlen]
    elif rate == 16:
        S[0] ^= bytes_to_int(p_padded[block : block + 8])
        S[1] ^= bytes_to_int(p_padded[block + 8 : block + 16])
        ciphertext += (
            int_to_bytes(S[0], 8)[: min(8, p_lastlen)]
            + int_to_bytes(S[1], 8)[: max(0, p_lastlen - 8)]
        )
    return ciphertext


def ascon_process_ciphertext(S, b, rate, ciphertext):
    c_lastlen = len(ciphertext) % rate
    c_padded = ciphertext + zero_bytes(rate - c_lastlen)

    # first t-1 blocks
    plaintext = to_bytes([])
    for block in range(0, len(c_padded) - rate, rate):
        if rate == 8:
            Ci = bytes_to_int(c_padded[block : block + 8])
            plaintext += int_to_bytes(S[0] ^ Ci, 8)
            S[0] = Ci
        elif rate == 16:
            Ci = (
                bytes_to_int(c_padded[block : block + 8]),
                bytes_to_int(c_padded[block + 8 : block + 16]),
            )
            plaintext += int_to_bytes(S[0] ^ Ci[0], 8) + int_to_bytes(S[1] ^ Ci[1], 8)
            S[0] = Ci[0]
            S[1] = Ci[1]

        ascon_permutation(S, b)

    # last block t
    block = len(c_padded) - rate
    if rate == 8:
        c_padding1 = 0x80 << (rate - c_lastlen - 1) * 8
        c_mask = 0xFFFFFFFFFFFFFFFF >> (c_lastlen * 8)
        Ci = bytes_to_int(c_padded[block : block + 8])
```

```
        plaintext += int_to_bytes(Ci ^ S[0], 8)[:c_lastlen]
        S[0] = Ci ^ (S[0] & c_mask) ^ c_padding1
    elif rate == 16:
        c_lastlen_word = c_lastlen % 8
        c_padding1 = 0x80 << (8 - c_lastlen_word - 1) * 8
        c_mask = 0xFFFFFFFFFFFFFFFF >> (c_lastlen_word * 8)
        Ci = (
            bytes_to_int(c_padded[block : block + 8]),
            bytes_to_int(c_padded[block + 8 : block + 16]),
        )
        plaintext += (int_to_bytes(S[0] ^ Ci[0], 8) + int_to_bytes(S[1] ^ Ci[1], 8))[
            :c_lastlen
        ]
        if c_lastlen < 8:
            S[0] = Ci[0] ^ (S[0] & c_mask) ^ c_padding1
        else:
            S[0] = Ci[0]
            S[1] = Ci[1] ^ (S[1] & c_mask) ^ c_padding1
    return plaintext


def ascon_finalize(S, rate, a, key):
    assert len(key) in [16, 20]
    S[rate // 8 + 0] ^= bytes_to_int(key[0:8])
    S[rate // 8 + 1] ^= bytes_to_int(key[8:16])
    S[rate // 8 + 2] ^= bytes_to_int(key[16:] + zero_bytes(24 - len(key)))

    ascon_permutation(S, a)

    S[3] ^= bytes_to_int(key[-16:-8])
    S[4] ^= bytes_to_int(key[-8:])
    tag = int_to_bytes(S[3], 8) + int_to_bytes(S[4], 8)
    return tag


def ascon_permutation(S, rounds=1):
    assert rounds <= 12
    for r in range(12 - rounds, 12):
        # --- add round constants ---
        S[2] ^= 0xF0 - r * 0x10 + r * 0x1

        # --- substitution layer ---
        S[0] ^= S[4]
        S[4] ^= S[3]
        S[2] ^= S[1]
        T = [(S[i] ^ 0xFFFFFFFFFFFFFFFF) & S[(i + 1) % 5] for i in range(5)]
        for i in range(5):
            S[i] ^= T[(i + 1) % 5]
        S[1] ^= S[0]
        S[0] ^= S[4]
        S[3] ^= S[2]
        S[2] ^= 0xFFFFFFFFFFFFFFFF

