Rijndael Cipher Stream – RCS 1.0

Revision 1.0f, October 11, 2024

John G. Underhill – john.underhill@protonmail.com

This document is an engineering level description of the RCS encrypted and authenticated stream cipher. In its contents, a guide to implementing RCS, an explanation of its design, as well as references to its component primitives and supporting documentation.

1. Introduction and Background

The Rijndael Cipher Stream (RCS) is a modern, authenticated stream cipher designed to provide robust security against both classical and quantum computational threats. It builds upon the principles of Rijndael-256, which extends the original Rijndael (AES) cipher to a 256-bit block size. RCS integrates the Rijndael transformations with advanced cryptographic components like cSHAKE for key expansion and either the QMAC or KMAC MAC functions for authentication to offer a strong AEAD (Authenticated Encryption with Associated Data) framework.

RCS was developed in response to the increasing demand for cryptographic primitives that can withstand attacks from quantum computers, which threaten the security of traditional ciphers. By enhancing the key schedule with cSHAKE, employing a large state block size, and adding additional transformation rounds, RCS achieves a high level of resistance to differential and linear cryptanalysis, as well as quantum-based threats such as Grover's algorithm.

Objectives of RCS include:

- Providing secure, high-performance authenticated encryption.
- Ensuring compatibility with existing AES-NI hardware optimizations.
- Resisting both classical and post-quantum cryptanalytic attacks.
- Offering flexibility in key size (256-bit and 512-bit) and number of rounds (22 rounds used in RCS-256, and 30 rounds used in RCS-512).

RCS is particularly suited for applications in secure communications, data protection, and environments that require long-term confidentiality guarantees.

2. Mathematical Foundations

RCS employs the Rijndael substitution-permutation network (SPN) as its core rounds transformation function. The mathematical operations of RCS extend those of AES to operate with a 256-bit block size. The following describes the key transformations used within each round of RCS:

2.1 State Representation

The state in RCS is represented as a 4 × 8 matrix of bytes (32 bytes total), where each byte is treated as an element of the finite field GF(2^8). The matrix operations in the cipher leverage this representation to achieve high diffusion and resistance to differential attacks.

2.2 Core Functions

The main transformations applied to the state matrix in RCS are:

- **SubBytes**: This is a non-linear substitution step where each byte in the state matrix is replaced with its corresponding value from the Rijndael S-Box. The S-Box is designed to resist linear and differential cryptanalysis.
- **ShiftRows**: This operation involves cyclically shifting the rows of the state matrix to the left by a certain offset. This step introduces diffusion by ensuring that bytes are moved to different positions.
- **MixColumns**: This is a linear mixing operation applied to each column of the state matrix. It treats each column as a four-term polynomial over GF(2^8) and then multiplies it by a fixed polynomial modulo x4+1x^4 + 1x4+1.
- **AddRoundKey**: In this step, the state matrix is XORed with a round key derived from the key expansion process. This operation adds confusion to the cipher by combining the state with the key material.

The RCS cipher increases the number of mixing rounds compared to AES to improve its resistance against cryptanalytic attacks:

- **RCS-256**: Uses 22 rounds of transformations.
- **RCS-512**: Uses 30 rounds of transformations.

3. Key Expansion and Scheduling

The key schedule in RCS uses the Keccak function cSHAKE, a variant of the SHA-3 family, which serves as a cryptographic eXtendable Output Function (XOF). This key expansion mechanism is specifically designed to overcome the weaknesses of traditional AES key schedules by generating highly diffused round keys that resist differential attacks.

3.1 cSHAKE-Based Expansion

- **Initialization**: The cSHAKE function absorbs the input key and an optional custom string to generate a pseudo-random sequence of bytes. This sequence is used to populate the round key array.

- **Round Key Generation**: The round keys are derived iteratively from the cSHAKE output, ensuring that each round key is cryptographically independent from the others, which significantly increases the difficulty of key recovery attacks.
- **Variants**: RCS supports both 256-bit and 512-bit key sizes, with the number of rounds set to 22 for RCS-256 and 30 for RCS-512.

The use of cSHAKE ensures that the key expansion process is resistant to side-channel attacks and related-key attacks, making RCS suitable for high-security environments.

4. Operational Details

4.1 Encryption and Decryption

AAD

Additional data must be added after the cipher has been initialized, and before the message is encrypted or decrypted. The additional data is added to the internal MAC state.

Encryption

The nonce is added to the keyed MAC function.

