Cipher Stream eXtended – CSX 1.0

Revision 1.0c, October 12, 2024

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

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

1. Introduction

The CSX (Cipher Stream eXtended) is an authenticated stream cipher, designed for post-quantum secure communication. It incorporates a 512-bit key and uses either the Keccak MAC function (KMAC) or the post-quantum QMAC for message authentication. The cipher is optimized for both performance and security, utilizing advanced SIMD instructions and providing a flexible and tweakable initialization mechanism.

1.1 Purpose

The purpose of this specification is to define the CSX cipher in detail, describing its encryption and authentication mechanisms, and its use of the cSHAKE XOF function for input key expansion. This document also provides a security analysis of the cipher and outlines its potential applications in high-security environments such as post-quantum cryptography.

1.2 Scope

This specification covers the technical aspects of the CSX cipher, including its encryption process, authentication mechanism, key expansion, and nonce management. It also provides implementation details, including how to initialize and transform the state, the structure of the cipher, and the cryptographic functions used in its construction.

2. Cryptographic Overview

2.1 CSX Structure

The CSX cipher is a stream cipher with authenticated encryption and associated data (AEAD) functionality. The encryption rounds function was inspired by the ChaCha stream cipher, while the authentication mechanism relies on either the Keccak KMAC (Keccak Message Authentication Code) or QMAC a post-quantum MAC function. The core design of the cipher incorporates the following elements:

- **Key Size**: 512 bits.
- **Nonce Size**: 128 bits (16 bytes).

- **Block Size**: 1024 bits (128 bytes).
- **Rounds**: 40 rounds of permutation.
- **Message Authentication**: KMAC-512 or a reduced round variant KMAC-R12 for faster authentication, or QMAC for maximum performance.

CSX is a stream cipher that uses a MAC function, with authenticated encryption and additional data (AEAD). The encryption uses a counter mode mechanism, where the state, nonce, and constant values are permuted and used to generate a key-stream, which is XORed with the plaintext to produce the ciphertext. The authentication process generates a MAC that is appended to the ciphertext for message integrity protection.

2.2 ChaCha-like Stream Cipher Design

CSX uses a rounds-based mixing function similar to the ChaCha stream cipher rounds function for encryption. It operates on 64-bit words, and the internal state is arranged as a 4x4 matrix of 64-bit integers. The cipher expands the key into an internal state that mixes the key, nonce, and constant values. CSX also supports vectorized implementations using AVX2 and AVX-512 instruction sets, allowing for efficient processing of multiple blocks simultaneously.

2.3 KMAC Authentication

The authentication process is built around the KMAC-512 function, a variant of the Keccak function tailored for keyed hashing. KMAC produces an authentication tag, ensuring message integrity and authenticity. In the standard mode, CSX uses KMAC-512 with 24 rounds. A reduced round variant, KMAC-R12, is also available for environments where authentication performance is prioritized over security, reducing the number of rounds in the Keccak permutation to 12 rounds. An implementation that uses just the stream cipher with no authentication is also configurable.

2.4 QMAC Authentication

The QMAC MAC function combines Keccak (SHAKE XOF function) to generate hash and finalization keys (H and F) with 256-bit carry-less 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.

3. Mathematical Description

3.1 Internal State

The CSX cipher internal state is represented as a 4x4 matrix of 64-bit unsigned integers, similar to ChaCha's matrix but extended to operate with 64-bit integers and 1024-bit blocks. The state matrix is initialized with the following components:

- **Key (K)**: The 512-bit secret key initialization parameter.

- **Nonce (N)**: The 128-bit nonce initialization parameter.
- **Info (C)**: The information string initialization parameter.

State Initialization

The state is initialized as follows:

- **Rows 0-1**: The 512-bit key is split into eight 64-bit integers, which populate the first two rows of the matrix.
- **Rows 2-3**: The constant string takes up the third row, and the nonce is split into two 64-bit integers, which populate the start of the fourth row, followed by the remainder of the constant string.

The state matrix is initialized as:

K_0	K_1	K_2	K_3
K_4	K_5	K_6	K_7
C_0	C_1	C_2	C_3
N_0	N_1	C_4	C_5

Where $K_0, K_1, \ldots, K_7$ are parts of the key, N_0 and N_1 are parts of the nonce, and $C_0, C_1, C_2, C_3, C_4, C_5$ are the constant string.