        # --- linear diffusion layer ---
        S[0] ^= rotr(S[0], 19) ^ rotr(S[0], 28)
        S[1] ^= rotr(S[1], 61) ^ rotr(S[1], 39)
        S[2] ^= rotr(S[2], 1) ^ rotr(S[2], 6)
        S[3] ^= rotr(S[3], 10) ^ rotr(S[3], 17)
        S[4] ^= rotr(S[4], 7) ^ rotr(S[4], 41)


def get_random_bytes(num):
    import os

    return to_bytes(os.urandom(num))
```

```
195
196
197 def zero_bytes(n):
198     return n * b"\x00"
199
200
201 def to_bytes(l):
202     return bytes(bytearray(l))
203
204
205 def bytes_to_int(bytes):
206     return sum(
207         [bi << ((len(bytes) - 1 - i) * 8) for i, bi in enumerate(to_bytes(bytes))]
208     )
209
210
211 def bytes_to_state(bytes):
212     return [bytes_to_int(bytes[8 * w : 8 * (w + 1)]) for w in range(5)]
213
214
215 def int_to_bytes(integer, nbytes):
216     return to_bytes([(integer >> ((nbytes - 1 - i) * 8)) % 256 for i in range(nbytes)])
217
218
219 def rotr(val, r):
220     return (val >> r) | ((val & (1 << r) - 1) << (64 - r))
221
222
223 def bytes_to_hex(b):
224     return b.hex()
225
226
227 def print_text(data):
228     maxlen = max([len(text) for (text, val) in data])
229     for text, val in data:
230         print(
231             "{text}:{align} 0x{val} ({length} bytes)".format(
232                 text=text,
233                 align=((maxlen - len(text)) * " "),
234                 val=bytes_to_hex(val),
235                 length=len(val),
236             )
237         )
238
239
240 def ascon():
241     keysize = 16
242     print("=== demo encryption ===")
243
244     key = get_random_bytes(keysize)
245     nonce = get_random_bytes(16)
246
247     associateddata = b"how are you"
248     plaintext = b"hello"
249
250     ciphertext = ascon_encrypt(key, nonce, associateddata, plaintext)
251     receivedplaintext = ascon_decrypt(key, nonce, associateddata, ciphertext)
252
253     if receivedplaintext == None:
254         print("verification failed!")
255
256     print_text(
257         [
258             ("key", key),
259             ("nonce", nonce),
```

```
            ("plaintext", plaintext),
            ("ass.data", associateddata),
            ("ciphertext", ciphertext[:-16]),
            ("tag", ciphertext[-16:]),
            ("received", receivedplaintext),
        ]
    )