A message of length n is passed to the transform, the CTR mode encrypts the message by encrypting the nonce, performing a bitwise XOR on the message using the encrypted nonce as a pseudo-random keystream. After each block of the message (32 bytes ot 256 bits) is encrypted, the nonce acting as a monotonic counter is incremented, ensuring that the next block of encrypted output is fully differentiated.

The ciphertext is added to the MAC state, and the encoding size is also added (initial-value + aad-size + nonce-size + message-length). The hash is then finalized, generating an output hash (message tag) which is appended to the ciphertext.

Decryption

Decryption follows the reverse order; first the cipher is initialized, the additional data is added to the keyed MAC function. The nonce is added, then the ciphertext. The MAC is finalized and the hash is compared to the tag appended to the ciphertext. If the MAC calculation matches the tag appended to the ciphertext, the message authenticity is verified, and the ciphertext is passed to the CTR function. The CTR wraps the keyed encryption function and nonce, and is run on the ciphertext decrypting the plaintext message.

4.2 Encryption Process

The encryption process of RCS follows a sequence of iterative rounds, each consisting of the core Rijndael transformations:

1. **Initial Round**: The state matrix undergoes the AddRoundKey transformation using the initial round key.
2. **Main Rounds**: The state is then processed through a series of rounds, each involving the SubBytes, ShiftRows, MixColumns, and AddRoundKey transformations.
3. **Final Round**: The last round excludes the MixColumns step, as its primary purpose is to maintain high efficiency while still ensuring sufficient diffusion in the ciphertext.

4.3 Decryption Process

The decryption process in RCS reverses the transformations applied during encryption. It follows the same sequence of operations but in the reverse order.

4.4 Counter Mode Operation

RCS operates in a counter mode (CTR), where the block cipher is used to generate a pseudo-random stream by treating the nonce as a monotonic counter incremented after each block is transformed, encrypting the counter to create a key-stream. This key-stream is XORed with the plaintext to produce the ciphertext. The use of CTR mode allows RCS to function efficiently as a stream cipher with robust security properties.

5. Mathematical Description

Where:

- ad is the additional data
- c is the output ciphertext
- i is the optional info tweak
- k is the input cipher key
- l is the encoded length
- m is the input message
- mk is the MAC key
- n is the nonce
- rk is the cipher key
- tag is the MAC tag
- Comp a timing neutral array comparison
- CTR the counter mode function
- E is the encryption function
- M is KMAC or QMAC
- S is customized SHAKE

Initialization; initialize cSHAKE with the key, the cipher name string or optional info tweak. The Keccak Squeeze function is called twice; once to create the round keys, the second time to create the MAC key.

$$rk, mk = \mathrm{S}(k \parallel i)$$

Additional Data; Additional data can be added to the MAC after initialization and before the message is encrypted or decrypted. This data is mixed in with the MAC state, allowing authentication of related metadata.

$$M_{state} = \mathrm{M}_{mk}(ad)$$

Encryption; the keyed encryption function wrapped in a CTR mode encrypts the nonce, then increments it, and XORs the pseudo-random output stream with the message to create the ciphertext.

$$c = \mathrm{CTR}(\mathrm{E}_{rk}, n, m)$$

The MAC function adds the nonce, the ciphertext, and the encoding length to the MAC, then calculates the MAC tag.

$$tag = \mathrm{M}_{mk}(n \parallel c \parallel l)$$

The MAC tag is appended to the ciphertext.

$$c = c \parallel tag$$

Decryption; MAC the nonce, ciphertext, and encoding length, and compare it to the tag appended to the ciphertext.

$$tmp = \mathrm{M}_{mk}(n \parallel c \parallel l)$$

If the hashes match, decrypt the message by running the CTR on the ciphertext.

```
if Comp($tmp$, $tag$) are equal
    $m = \mathrm{CTR}(\mathrm{E}_{rk}, n, c)$
else
    return $False$
```

6. Cryptographic Analysis

6.1 Resistance to Classical Attacks

RCS is designed to resist traditional cryptanalytic techniques such as:

- **Linear Cryptanalysis**: The high number of rounds in RCS ensures that finding linear approximations of the cipher's behavior is computationally infeasible.
- **Differential Cryptanalysis**: The use of a large block size (256 bits) and a robust key schedule significantly reduces the chances of differential characteristics propagating through the rounds.