3.2 ChaCha-like Permutation

CSX performs 40 rounds of a permutation operation that follows a similar structure to the ChaCha permutation. Each round consists of a series of quarter-round operations that update the state matrix. A quarter-round operation involves four elements of the matrix and performs the following steps:

1. **Addition**: Two elements are added together modulo 2^{64}.
2. **XOR**: The result is XORed with another element.
3. **Rotation**: The result is bitwise rotated by a predefined constant.

The quarter-round operation is applied to both diagonal and non-diagonal elements of the matrix to ensure a strong diffusion of input data across the state.

The rotational constants.

38	19	10	55	33	4	51	13
16	34	56	51	4	53	42	41
34	41	59	17	23	31	37	20
31	44	47	46	12	47	44	30

3.3 Key-stream Generation

After 40 rounds of mixing, the internal state is transformed into a 1024-bit key-stream block. This key-stream is XORed with the plaintext to produce the ciphertext. The 128-bit counter is incremented after each block to ensure unique key-stream blocks for every block of data.

3.4 KMAC-based Authentication

The message authentication is handled by KMAC-512. The authentication process operates as follows:

1. **Keyed Initialization**: KMAC is initialized with the user input key and optional tweak and expanded using the cSHAKE function.
2. **Message Input**: The ciphertext, nonce, encoding length, and associated data (if any) are processed by KMAC to generate the authentication tag.
3. **Tag Generation**: The finalized authentication tag is generated and appended to the ciphertext.

The authentication process ensures the integrity and authenticity of both the ciphertext and associated data.

3.5 QMAC-based Authentication

The message authentication is done with QMAC-512, which uses SHAKE-512 to generate the hash and finalization keys, to create a 512-bit secure authentication.

1. **Keyed Initialization**: QMAC is initialized with the user input key and optional tweak and expanded using the cSHAKE function.
2. **Message Input**: The ciphertext, nonce, encoding length, and associated data (if any) are processed by QMAC to generate the authentication tag.

3. **Tag Generation**: The finalized authentication tag is generated and appended to the ciphertext.

4. Encryption and Decryption Process

4.1 Encryption Process

The encryption process in the CSX cipher follows an encrypt-then-MAC scheme, ensuring both confidentiality and authenticity of the data. The key aspects of the encryption process are described below.

4.1.1 Key Expansion

CSX uses the cSHAKE-512 function for key expansion. The 512-bit input key is processed by cSHAKE to generate both the cipher key and the authentication key. This key expansion step ensures a uniform and secure derivation of keys used in both the encryption and the MAC stages. The expanded key is divided into two parts:

- **Cipher Key**: Used in the permutation for generating the key-stream.
- **MAC Key**: Used in the KMAC-512 or QMAC operation for message authentication.

This is done with two separate calls to the Keccak Squeeze function, distancing the XOF relational output between the MAC and cipher keys by a permutation call.

4.1.2 Initialization

The state matrix is initialized using the key, nonce, and info string. The 512-bit key is split into 64-bit integers, which are loaded into the first two rows of the state matrix. The first 256 bits of the constant string populate the third row, the nonce fills the first two slots of the fourth row, with the remainder of the info string populating the last two 64-bit integers of the fourth row.

4.1.3 Key-stream Generation

The key-stream is generated by applying the CSX permutation on the internal state. This permutation is repeated for 40 rounds, mixing the elements of the state matrix using addition, bitwise XOR, and rotation operations. The resulting 1024-bit output is the key-stream block, which is XORed with the plaintext to produce the ciphertext.

The key-stream generation is performed for each block of data, with the counter incremented after each block is permuted to ensure that every key-stream block is unique.

4.1.4 Encryption Steps

1. **Key Expansion**: The input key is expanded using cSHAKE-512 to produce the cipher and MAC keys.

2. **State Initialization**: The internal state is initialized with the cipher key, nonce, and constant string.
3. **Key-stream Generation**: The rounds permutation function is applied to generate the key-stream.
4. **XOR with Plaintext**: The key-stream is bitwise XORed with the plaintext to produce the ciphertext.
5. **Authentication Tag Generation**: QMAC or KMAC-512 processes the ciphertext, nonce, associated data, and the encoding length, to generate the authentication tag.
6. **Append Authentication Tag**: The authentication tag is appended to the ciphertext.