ascon()
```

3) Software Implementation:

- **Parameters** The family members are parametrized by the key length $k \leq 128$ bits, the rate r and internal round numbers a and b. The inputs for the encryption procedure E are the plaintext P , associated data A, a secret key K with k bits and a public message number (nonce) N with k bits.

$$E_{a,b,k,r}(K, N, A, P) = (C, T)$$

The decryption and verification procedure D takes as input the key K, nonce N , associated data A, ciphertext C and tag T , and outputs the plaintext P if the verification of the tag is correct or $\perp$ if the verification of the tag fails

$$D_{a,b,k,r}(K, N, A, C, T) \in (P, \perp)$$

- **Padding** Ascon has a message block size of r bits. The padding process appends a single 1 and the smallest number of 0s to the plaintext P such that the length of the padded plaintext is a multiple of r bits. The resulting padded plaintext is split into t blocks of r bits. The same is done to the Associated data A.
- **Initialization** The 320-bit initial state of Ascon is formed by the secret key K and nonce N (both k bits), as well as an IV specifying the algorithm (including the key size k, the rate r, the initialization and finalization round number a, and the intermediate round number b, each written as an 8-bit integer)

$$IV = k \parallel r \parallel a \parallel 0^{288-2k} \parallel$$

$$S = IV \parallel K \parallel N$$

In the initialization, a rounds of the round transformation p are applied to the initial state, followed by an xor of the secret key K

$$S \leftarrow p^a(S) \oplus (0^{320-k}) \parallel K$$

- **Processing Associated Data** Each (padded) associated data block Ai with i = 1, . . . , s is processed as follows. The block Ai is xored to the first r bits Sr of the internal state S. Then, the whole state S is transformed by the permutation pb using b rounds

$$S \leftarrow p^b((S_r \oplus A_i \parallel S_c)$$

Later a single-bit domain separation constant is xored to the internal state S

$$s \leftarrow S \oplus (0^{319} \parallel 1)$$

- **Processing Plaintext/Ciphertext**
Encryption In each iteration, one (padded) plaintext block Pi with i = 1, ..., t is xored to the first r bits Sr of the internal state S, followed by the extraction of one ciphertext block Ci . For each block except the last one, the whole internal state S is transformed by the permutation pb using b rounds

$$C_i \leftarrow S_r \oplus P_i$$

$$S \leftarrow p^b(C_i \parallel S_c), \quad if \quad 1 \leq i$$

$$S \leftarrow C_i \parallel S_c, \quad if \quad 1 \leq i = t$$

The last ciphertext block is truncated to the unpadded length of the last plaintext block- fragment

$$C_t \leftarrow \lfloor C_t \rfloor_l$$

Thus, the length of the last ciphertext block Ct is between 0 and r-1 bits, and the total length of the ciphertext C is exactly the same as for the original plaintext P .

Decryption In each iteration except the last one, the plaintext block Pi is computed by xoring the ciphertext block Ci with the first r bits Sr of the internal state. Then, the first r bits of the internal state, Sr , are replaced by Ci . Finally, for each ciphertext block except the last one, the internal state is transformed by b rounds of the permutation pb

$$P_i \leftarrow S_r \oplus C_i$$

$$S \leftarrow p^b(C_i \parallel S_c), \quad 1 \leq i < t$$

For the last, truncated ciphertext block, the procedure differs slightly

$$P_t \leftarrow \lfloor S_r \rfloor_l \oplus C_t$$

$$S \leftarrow C_t \parallel (\lceil S_r \rceil^{r-l} \oplus (1 \parallel 0^{r-1-l})) \parallel S_c$$

- **Finalization** In the finalization, the secret key K is xored to the internal state and the state is transformed by the permutation pa using a rounds. The tag T consists of the last k bits of the state xored with the key K

$$S \leftarrow p^a(S \oplus (0^r \parallel K \parallel 0^{c-k}))$$

$$T \leftarrow \lceil S \rceil_k \oplus K$$

The encryption algorithm returns the tag T together with the ciphertext C1 , . . . , Ct . The decryption algorithm returns the plaintext P1 , . . . , Pt only if the calculated tag value matches the received tag value.

- **Permutations** The permutations iteratively apply an SPN-based round transformation p that in turn consists of three subtransformations constant addition(pc), substitution layer(ps) and linear diffusion layer(pl). For the description and application of the round transformations, the 320-bit state S is split into five 64-bit registers words
- **Addition of Constants** Each round p starts with the constant-addition operation pC which adds a round constants cr to the register word x2 of the state S

$$x_2 \leftarrow x_2 \oplus c_r$$

- **Substitution Layer** In the substitution layer pS , 64 parallel applications of the 5-bit S-box S(x) are performed on the 320-bit state. The S-box will typically be implemented in its bitsliced form, with operations performed on the entire 64-bit words.
- **Linear Diffusion Layer** The linear diffusion layer pL of Ascon is used to provide diffusion within each of the five 64-bit register words xi of the 320-bit state S.

$$\Sigma_0(x_0) = x_0 \oplus (x_0 \gg 19) \oplus (x_0 \gg 28)$$

$$\Sigma_1(x_1) = x_1 \oplus (x_1 \gg 61) \oplus (x_1 \gg 39)$$

$$\Sigma_2(x_2) = x_2 \oplus (x_2 \gg 1) \oplus (x_2 \gg 6)$$

$$\Sigma_3(x_3) = x_3 \oplus (x_3 \gg 10) \oplus (x_3 \gg 17)$$

$$\Sigma_4(x_4) = x_4 \oplus (x_4 \gg 7) \oplus (x_4 \gg 41)$$

IV. Results/Output

We use profiling methods such as timeit, memory profiler, cprofile to benchmark the Ascon algorithm.

- **Timeit module** The timeit module in Python is a convenient tool for measuring the execution time of small code snippets or functions. It provides a simple way to time the execution of Python code, allowing you to compare the performance of different implementations or optimizations.
- **Cprofile module** cProfile is a built-in Python module used for profiling Python programs. Profiling involves analyzing the performance characteristics of a program, such as the time taken by various functions and methods, the number of times they are called, and memory usage. Profiling helps in identifying performance bottlenecks and optimizing code for better efficiency.
- **Memory profiler module** The memory profiler module is a third-party Python tool used for profiling memory usage in Python programs. It allows you to monitor how much memory your Python code consumes and identify memory-intensive parts of your program. One of the key features of memory profiler is its ability to profile memory usage line by line. By adding a decorator (@profile) to the functions you want to profile, memory profiler can track memory usage at each line of the function, providing detailed insights into memory allocation and deallocation.