6.2 Quantum Resistance

- **Grover's Algorithm**: The increased key sizes in RCS (up to 512 bits) effectively doubles the complexity of attacks based on Grover's algorithm, making it impractical to perform brute-force searches on quantum computers.
- **Quantum Algebraic Attacks**: The use of non-linear S-Boxes and the complexity of the MixColumns transformation add layers of security against algebraic techniques that might exploit quantum computational power.

6.3 Side-Channel Attack Mitigations

RCS implements countermeasures against side-channel attacks such as Differential Power Analysis (DPA) and Timing Attacks by utilizing a timing neutral key schedule (Keccak) and performing constant-time operations wherever possible. The RCS implementation uses intrinsic instructions AVX/AVX2/AVX-512 when available, and AES-NI CPU embedded AES instructions in the transform function.

7. Authentication Mechanism

RCS employs an integrated authentication mechanism using either the KMAC (Keccak Message Authentication Code) or QMAC MAC functions. This mechanism ensures the integrity and authenticity of the data, making RCS a robust choice for authenticated encryption with associated data (AEAD).

7.1 KMAC Functionality

KMAC is a keyed cryptographic hash function based on the Keccak sponge construction. It is designed to provide message authentication while maintaining the cryptographic strength of SHA-3. RCS leverages either the KMAC-R24 (standard rounds) and KMAC-R12 (reduced rounds) or QMAC MAC functions depending on the authentication security requirements and performance constraints.

KMAC Operation in RCS:

1. **Keyed Initialization**: During the setup phase, KMAC is initialized with a portion of the key material derived from the cSHAKE key expansion. This ensures that the generated MAC is tied directly to the specific key used in the encryption process.

2. **Absorbing Phase**: The input message (ciphertext) and any associated data are absorbed into the KMAC function in blocks.
3. **Squeezing Phase**: The MAC is generated by squeezing the sponge construction to produce a fixed-length output, which is then appended to the ciphertext.

7.2 QMAC Operation in RCS:

The QMAC MAC function combines Keccak (SHAKE XOF function) to generate hash and finalization keys (H and F) with 256-bit carryless multiplication and Galois field reduction using GD(2^256). Based on the GMAC construction, QMAC is a high performance MAC function with post-quantum security and resilience.

7.3 Authentication Code Generation

- **For RCS-256**: The QMAC and KMAC produces a 256-bit MAC code to ensure the integrity and authenticity of the ciphertext.
- **For RCS-512**: The KMAC generates a 512-bit MAC code, providing an even higher level of security for critical data. QMAC generates a 256-bit MAC code.

The generated MAC is appended to the ciphertext during encryption. During decryption, the integrity of the ciphertext is verified by recomputing the MAC on the ciphertext and comparing it with the received value. If the MAC values do not match, the ciphertext is considered to have been tampered with, and the function is aborted before decryption begins.

8. Security Considerations

8.1 Best Practices for Key Management

RCS requires careful management of cryptographic keys to maintain its security properties. The following best practices are recommended:

- **Unique Key and Nonce Pairs**: Each encryption operation must use a unique key-nonce combination to prevent key reuse and ensure security against replay attacks.
- **Key Storage**: Keys should be stored securely using hardware security modules (HSMs) or other secure key management techniques.
- **Regular Key Rotation**: Implement a policy of regular key rotation to reduce the risk of key exposure and limit the amount of data encrypted with a single key.

8.2 Nonce Handling and Recommendations

Nonces play a critical role in the security of the Rijndael Cipher Stream. To avoid nonce reuse and the potential compromise of encryption security:

- **Random Nonce Generation**: Nonces should be generated using a cryptographically secure pseudo random number generator (CSPRNG).
- **Nonce Length**: The nonce in RCS is 256 bits, providing a large enough space to ensure uniqueness even with a high number of encryption operations.
- **Incremental Nonces**: If a CSPRNG is not available, nonces can be generated in an incremental manner to guarantee they are never repeated.

8.3 Side-Channel Attack Mitigations

RCS includes built-in countermeasures against side-channel attacks:

- **Timing Attack Resistance**: All critical operations are performed in constant time to prevent attackers from inferring secret information based on processing time variations.
- **DPA Resistance**: The use of AES-NI hardware acceleration reduces the risk of differential power analysis (DPA) attacks by minimizing the exposure of intermediate state information.

8.4 Potential Vulnerabilities and Mitigation Strategies

RCS, like all cryptographic systems, has potential vulnerabilities that must be addressed:

- **Brute-Force Attacks**: The large key sizes (256-bit and 512-bit) in RCS make brute-force attacks computationally infeasible, even with quantum computing advancements.
- **Quantum Attacks**: RCS's reliance on cSHAKE and its increased number of rounds provide a strong defense against quantum algorithms, such as those based on Grover's search.