4.2 Decryption Process

The decryption process steps in CSX follow the inverse of the encryption steps. The ciphertext is first authenticated, and if the authentication succeeds, the plaintext is recovered by XOR of the key-stream with the ciphertext. The key steps in the decryption process are described below.

4.2.1 Authentication Check

Before decryption, the ciphertext is authenticated using QMAC or KMAC-512. The ciphertext, associated data, nonce, and encoding length (including AEAD data length) are processed by QMAC-512 or KMAC-512, and the generated hash is compared with the hash tag appended to the ciphertext. If the tags match, the ciphertext is deemed authentic, and decryption can proceed. If the tags do not match, the decryption process is aborted and the transform function returns *false*.

4.2.2 Key-stream Generation and Decryption

Once the ciphertext is authenticated, the same key-stream generation process used in encryption is applied. The state is initialized with the same key, nonce, and constant string values, and the CSX permutation generates the key-stream. The key-stream is then XORed with the ciphertext to recover the plaintext.

4.2.3 Decryption Steps

1. **Key Expansion**: The input key is expanded using cSHAKE-512 to produce the cipher and MAC keys.
2. **State Initialization**: The internal state is initialized with the key, nonce, and constant string.
3. **Authentication Check**: The ciphertext, nonce, associated data, and encoding length, are processed using QMAC-512 or KMAC-512, and the output hash is compared to the authentication tag for equality.

4. **Key-stream Generation**: The permutation rounds function is applied to generate the key-stream.
5. **XOR with Ciphertext**: The key-stream is XORed with the ciphertext to recover the plaintext.

4.3 Performance Considerations

CSX benefits from vectorized Keccak implementations, including AVX2 and AVX512 optimizations, which allow it to process multiple blocks in parallel. These optimizations significantly improve performance on modern processors that support these instruction sets. CSX processes blocks of 1024 bits (128 bytes) at a time, and its parallel processing capabilities make it highly efficient for high-throughput encryption.

5. Mathematical Description

Where:

- k is the input cipher key
- n is the nonce
- i is the info string
- c is the output ciphertext
- ad is the additional data
- m is the input message
- tag is the MAC tag
- mk is the MAC key
- rk is the cipher key
- S is customized SHAKE
- M is QMAC or KMAC
- E is the encryption function

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

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

The key, nonce, and info are added to the cipher state.

$$C_{s\ tate} = \{\ k, n, c\ \}$$

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 data, like a serialized packet header.

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

Encryption; the cipher generates the key-stream, and XORs the key-stream with the message to produce the ciphertext.

$$c = \mathrm{E}_{ck}(m)$$

The MAC function adds the ciphertext, nonce, and MAC input length (including AAD), to the MAC state, then calculates the MAC tag.

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

The MAC tag is appended to the ciphertext.

$$c = c \parallel tag$$

Decryption; MAC the ciphertext, nonce, and MAC input length (including AAD), and compare it to the tag appended to the ciphertext.

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

If the hashes match, decrypt the ciphertext.

if Compare(*tmp*, *tag*) are equal
    $m = \mathrm{E}_{ck}(c)$
else
    return *false*

6. Implementation Guidance

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

6.1 Code Examples

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

Encryption

```
void csx_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_csx_state ctx;

    /* load the key structure */
```

```
	qsc_csx_keyparams kp = { key, QSC_CSX_KEY_SIZE, nonce, NULL, 0 };

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

Decryption

```
bool csx_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_csx_state ctx;
	size_t mlen;
	bool res;

	/* load the key structure */
	qsc_csx_keyparams kp = { key, QSC_CSX_KEY_SIZE, nonce, NULL, 0 };