```
=== demo encryption ===
key:        0xc615dcca190a36549d9a122bff0fe062 (16 bytes)
nonce:      0xb13bdf3a8a7eb7a5c6cf1e13ef90fd3d (16 bytes)
plaintext:  0x68656c6c6f (5 bytes)
ass.data:   0x686f7720617265207967f75 (11 bytes)
ciphertext: 0xbbc26dbde1 (5 bytes)
tag:        0xb72837c4b3544a99b4682d1974b3d7c8 (16 bytes)
received:   0x68656c6c6f (5 bytes)
Execution time: 3.622211336998589
```

Fig. 2. timeit module runtime calculation in python

```
         1370 function calls in 0.001 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.001    0.001 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 ascon.py:102(ascon_process_ciphertext)
        2    0.000    0.000    0.000    0.000 ascon.py:151(ascon_finalize)
        8    0.000    0.000    0.000    0.000 ascon.py:165(ascon_permutation)
       72    0.000    0.000    0.000    0.000 ascon.py:175(<listcomp>)
        1    0.000    0.000    0.000    0.000 ascon.py:19(ascon_decrypt)
        2    0.000    0.000    0.000    0.000 ascon.py:191(get_random_bytes)
       10    0.000    0.000    0.000    0.000 ascon.py:197(zero_bytes)
       51    0.000    0.000    0.000    0.000 ascon.py:201(to_bytes)
       36    0.000    0.000    0.000    0.000 ascon.py:205(bytes_to_int)
       36    0.000    0.000    0.000    0.000 ascon.py:207(<listcomp>)
        4    0.000    0.000    0.000    0.000 ascon.py:211(bytes_to_state)
        4    0.000    0.000    0.000    0.000 ascon.py:212(<listcomp>)
        6    0.000    0.000    0.000    0.000 ascon.py:215(int_to_bytes)
        6    0.000    0.000    0.000    0.000 ascon.py:216(<listcomp>)
      720    0.000    0.000    0.000    0.000 ascon.py:219(rotr)
        7    0.000    0.000    0.000    0.000 ascon.py:223(bytes_to_hex)
        1    0.000    0.000    0.000    0.000 ascon.py:227(print_text)
        1    0.000    0.000    0.000    0.000 ascon.py:228(<listcomp>)
        1    0.000    0.000    0.001    0.001 ascon.py:240(ascon)
        2    0.000    0.000    0.000    0.000 ascon.py:36(ascon_initialize)
        1    0.000    0.000    0.000    0.000 ascon.py:5(ascon_encrypt)
        2    0.000    0.000    0.000    0.000 ascon.py:52(ascon_process_associated_data)
        1    0.000    0.000    0.000    0.000 ascon.py:69(ascon_process_plaintext)
        1    0.000    0.000    0.001    0.001 {built-in method builtins.exec}
      331    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        1    0.000    0.000    0.000    0.000 {built-in method builtins.max}
        8    0.000    0.000    0.000    0.000 {built-in method builtins.print}
       36    0.000    0.000    0.000    0.000 {built-in method builtins.sum}
        2    0.000    0.000    0.000    0.000 {built-in method posix.urandom}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        7    0.000    0.000    0.000    0.000 {method 'format' of 'str' objects}
        7    0.000    0.000    0.000    0.000 {method 'hex' of 'bytes' objects}
```

Fig. 3. cprofile execution in python

V. Conclusion and Future Directions

Conclusion of the Ascon-128 algorithm

- **Lightweight and Flexible in Hardware.** Current implementation results show that Ascon provides excellent implementation characteristics in terms of size and speed. Due to the small state size and the elegant structure of Ascon's round function, it is additionally possible to provide hardware implementations that are trimmed towards providing either a smaller area or higher speed .
- **Bitsliced in Software.** Ascon is designed to facilitate bitsliced software imple- mentations. The internal permutation is based on very simple operations that are intuitively defined in terms of simple word-wise (64-bit) standard operations. These operations are also well-suited for processors with smaller word sizes, and can take advantage of pipelining and parallelization features of high-end processors.

