Report

Lakshay 2021059
Ekansh 2021044
Sahil 2021281

Rocca -

Introduction:

Rocca is a type of **cryptographic cipher**, designed to encrypt plaintext. Ciphers like Rocca are often optimized for high speed and low resource consumption, making them suitable for real-time encryption, especially in resource-constrained environments.The goal of Rocca is to meet the requirement in 6G systems in terms of both performance and security. For performance, Rocca achieves an encryption/decryption speed of more than 100 Gbps in both raw encryption scheme and AEAD scheme.

Overview of Algorithm:

The algorithm uses two 128 bit keys namely k1 and k2 the entire key k is obtained by concatenating k1 with k2 (k => k1 | k2) along with this a 128 bit nonce is also used while initializing the initial state of the system. This algorithm on encryption not only generates a cipher text but also a tag which can be used for authentication.

Components:

1) **Keys:** Two 128 bit keys k1 and k2 are used for encryption

2) **Nonce:** A 128 bit nonce is used to prevent replay attacks.
3) **Associated Data (AD):** This is used for authentication but not encryption can be any metadata of the file.
4) **Plaintext:** Data which we are encrypting
5) **S (State):** Initial state (a vector of eight 32 bytes keys)

Procedure:

Initialisation Phase: We set up the initial state using our keys and nonce and some constants (z0 and z1). The initial phase is the step which governs what will be the output after encryption so that it relies on keys.

```
def rocca_initialization(N, K0, K1):
 global rocca_Z0, rocca_Z1, rocca_S
 rocca_S[0], rocca_S[1], rocca_S[2], rocca_S[3] = K1, N, rocca_Z0,
rocca_Z1
 rocca_S[4], rocca_S[5], rocca_S[6], rocca_S[7] = XOR(N, K1), "0" * 32,
K0, "0" * 32
 for i in range(20): rocca_R(rocca_Z0, rocca_Z1)
 # rocca_S[0] = XOR(rocca_S[0], K0) # Not Applicable for Original Paper
 # rocca_S[4] = XOR(rocca_S[4], K1) # Not Applicable for Original Paper
 return None
```

Padding: Then we pad our message block and associated data with 0s to a fixed size so that they are a multiple of 256 bit size blocks. Also if associated data is provided we process that to have an authentication mechanism.

```
m = ""
for i in AD:
  m += str(bin(int(i, 16))[2:].zfill(4))

if(len(m) > 0):
```

```
  lAD = len(m)
  d = lAD // 256
  _AD = m
  # Padding required
  if lAD%256 != 0:
    # print("[&] Adding Padding to AD")
    _AD = m + ''.join(['0'] * ((d + 1) * 256 - lAD))
  rocca_processAD(_AD)

m_bits = ""
for i in M:
  m_bits += str(bin(int(i, 16))[2:].zfill(4))

if(len(m_bits) > 0):
  lM = len(m_bits)
  d = (lM) // 256
  _M = m_bits

  # Padding required
  if (lM%256) != 0:
    # print("[&] Adding Padding to M")
    _M = m_bits + ''.join(['0'] * ((d + 1) * 256 - lM))
```

Encrypting the Plain Text: Then we encrypt the message (32 byte blocks) with some combination of our current state ensuring that encryption is dependent on the evolving states. Every time a block is encrypted, the state gets updated.

```
  # print("[&] Encrypting")
  C = '#' * len(_M)
  C = rocca_encryption(_M, C)
  C = C[:len(m_bits)]
```

Tag Generation: Finally tag is generated using the final state and encoded length of associated data and message. The tag ensures that the cipher is not tempered.

```
def rocca_finalization(l_AD, l_M):
 global rocca_Z0, rocca_Z1, rocca_S
 for i in range(20): rocca_R(l_AD, l_M)
 T = "0" * 32
 for i in range(8): T = XOR(T, rocca_S[i])
 return T
```

Key Operations:

XOR Operation: This function is used extensively in the algorithm. It takes the bitwise XOR of the two input blocks provided thereby helping us in encryption.

AES Encryption: This function is used to encrypt the plaintext after we have set our state (undergoing 20 rounds). The function not only encrypts the plain text but keeps on updating the state for future blocks which has an avalanche effect.

State Update: The state is updated in each round using a combination of XOR operations and AES transformations preventing linear cryptanalysis.

Tag Finalization: Finally the tag is derived after all the above steps which is used for authentication only.

Functions Implemented:

1) **rocca_encrypt:** Responsible for encryption along with cipher and tag generation
2) **rocca_initialization:** Initializes the state variables (the S vector)
3) **rocca_encryption:** Updates the state variable alongside encrypting the plain text.
4) **rocca_finalization:** Responsible for generating the tag
5) **helper_function:** And many more helper functions responsible for converting forms.