9. Comparison with Other Stream Ciphers

RCS is designed to offer significant advantages over other stream ciphers, particularly in terms of security and performance. Here is a comparative analysis with other widely-used stream ciphers.

9.1 Comparison with RC4

- **Security**: RC4 is known for several vulnerabilities, including biases in the keystream. RCS overcomes these issues with a robust design and a large key size.
- **Performance**: While RC4 is lightweight and fast, its security flaws make it unsuitable for modern applications. RCS provides enhanced security while maintaining efficient performance through hardware acceleration.

9.2 Comparison with Salsa20/ChaCha20

- **Key Size Flexibility**: Unlike Salsa20 and ChaCha20, which primarily use 128-bit or 256-bit keys, RCS supports both 256-bit and 512-bit keys, providing options for varying levels of security.

- **Authentication**: RCS integrates either KMAC or QMAC authentication, while ChaCha20 typically relies on the Poly1305 MAC. KMAC and QMAC offer a robust alternative for scenarios requiring stringent security guarantees.

9.3 Unique Advantages of RCS

- **Post-Quantum Security**: The use of Rijndael-256 and cSHAKE-based key scheduling makes RCS more resilient against quantum attacks than most traditional stream ciphers.

- **AES-NI Integration**: RCS takes full advantage of hardware-accelerated AES instructions, significantly boosting its encryption speed on compatible CPUs.

- **AEAD Mode**: RCS has an additional data function, that can add data to the MAC function. This allows for the integrity of related data (e.g. packet headers) to be authenticated by the MAC function.

9.4 Integration of Advanced Cryptographic Components

- **Rijndael-256 Foundation**: RCS builds on the proven security of the Rijndael cipher (the basis of AES) but uses a 256-bit block size. This provides a larger state space, improving its resistance to cryptanalytic attacks.

- **cSHAKE Key Expansion**: By replacing the traditional AES key schedule with cSHAKE, RCS enhances its security against differential and related-key attacks. The use of a cryptographic XOF (extendable output function) like cSHAKE ensures that round keys are cryptographically independent, which is a significant improvement over the original Rijndael key schedule.

- **KMAC for Authentication**: The adoption of KMAC (Keccak-based MAC) for message authentication is a strong design choice. KMAC leverages the security of the SHA-3 family, providing robust integrity and authenticity checks without requiring additional cryptographic primitives.

- **QMAC for Authentication**: The adoption of QMAC (Quantum safe MAC) for message authentication is also a strong design choice. QMAC leverages the proven security of Galois field mathematics in a post-quantum secure GMAC like construction, for maximum performance and security.

9.5 Quantum Resistance:

- **Larger Key Sizes**: With support for both 256-bit and 512-bit keys, RCS significantly increases its resistance to attacks from quantum algorithms like Grover's search. Grover's algorithm can effectively halve the key length, so the larger keys in RCS help maintain a strong security margin.

- **Increased Rounds**: The decision to use 22 rounds for RCS-256 and 30 rounds for RCS-512 further strengthens the cipher's resilience against quantum and classical attacks. The increased number of rounds complicates the analysis of the cipher's internal state, making it more resistant to cryptanalytic techniques.

9.6 Hardware Optimization and Performance:

- **AES-NI and AVX Support**: RCS is designed to take full advantage of modern CPU instructions like AES-NI and AVX, AVX2, and AVX-512 instruction sets, which can significantly speed up encryption and decryption operations. This makes RCS suitable for high-performance applications where throughput is critical.
- **Counter Mode (CTR) Operation**: Using CTR mode for encryption allows RCS to function as a true stream cipher with parallelizable encryption processes, enhancing its efficiency in both software and hardware implementations.

9.7 Flexibility and Configurability:

- **Tweakable Cipher**: The inclusion of a customizable parameter (the "info" field in the key parameters) allows for domain separation and adaptation to specific use cases. This flexibility makes RCS versatile and well-suited for various cryptographic applications.
- **Authentication Configurability**: RCS supports the full-round KMAC-R24 and a reduced-round KMAC-R12, allowing users to trade off between performance and security based on their specific needs. RCS also supports the high-performance QMAC construction, for maximum performance and post-quantum security.