```
Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
   239     23.8 MiB     23.8 MiB           1   @profile
   240                                         def ascon():
   241     23.8 MiB      0.0 MiB           1       keysize = 16
   242     23.8 MiB      0.0 MiB           1       print("=== demo encryption ===")
   243
   244     23.8 MiB      0.0 MiB           1       key = get_random_bytes(keysize)
   245     23.8 MiB      0.0 MiB           1       nonce = get_random_bytes(16)
   246
   247     23.8 MiB      0.0 MiB           1       associateddata = b"how are you"
   248     23.8 MiB      0.0 MiB           1       plaintext = b"hello"
   249
   250     23.8 MiB      0.0 MiB           1       ciphertext = ascon_encrypt(key, nonce, associateddata, plaintext)
   251     23.8 MiB      0.0 MiB           1       receivedplaintext = ascon_decrypt(key, nonce, associateddata, ciphertext)
   252
   253     23.8 MiB      0.0 MiB           1       if receivedplaintext == None:
   254                                                 print("verification failed!")
   255
   256     23.8 MiB      0.0 MiB           2       print_text(
   257     23.8 MiB      0.0 MiB           1           [
   258     23.8 MiB      0.0 MiB           1               ("key", key),
   259     23.8 MiB      0.0 MiB           1               ("nonce", nonce),
   260     23.8 MiB      0.0 MiB           1               ("plaintext", plaintext),
   261     23.8 MiB      0.0 MiB           1               ("ass.data", associateddata),
   262     23.8 MiB      0.0 MiB           1               ("ciphertext", ciphertext[:-16]),
   263     23.8 MiB      0.0 MiB           1               ("tag", ciphertext[-16:]),
   264     23.8 MiB      0.0 MiB           1               ("received", receivedplaintext),
   265                                                 ]
   266                                             )
```

Fig. 4. memory profiler usage calculation of the algorithm

- **Easy Integration of Side-Channel Countermeasures.** Ascon can be imple- mented efficiently on platforms and applications where side-channel resistance is im- portant. The very efficient bitsliced implementation of the S-boxes prevents cache- timing attacks, since no look-up tables are required.
- **Balanced Design.** While there is arguably a need for ciphers designed for corner cases, such as extremely lightweight designs with very low area footprints, or ciphers designed to provide very high speed on specific software platforms, we follow a more balanced design approach.
- **Online.** The Ascon cipher is online and can encrypt plaintext blocks before subse- quent plaintexts or the plaintext length are known. The same holds for decryption, which decrypts ciphertext blocks online in the order they were computed during en- cryption.
- **Single-Pass.** For both encryption and decryption, just one pass over the data is required.
- **Inverse-Free.** Ascon does not need to implement any inverse operations. In other words, the permu- tations pa and pb are only evaluated in one direction for both en- cryption and decryption, which significantly reduces the area overhead for implemen- tations.
- **High Key Agility.** Ascon neither needs a key schedule, nor expands the key by any other means. Therefore, there are no hidden setup costs when the key is changed.
- **Simplicity.** Ascon is intuitively defined on 64-bit words using only the common bitwise Boolean functions AND, OR, XOR, NOT, and ROT (bitwise rotation). This significantly reduces the effort of implementing the algorithm on new target platforms.
