KMAC and cSHAKE AEAD Specification

Stephan Müller

2023-12-22

Abstract

This document specifies the cryptographic algorithms of KMAC AEAD and cSHAKE AEAD. A reference implementation of the algorithms is given with leancrypto.

Contents

1 KMAC AEAD Algorithm

This specification defines a symmetric stream cipher algorithm using the authenticated encryption with additional data (AEAD) approach. This algorithm can be used to encrypt and decrypt arbitrary user data. The cipher algorithm uses the KMAC algorithm to generate a key stream which is XORed with either the plaintext (encryption) or ciphertext (decryption) data. The KMAC is initialized with the user-provided key and the user-provided IV. In addition, a second KMAC instance is initialized which calculates a keyed-message digest of the ciphertext to create a message authentication tag. This message authentication tag is used during decryption to verify the integrity of the ciphertext.

1.1 Introduction

This specification defines a symmetric stream cipher algorithm using the authenticated encryption with additional data (AEAD) approach. This algorithm can be used to encrypt and decrypt arbitrary user data.

The base of the algorithm is the generation of a key stream using KMAC which is XORed with the plaintext for the encryption operation, or with the ciphertext for the decryption operation.

The algorithm also applies an Encrypt-Then-MAC by calculating a message authentication tag using KMAC over the ciphertext. During decryption, this calculated message authentication tag is compared with the message authentication tag obtained during the encryption operation. If both values show a mismatch, the authentication fails and the decryption operation is terminated. Only when both message authentication tags are identical the decryption operation completes successfully and returns the decrypted message.

The key along with the IV are used to initialize the KMAC algorithm for generating the key stream. The first output block from the KMAC is used to initialize the authenticating KMAC instance used to calculate the message authentication tag.

The size of the key is defined to be 256 bits when using KMAC-256. The size of the IV can be selected by the caller. The algorithm supports any IV size, including having no IV.

As part of the authentication, the algorithm allows the addition of additional authenticated data (AAD) of arbitrary size. This AAD is inserted into the authentication KMAC instance during calculating the message authentication tag.

1.2 KMAC-based AEAD Cipher Algorithm

1.2.1 Notation

The KMAC-hash denotes the KMACXOF256 function [SP800-185]. The KMAC-hash has 4 arguments: the key `K`, the main input bit string `X`, the requested output length `L` in bits, and an optional customization bit string `S`.

The inputs to the KMAC-hash function are specified with references to these parameters.

1.2.2 Common Processing of Data

```
KMAC-Crypt(key, IV, input data) -> output data, auth key
```

Inputs:

- key: The caller-provided key of size 256 bits
- IV: The caller-provided initialization vector. The IV can have any size including an empty bit-string. See section KMAC AEAD Cryptographic Aspects for a discussion of the IV, however.
- input data: The caller-provided input data - in case of encryption, the caller provides the plaintext data, in case of decryption, caller provides the ciphertext data.

Outputs:

- output data: The resulting data - in case of encryption, the ciphertext is produced, in case of decryption, the plaintext is returned.
- auth key: The key that is used for the KMAC operation calculating the message authentication tag.

The common processing of data is performed as follows:

```
input length = size of input data in bits

KS = KMAC(K = key,
          X = "",
          L = 256 bits + input length,
          S = IV)

auth key = 256 left-most bits of KS

KS crypt = all right-most bits of KS starting with the 256th bit

output data = input data XOR KS crypt
```

1.2.3 Calculating of Message Authentication Tag

```
KMAC-Auth(auth key, AAD, ciphertext, taglen) -> tag
```

Input:

- auth key: The key that is used for the KMAC operation calculating the message authentication tag.
- AAD: The caller-provided additional authenticated data. The AAD can have any size including an empty bit-string.

- ciphertext: The ciphertext obtained from the encryption operation or provided to the decryption operation.
- taglen: The length of the message authentication tag to be generated.

Output:

- tag: The message authentication tag that can be exchanged with the recipient over insecure channels.

The calculation of the message authentication tag is performed as follows:

```
tag = KMAC(K = auth key,
           X = ciphertext || AAD,
           L = taglen,
           S = "")
```

Note, the implementation must ensure that after generating the message authentication tag, the KMAC state is re-initialized with the authentication key to allow multiple successive encryption or decryption operations. Each ciphertext used with an encryption / decryption operation must be be processed with the given KMAC invocation. This is required as otherwise the re-invocation of the encryption / decryption operation causes the KMAC state to be used in an undefined way, i.e. invoking the Keccak absorb function after the Keccak squeeze operation was invoked.