10. Implementation Guidance

This section provides detailed implementation guidelines to help developers integrate RCS into cryptographic systems effectively.

10.1 Code Examples

Example implementations of RCS encryption and decryption using the provided functions:

RCS-256 Encryption

```
static void rcs256_encrypt_example(uint8_t* enc, const uint8_t* msg, size_t msglen, const uint8_t* key,
const uint8_t* nonce, const uint8_t* data, size_t datalen)
{
	qsc_rcs_state ctx;

	/* load the key structure */
	qsc_rcs_keyparams kp = { key, QSC_RCS256_KEY_SIZE, nonce, NULL, 0 };

	/* initialize the cipher state for encryption */
	qsc_rcs_initialize(&ctx, &kp, true);
	/* add the associated data to the mac */
	qsc_rcs_set_associated(&ctx, data, datalen);
	/* encrypt and mac the message */
	qsc_rcs_transform(&ctx, enc, msg, msglen);
	/* dispose of the cipher state */
	qsc_rcs_dispose(&ctx);
}
```

RCS-256 Decryption

```
static bool rcs256_decrypt_example(uint8_t* msg, const uint8_t* enc, size_t enclen, const uint8_t* key,
const uint8_t* nonce, const uint8_t* data, size_t datalen)
{
	qsc_rcs_state ctx;
	size_t mlen;
	bool res;

	/* load the key structure */
	qsc_rcs_keyparams kp = { key, QSC_RCS256_KEY_SIZE, nonce, NULL, 0 };

	/* initialize the cipher state for decryption */
	qsc_rcs_initialize(&ctx, &kp, false);
	/* add the associated data to the mac */
	qsc_rcs_set_associated(&ctx, data, datalen);
	/* mac and decrypt the message */
	mlen = enclen - QSC_RCS256_MAC_SIZE;
	res = qsc_rcs_transform(&ctx, msg, enc, mlen);
	/* dispose of the cipher state */
	qsc_rcs_dispose(&ctx);

	return res;
}
```

10.2 Hardware Acceleration

RCS supports AES-NI and AVX, AVX2, and AVX-512 instructions to enhance performance. When enabled, these hardware features significantly reduce the time required to perform the encryption and decryption operations.

- **Enabling AES-NI**: Developers can enable AES-NI by defining the QSC_SYSTEM_AESNI_ENABLED constant in their build configuration.
- **AVX Optimization**: The AVX instruction set allows for parallel processing. The CTR functions are vectorized, which along with AES-NI instructions can process multiple blocks in parallel for high performance throughput.

10.3 Known Answer Tests

Message=000102030405060708090A0B0C0D0E0F000102030405060708090A0B0C0D0E0F

Key=000102030405060708090A0B0C0D0E0F000102030405060708090A0B0C0D0E0F

Nonce=FFFEFDFCFBFAF9F8F7F6F5F4F3F2F1F0DFDEDDDCDBDAD9D8D7D6D5D4D3D2D1D0

RCS-256-K24:

Output=7940917E9219A31248946F71647B15421535941574F84F79F6110C1F2F776D03

F38582F301390A6B8807C75914CE0CF410051D73CAE97D1D295CB0420146E179

RCS-256-K12:

Output=7940917E9219A31248946F71647B15421535941574F84F79F6110C1F2F776D03

225B05B1FB100A4D9208522BACB1AEBEE62A94D19BFF53B41ACE75D031926707

RCS-256

Output= 7940917E9219A31248946F71647B15421535941574F84F79F6110C1F2F776D03

11. Use Cases and Applications

RCS is well-suited for a variety of secure communication and data protection applications, including:

11.1 Secure Messaging

RCS can be employed in secure messaging platforms where confidentiality and message integrity are crucial. Its high resistance to cryptanalysis makes it ideal for protecting sensitive communications.

11.2 Data Encryption for Storage

With its strong authentication features and large key sizes, RCS is a robust choice for encrypting data at rest. It ensures that stored data remains secure against unauthorized access, even if attackers have access to quantum computing capabilities.

11.3 Network Security

RCS's fast encryption and decryption, aided by hardware acceleration, make it suitable for use in high-throughput network protocols that require secure data transmission, such as VPNs and encrypted web traffic.

12. Appendices

12.1 References

- "The Rijndael Block Cipher: AES Proposal" by Joan Daemen and Vincent Rijmen.

- "Post-Quantum Cryptography: Current State and Future Directions" by Michele Mosca.
- NIST SP 800-185: "SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash, and ParallelHash."