- **Robustness.** Ascon is a nonce-based scheme. As with any authenticated encryption scheme, repeating nonces is a misuse setting, and implies a loss of semantic security. Since Ascon is based on the MonkeyDuplex construction, it inherits many of its properties in misuse settings.

Future Directions and Research in the area of Ascon-128 and Light weight cryptography

- **Improving Side-Channel Resistance:** Ascon is designed to facilitate efficient implementation of side-channel countermeasures such as masking and threshold implementations. Further research could explore developing and analyzing optimized side-channel resistant hardware and software implementations of Ascon.
- **Cryptanalysis:** While presenting some cryptanalysis results up to 4 rounds, further cryptanalysis of the full-round Ascon permutations and the overall authenticated encryption scheme could uncover potential weaknesses or improve the understanding of its security margins.
- **Hardware and Software Optimizations:** Ascon's efficiency in both hardware and software implementations is highlighted. Future research could focus on developing even more optimized and high-performance Ascon implementations, especially for constrained devices or specialized hardware.
- **Integrating Ascon in Larger Systems:** Exploring how Ascon can be integrated and used effectively in various application domains, such as the Internet of Things, could lead to new insights and potential improvements to the design.

- **Proofs and Security Analyses:** Building upon the existing security claims, further theoretical analysis and provable security results for Ascon's mode of operation and its resistance against various attack vectors could strengthen the confidence in its security.
- **Comparisons to Other Authenticated Encryption Schemes:** Conducting in-depth comparisons of Ascon's performance, security, and implementation characteristics against other CAESAR finalists or standardized authenticated encryption schemes could provide valuable insights.
- **Extending the Ascon Family:** Exploring additional parameter sets or variants of the Ascon design, such as supporting different key or rate sizes, could broaden the applicability of the algorithm to a wider range of use cases.

VI. References

1) Megha Agrawal, Donghoon Chang, and Somitra Sanadhya. *sp-AELM: Sponge based authenticated encryption scheme for memory constrained devices*. In Ernest Foo and Douglas Stebila, editors, *Information Security and Privacy – ACISP 2015*, volume 9144 of LNCS, pages 451–468. Springer, 2015.
2) Elena Andreeva, Joan Daemen, Bart Mennink, and Gilles Van Assche. *Security of keyed sponge constructions using a modular proof approach*. In Gregor Leander, editor, *Fast Software Encryption – FSE 2015*, volume 9054 of LNCS, pages 364–384. Springer, 2015.
3) Ralph Ankele and Robin Ankele. *Software benchmarking of the 2nd round CAESAR candidates*. Cryptology ePrint Archive, Report 2016/740, 2016.
4) Daniel J. Bernstein and Tanja Lange (editors). *eBACS: ECRYPT Benchmarking of Cryptographic Systems*. https://bench.cr.yp.to. (accessed 15 September 2016).
5) Guido Bertoni, Joan Daemen, Nicolas Debande, Thanh-Ha Le, Michaël Peeters, and Gilles Van Assche. *Power analysis of hardware implementations protected with secret sharing*. In IEEE/ACM International Symposium on Microarchitecture – MICRO 2012, pages 9–16. IEEE Computer Society, 2012.
6) Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. *Sponge Functions*. ECRYPT Hash Workshop 2007, May 2007.
7) Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. *Duplexing the sponge: Single-pass authenticated encryption and other applications*. In Ali Miri and Serge Vaudenay, editors, *Selected Areas in Cryptography – SAC 2011*, volume 7118 of LNCS, pages 320–337. Springer, 2011.