	/* initialize the cipher state for decryption */
	qsc_csx_initialize(&ctx, &kp, false);
	/* add the associated data to the mac */
	qsc_csx_set_associated(&ctx, data, datalen);
	/* mac and decrypt the message */
	mlen = enclen - QSC_CSX_MAC_SIZE;
	res = qsc_csx_transform(&ctx, msg, enc, mlen);
	/* dispose of the cipher state */
	qsc_csx_dispose(&ctx);

	return res;
}
```

7. Security Considerations

The security of the CSX cipher is based on well-established cryptographic primitives, including the design of the ChaCha stream cipher, the cSHAKE-512 function, and the QMAC-512 or KMAC-512 message authentication code. Each of these components provides a high level of cryptographic strength, making CSX a suitable choice for applications that require authenticated encryption.

7.1 Security of ChaCha Stream Cipher

ChaCha is a stream cipher designed by Daniel J. Bernstein, which has been extensively analyzed and widely adopted for its security and performance. The implementation variant used in CSX (CSX-512) operates with 40 rounds, which is considered more than sufficient to resist future cryptanalytic attacks, including those using quantum computers.

ChaCha Strengths:

- **High Performance**: ChaCha is highly optimized for performance on both software and hardware, making it suitable for a wide range of platforms.
- **Resilience Against Cryptanalysis**: ChaCha has been subjected to extensive cryptanalysis and has resisted all known attacks when configured with 20 or more rounds.
- **No Known Weaknesses**: No practical attacks have been identified against ChaCha when used with a sufficient number of rounds, ensuring its robustness for both current and future use.

7.2 Security of cSHAKE and KMAC

The cSHAKE and KMAC constructions are based on the Keccak sponge function, which forms the core of the SHA-3 standard. These constructions provide strong cryptographic guarantees, including collision resistance, pre-image resistance, and pseudorandom output.

7.2.1 cSHAKE as a Key Derivation Function (KDF)

cSHAKE is used in CSX to expand the input key into multiple keys for encryption and MAC generation. Its security is based on the same principles as SHA-3, providing resistance against brute-force attacks, key collisions, and other key-recovery attacks.

7.2.2 KMAC Security:

- **Message Authentication**: KMAC ensures the authenticity and integrity of both ciphertext and associated data. By using the KMAC-512 function, CSX provides a strong authentication mechanism that is strongly resistant to forgery attacks.
- **Security Bounds**: The 512-bit security level of KMAC ensures that it is resistant to all practical forgery and collision attacks.

7.2.3 QMAC Security:

- **Message Authentication**: QMAC ensures the authenticity and integrity of both ciphertext and associated data. By using the QMAC-512 function, CSX provides a strong authentication mechanism that is strongly resistant to forgery attacks.
- **Security Bounds**: The 512-bit security level of QMAC ensures that it is resistant to all practical forgery and collision attacks.

7.3 Resistance to Side-Channel Attacks

CSX is designed to minimize its susceptibility to side-channel attacks, such as timing attacks and power analysis. This is achieved through the following strategies:

- **Constant-Time Operations**: The core operations of CSX are designed to execute in constant time, preventing attackers from gaining insights into the key or internal state through timing variations.
- **Minimal Data Leakage**: Sensitive information, such as key material and intermediate states, is securely erased from memory as soon as it is no longer needed, reducing the risk of leakage through memory analysis.

7.4 Protection Against Quantum Attacks

As cryptographic research progresses, quantum computers pose a serious threat to many classical cryptographic algorithms. While symmetric key ciphers like ChaCha are less vulnerable to quantum attacks than asymmetric algorithms, CSX incorporates forward-thinking design choices to mitigate quantum risks:

- **Increased Key Size**: The 512-bit key size of CSX provides a higher security margin, making it more resistant to attacks from quantum computers compared to smaller key sizes, thus restoring the security margins to a modern symmetric cipher.
- **Post-Quantum Security Awareness**: By relying on Keccak-based primitives (cSHAKE and KMAC), CSX inherits the post-quantum resistance characteristics of these constructions, which are considered secure against quantum cryptanalysis.

7.5 Authentication and Integrity Guarantees

The authenticated encryption mechanism used by CSX ensures both confidentiality and integrity of the encrypted data. The following guarantees are provided:

- **Confidentiality**: The CSX stream cipher ensures that the plaintext remains confidential, even in the presence of an attacker with access to the ciphertext.
- **Integrity**: The KMAC-512 function provides strong integrity guarantees, ensuring that any modifications to the ciphertext or associated data are detected and rejected during decryption.
- **Authenticity**: Only someone with access to the correct key can produce a valid ciphertext and MAC, ensuring the authenticity of the sender.