AES S-box and Conclusion

The AES S-box is a key component in encryption, providing non-linearity through byte substitution, which strengthens resistance to cryptanalysis. It uses finite field mathematics to replace each byte with a corresponding value, ensuring security in transformations.

Rocca is a secure cipher combining encryption and authentication for data integrity and confidentiality. It uses AES-based transformations, a unique initialization phase, and XOR operations to protect against replay, tampering, and chosen-plaintext attacks. By integrating encryption and authentication, Rocca offers a robust, efficient solution for modern cryptographic needs.

Time Estimates:

In order to calculate the time it took to encrypt plaintexts of different lengths we imported the time library.

We used multiple messages of different length such as 64, 256, 1024, 8192, 81920 bits and calculated time for each message.

For time calculation we use time.time() function to calculate the current time before and after encryption and simply subtracted to determine the time algorithm took.

```
# Analysis
K0 = "00" * 16
K1 = "00" * 16
N  = "00" * 16
AD = "00" * 32
BYTES = [64, 256, 1024, 8192, 81920]
for i in range(5):
 print(f"----------------------Test Case {i}-------------------------------")
 Bytes = BYTES[i]
 M = "00" * Bytes
 print(f"K0 : {K0}")
 print(f"K1 : {K1}")
 print(f"N : {N}")
 print(f"AD : {AD}")
 print(f"Message Length[in bytes] : {Bytes}")
 print()
 print("Starting")
 start_time = time.time()
 rocca_encrypt(K0, K1, N, AD, M)
 end_time = time.time()
 print("Finished")
 elapsed_time = end_time - start_time
 print("Time Elapsed:", elapsed_time)
 print("Speed: ", Bytes * 8 / (elapsed_time * 1000), "Mbps")
 print()
```

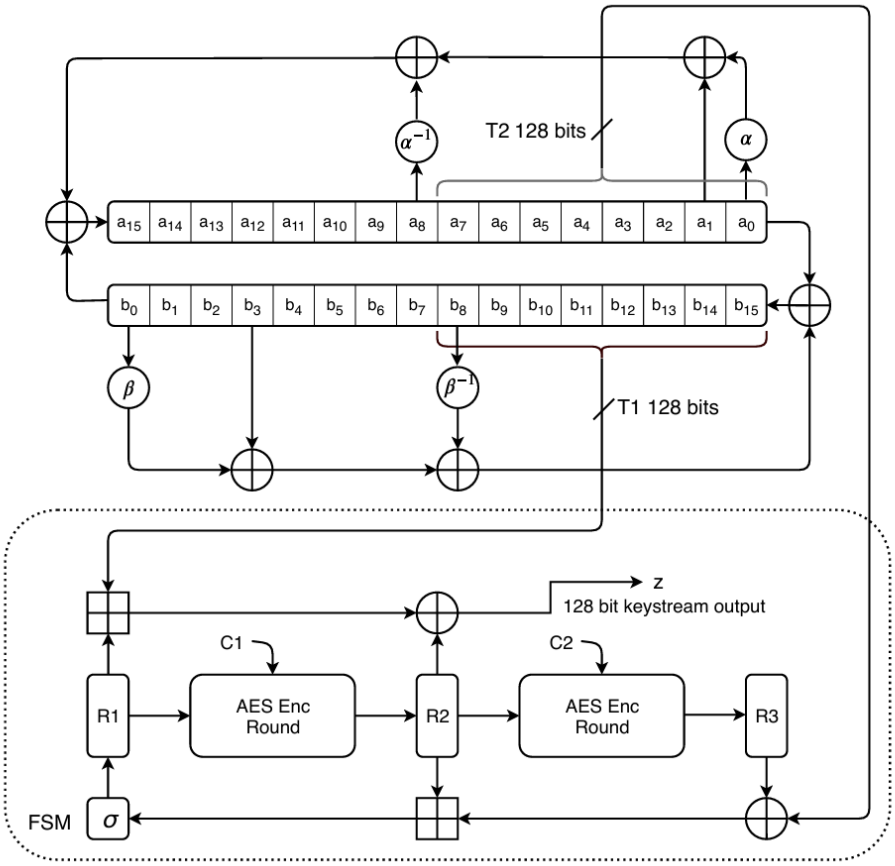
Algorithm	Size of input plain text				
	81920	8192	1024	256	64
Rocca	61.98	65.55	37.81	16.63	4.45

SNOW-V :

Introduction

SNOW-V is an advanced stream cipher developed to deliver high-speed encryption, making it well-suited for virtualized environments, particularly those expected in upcoming 5G mobile communication systems. The design of SNOW-V builds upon the earlier SNOW 3G architecture, incorporating enhancements that address the growing demands of modern processors. These improvements include support for SIMD (Single Instruction, Multiple Data) instructions and the use of the AES encryption standard, ensuring that SNOW-V can meet the performance and security requirements of today's high-throughput applications.

