Report on Rocca Encryption Algorithm

Introduction

The Rocca encryption algorithm is a symmetric stream cipher designed for secure data encryption in modern cryptographic applications. Rocca operates based on a combination of the Advanced Encryption Standard (AES) for cryptographic transformations and a robust initialization phase using a nonce and keys. The key feature of Rocca is its ability to efficiently process both the encryption of the message and the authentication of associated data (a key aspect of modern authenticated encryption schemes).

Overview of Rocca Algorithm

The Rocca encryption algorithm utilizes two 128-bit keys (key0 and key1) and a unique 128-bit nonce. The algorithm is designed to encrypt plaintext data while also supporting the authentication of associated data, such as headers or metadata, which are not encrypted but need to be authenticated.

Key Components

1. **Keys**: Two 128-bit keys (key0 and key1) are used to initialize the state and drive the encryption transformations.
2. **Nonce**: A unique 128-bit value that ensures the freshness of each encryption, preventing replay attacks.
3. **Associated Data**: Data that needs to be authenticated but is not encrypted. This could include headers or metadata.
4. **Plaintext**: The message or data that is to be encrypted.
5. **Initial State**: The state used to initialize the encryption process, consisting of a combination of keys, nonce, and fixed constants.

Steps Involved in Rocca Encryption

1. **Initialization Phase**:
   - The initial state is set up using the provided nonce, keys, and two fixed constants (z0 and z1). This phase prepares the encryption state, ensuring that each encryption operation is unique.
   - The state is then transformed over 20 rounds using AES-based updates, mixing the nonce, keys, and constants into the state.
2. **Padding**:
   - The input data (both associated data and plaintext) is padded to a 32-byte block size to ensure proper alignment with the encryption algorithm. The padding is achieved by appending zero bytes to the data until it reaches the required block size.
3. **Processing Associated Data**:

- If associated data is provided, it is processed first, updating the encryption state for each 32-byte block. The associated data is not encrypted, but it is authenticated to ensure its integrity.

4. **Encrypting the Plaintext**:
    - The plaintext is then encrypted in 32-byte blocks. Each block is XORed with a transformed version of the current state, ensuring that the encryption is dependent on the evolving state.
    - After each block is encrypted, the state is updated for the next block.
5. **Authentication Tag Generation**:
    - After encrypting the plaintext, an authentication tag is generated. This tag is derived from the final state of the cipher, the lengths of the associated data and plaintext, and the processed data itself.
    - The tag ensures that both the encrypted data and the associated data have not been altered.

Key Operations

1. **XOR Operation**: The XOR operation is used extensively throughout the algorithm. It combines data blocks at various stages, ensuring that the encryption and state transformation are both secure and unpredictable.
2. **AES Encryption**: AES encryption is applied during the state transformation process. Specifically, it is used to encrypt state blocks, mixing them to ensure that the encryption evolves over time.
3. **State Update**: The state is updated in each round using a combination of XOR operations and AES transformations. This ensures that the encryption process is non-linear and difficult to reverse without the proper keys.
4. **Tag Finalization**: The authentication tag is derived by applying additional rounds of the state update process using the lengths of the associated data and plaintext. The final state is used to compute the tag, ensuring data integrity.

Functions in the Rocca Encryption Implementation

1. `Rocca_encrypt`: This is the main function that performs the encryption. It initializes the state, processes the associated data, encrypts the plaintext, and generates the authentication tag.
2. `pad_to_block_size`: This function ensures that data is padded to a 32-byte block size, as required by the algorithm.
3. `update_round_state`: This function updates the cipher state using AES encryption and XOR operations for each round of the state transformation.
4. `initialize_state`: This function initializes the cipher's state using the nonce, keys, and fixed constants. It prepares the encryption state and applies multiple rounds of transformation.
5. `process_associated_data`: This function handles the processing of associated data, updating the state accordingly to ensure the integrity of this data.

6. **`encrypt_message`**: This function performs the encryption of the plaintext message in blocks, updating the ciphertext and state with each block.
7. **`finalize_state`**: This function is responsible for generating the authentication tag based on the final state and the lengths of the associated data and plaintext.
8. **`encode_length_as_little_endian`**: This function encodes the lengths of the associated data and plaintext in a 32-byte little-endian format, which is required for the tag generation.

AES S-box

The AES S-box is used as part of the state transformations, providing substitution during the AES encryption steps. The substitution is an essential step to introduce non-linearity into the encryption process, making it more secure against cryptanalysis.

Security Considerations

Rocca, like other authenticated encryption schemes, aims to protect against common cryptographic attacks such as replay attacks, tampering, and chosen-plaintext attacks. By combining encryption and authentication into a single process, Rocca ensures the integrity and confidentiality of both the encrypted message and the associated data.

Conclusion

Rocca is a robust and secure stream cipher suitable for applications requiring both encryption and authentication of data. Its use of AES-based transformations and a unique initialization phase ensures strong security guarantees. The algorithm is efficient, using a combination of XOR operations and AES encryption, and is highly adaptable to various cryptographic contexts where both confidentiality and integrity are required.

To compute the time required for encrypting plaintexts of various sizes using the Rocca stream cipher, the following process was carried out:

Experimental Setup:

1. **Plaintext Sizes**: The encryption time was measured for multiple plaintext sizes, specifically 128, 256, 1024, 8192, and 81920 bits.
2. **Key and Nonce Setup**: A 128-bit AES key (AES-128) was used, split into two 16-byte parts. The nonce and associated data were also defined as 128-bit values, represented as hexadecimal strings and converted into integer lists for the encryption process.
3. **State Initialization**: An initial state for the Rocca cipher was set up as a list of eight zero values, which could be modified based on further encryption specifications.