Design Overview



SNOW-V operates using a linear feedback shift register (LFSR) and a finite state machine (FSM), building on the structure of its predecessors but optimized for modern vectorized implementations. The cipher comprises two primary components: LFSR-A and LFSR-B, each consisting of 16 cells, each 16 bits wide. These LFSRs work together to produce high-speed encryption. The FSM includes three 128-bit registers and utilizes two AES encryption rounds to update its state, ensuring security and efficiency in the encryption process.

Initialization and Key Scheduling

The cipher is initialized with a 256-bit key and a 128-bit initialization vector (IV). The initialization process loads the key

and IV into the LFSRs and includes specific operations that mix the key into the cipher's state at the final steps, enhancing security against brute-force attacks by masking the key bits.

```
def keyiv_setup(self, key, iv):

        for i in range(8):

            self.LFSR_A[i] = (((iv[2 * i + 1] << 8) | iv[2 * i]) & divid)

            self.LFSR_A[i + 8] = (((key[2 * i + 1] << 8) | key[2 * i]) &
divid)

            self.LFSR_B[i] = 0x0000

            self.LFSR_B[i + 8] = (((key[2 * i + 17] << 8) | key[2 * i +
16]) & divid)



        # Set fsm registers to 0

        for i in range(4):

            self.R1[i] = self.R2[i] = self.R3[i] = 0x00000000



        # To do the initialisations

        for i in range(16):

            z = self.keystream()

            for j in range(8):

                # XOR it with LFSR_A

                self.LFSR_A[j + 8] ^= (((z[2 * j + 1] << 8) | z[2 * j]) &
divid)



            # As condition given for the round after 14
```

```
            if i == 14:
                for j in range(4):
                    a = ((key[4 * j + 3] << 8) | key[4 * j + 2]) & divid
                    b = ((key[4 * j + 1] << 8) | key[4 * j + 0]) & divid
                    self.R1[j] ^= ((a << 16) | b) & dividend

            # As condition given for the round after 15
            if i == 15:
                for j in range(4):
                    a = ((key[4 * j + 19] << 8) | key[4 * j + 18]) & divid
                    b = ((key[4 * j + 17] << 8) | key[4 * j + 16]) & divid
                    self.R1[j] ^= ((a << 16) | b) & dividend
```

Keystream Generation

Each cycle of SNOW-V produces 128 bits of keystream, aligning with the cipher's design to support high-throughput data transmission. This is particularly aimed at meeting future mobile telecommunication systems' high data rate requirements, which necessitate efficient, high-speed cryptographic solutions.

```
# To generate the key stream of the given length set by the user
    def generate_keystream(self, length):
        keystream = []
        while len(keystream) < length:
            z = self.keystream()
```

```
            keystream.extend(z)

        return keystream[:length]
```

Performance and Applications

SNOW-V is made for high-performance environments, using CPU capabilities, including SIMD instructions, to handle large vectors of integers efficiently, making it well-suited for environments where hardware acceleration is impossible. It can achieve significantly faster operations than traditional AES-256, making it a strong candidate for securing high-speed network traffic in 5G systems.

To check for the performance in terms of encrypting the text, we are checking it for some test cases through the code -

```
# Analysis

K   = "00" * 32

IV  = "00" * 16

BYTES = [64, 256, 1024, 8192, 81920]

# Creating an object of SnowV class

cipher = SnowV()



for i in range(5):

  print(f"------------------Test Case {i}----------------------")

  Bytes = BYTES[i]
```

```
M = "00" * Bytes

print(f"K : {K}")

print(f"IV : {IV}")

print(f"Message Length[in bytes] : {Bytes}")

print()

print("Starting")

start_time = time.time()

ks = cipher.generate_keystream(Bytes)

end_time = time.time()

print("Finished")

elapsed_time = end_time - start_time

print("Time Elapsed:", elapsed_time)

print("Speed: ", Bytes * 8 / (elapsed_time * 1000), "Mbps")

print()
```

Security Analysis

SNOW-V is built on AES-256, mainly focusing on the security that will be equivalent to or better. The algorithm ensures robust security in diverse scenarios and tries to resist attackers. The use of AES rounds in the FSM and the LFSR operations contributes to a high-security level that complies with the needs of current and future encryption standards.

Conclusion

SNOW-V is one of the great steps towards developing the stream ciphers capable of meeting the requirements for 5G encryption and potentially 6G encryption.

Akshat bkl

Algorithm	Size of input plaintexts (bits)				
	81920	8192	1024	256	64
SNOW-V (Mbps)	49.18	54.6	52.71	52.67	35.37