Encryption Process:

For each plaintext size:

1. **Plaintext Generation**: A helper function was used to generate a message filled with zero bytes corresponding to the specified size in bits. The message was then converted into a list of integers to be used as input for the encryption.
2. **Encryption and Timing**:
    - The start time was recorded using `time.perf_counter()`.
    - The Rocca encryption was performed by calling the `rocca_encrypt()` function, which takes the key, nonce, associated data, plaintext message, and the state as inputs, and returns the encrypted ciphertext and tag.
    - The end time was recorded using `time.perf_counter()` again once the encryption process was completed.

Time Calculation:

- The elapsed time for the encryption process was computed by subtracting the start time from the end time.
- The encryption speed in bits per second was calculated by dividing the plaintext size by the elapsed time.
- Finally, the encryption speed was converted from bits per second to megabits per second (Mbps) for easier interpretation.

This methodology provides the encryption speed in Mbps for each plaintext size, allowing for a comparison of how the encryption time scales with the size of the input data.

Report on Snow-V Encryption Algorithm

Introduction

SNOW-V is the latest addition to the SNOW family of stream ciphers, specifically designed to meet the demands of high-speed encryption in virtualized environments, such as those anticipated in future 5G systems. This algorithm builds upon the structure of its predecessors (e.g., SNOW 3G) but introduces significant enhancements to align with modern software architectures, ensuring improved performance on platforms utilizing vectorized processing and AES hardware acceleration.

SNOW-V retains the core design principles of the SNOW family, featuring two key components:

1. **Linear Feedback Shift Registers (LFSRs)**: A dual-register circular structure that ensures high throughput and state mixing.
2. **Finite State Machine (FSM)**: An updated FSM employing 128-bit registers and incorporating the full AES encryption round function for increased complexity and security.

The key advancements in SNOW-V include:

- Adoption of 128-bit registers for FSM components, expanding the state size.
- Optimized LFSRs operating at eight times the speed of the FSM, leveraging new generation polynomials.
- Integration of AES round functions for FSM updates, utilizing hardware acceleration for enhanced software performance.
- Masking key bits during initialization to fortify against initialization-based attacks.

Design Overview

SNOW-V operates through a carefully orchestrated interaction between its LFSR and FSM components. The design prioritizes efficient software implementation while maintaining robust cryptographic strength.

Linear Feedback Shift Registers (LFSRs)

The LFSR component of SNOW-V consists of two 16-cell registers, named **LFSR-A** and **LFSR-B**, with each cell representing a 16-bit value. These registers are governed by distinct polynomials:

- **LFSR-A Polynomial**: gA(x)=x16+x15+x12+x11+x8+x3+x2+x+1
- **LFSR-B Polynomial**: gB(x)=x16+x15+x14+x11+x8+x6+x5+x+1

The states of the LFSRs evolve according to recursive relations involving the taps α\alphaα and β\betaβ, defined as roots of the respective polynomials. At each clock cycle:

- The LFSRs output two 128-bit words, T1 and T2, derived from specific segments of LFSR-B and LFSR-A, respectively.
- The LFSRs are clocked eight times per FSM cycle, ensuring rapid updates to a 512-bit total state.

Finite State Machine (FSM)

The FSM comprises three 128-bit registers (R1, R2 and R3) and operates on the inputs T1 and T2 from the LFSRs. Each FSM cycle produces a 128-bit keystream block z via the following steps:

1. **Keystream Generation**:
   $z(t)=(R1(t) \boxplus_{32} T1(t)) \oplus R2(t)$
   Here, $\boxplus_{32}$ denotes the modular addition of 32-bit words.
2. **Register Updates**:
   - R1 is updated via a byte permutation σ applied to the result of $R2 \boxplus_{32} (R3 \oplus T2)$
   - R2 and R3 undergo transformations using the AES round function, with fixed constants C1 and C2 as inputs.

Initialization

The initialization phase prepares the cipher state using the 256-bit key and 128-bit initialization vector (IV):

1. Key and IV values populate the LFSR-A and LFSR-B registers.
2. Over 16 initialization steps, the cipher state is repeatedly updated, with keystream output XORed into the LFSRs to ensure diffusion.
3. At the final two steps, the key is XORed into R1 to finalize the initialization process.

Key Features

1. **Performance**:
   - Designed to leverage SIMD instructions and AES hardware acceleration for exceptional speed.
   - Outputs 128-bit keystream blocks per cycle, significantly outperforming predecessors like SNOW 3G.
2. **Security**:
   - Offers 256-bit cryptographic strength, aligning with the requirements for future-proofing against quantum and advanced computational attacks.
   - Incorporates safeguards against initialization-based and correlation attacks.

Conclusion

SNOW-V represents a critical evolution in stream cipher design, balancing speed and security for next-generation systems. Its innovative architecture ensures compatibility with modern processors while maintaining a high level of resistance to cryptanalytic attacks. As 5G deployments expand, SNOW-V's role in providing fast, secure communication in virtualized environments becomes indispensable.

Performance Evaluation

Algorithm	Size of input plaintext(bits)				
	128	256	1024	8192	81920
Rocca(Mbps)	0.055	0.113	0.402	1.76	3.25
Snow-v	0.074	0.123	0.454	1.006	2.98

For smaller input sizes (e.g., 256 bits), the performance difference may not be as significant. However, as input sizes increase (e.g., 1024 bits or 8192 bits), Rocca’s simpler structure allows it to outperform Snow V in terms of speed.**Rocca** tends to scale better with larger inputs, while **Snow V** experiences a more noticeable performance drop due to its complex operations.