1.2.4 Encryption Operation

`KMAC-Encrypt(key, IV, plaintext, AAD, taglen) -> ciphertext, tag`

Input:

- key: The caller-provided key of size 256 bits
- IV: The caller-provided initialization vector. The IV can have any size including an empty bit-string.
- plaintext: The caller-provided plaintext data.
- AAD: The caller-provided additional authenticated data.
- taglen: The length of the message authentication tag to be generated.

Output:

- ciphertext: The ciphertext that can exchanged with the recipient over insecure channels.
- tag: The message authentication tag that can be exchanged with the recipient over insecure channels.

The encryption operation is performed as follows:

```
ciphertext, auth key = KMAC-Crypt(key, IV, plaintext)
```

```
tag = KMAC-Auth(auth key, AAD, ciphertext, taglen)
```

1.2.5 Decryption Operation

```
KMAC-Decrypt(key, IV, ciphertext, AAD, tag) ->
    plaintext, authentication result
```

Input:

- key: The caller-provided key of size 256 bits
- IV: The caller-provided initialization vector. The IV can have any size including an empty bit-string.
- ciphertext: The ciphertext that was received from the send over insecure channels.
- AAD: The caller-provided additional authenticated data.
- tag: The message authentication tag that was received from the send over insecure channels.

Output:

- plaintext: The plaintext of the data.
- authentication result: A boolean indicator specifying whether the authentication was successful. If it was unsuccessful the caller shall reject the ciphertext.

The decryption operation is performed as follows:

```
plaintext, auth key = KMAC-Crypt(key, IV, ciphertext)

taglen = size of tag

new_tag = KMAC-Auth(auth key, AAD, ciphertext, taglen)

if (new_tag == tag)
  authentication result = success
else
  authentication result = failure
```

If the authentication result indicates a failure, the result of the decryption operation SHALL be discarded.

1.3 KMAC AEAD Cryptographic Aspects

The KMAC AEAD algorithm is a stream cipher which uses the XOR-construction method to perform encryption and decryption. This method is susceptible to attacks when the key stream is identical between different encryption operations. This this case, the key stream can be trivially removed and thus a decryption of the data is possible as follows:

```
ciphertext 1 = plaintext 1 XOR KS

ciphertext 2 = plaintext 2 XOR KS

ciphertext 1 XOR ciphertext 2 =
    (plaintext 1 XOR KS) XOR (plaintext 2 XOR KS) =
    plaintext 1 XOR plaintext 2
```

Thus, the security of the KMAC algorithm is based on the property that the key stream KS is unique for different encryption operations. The key stream is derived from the key and the IV using KMAC. In common use cases, the key may not be able to be modified. Yet, the IV can be modified. Common protocols allow the generation of a new IV during encryption and transmit the IV to the decryptor. Thus, the IV can be used as a diversifier to for the different encryption operations to obtain a different key stream.

As the KMAC algorithm's IV size is unspecified in size, the KMAC algorithm can handle any size that may be pre-defined by the use case or protocol consuming the KMAC AEAD algorithm.

Considering the avalanche effect of the underlying KECCAK algorithm, even a small IV may result in a completely different keystream rendering the aforementioned attack impossible.

The IV is not required to be a confidentially-protected value. It can be communicated in plaintext to the decryptor. This is due to the fact that the IV is used together with the key to generate the key stream using KMAC. An attacker is not able to construct either the key or the key stream by only possessing the IV. Furthermore, the key is defined to possess a cryptographic meaningful entropy (see section 2.3) which implies that the IV does not need to deliver additional entropy to ensure the strength of the KMAC AEAD algorithm.

It is permissible that the IV is generated either by a random number generator or using a deterministic construction method. The only requirement is that the probability in generating a key / IV collision is insignificantly low. This implies that considering the IV is only a diversifier for the key stream, and the fact that the IV is not required to be private, the random number generator is not required to possess a cryptographic meaningful strength.

The selection of KMAC for generating the keystream is based on the statement in [SP800-185] declaring Keccak is usable as a pseudorandom function.

The approach of Encrypt-Then-MAC is selected based on the analysis of [AUTHENC] table 3 considering on the finding that the MAC algorithm of KMAC is strongly unforgeable.

1.4 Normative References

[AUTHENC] Mihir Bellare and Chanathip Namprempre, Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm

[SP800-185] John Kelsey, Shu-jen Chang, Ray Perlne, NIST Special Publication 800-185 SHA-3 Derived Functions: cSHAKE, CSHAKE, TupleHash and ParallelHash, December 2016

2 cSHAKE AEAD Algorithm

This specification defines a symmetric stream cipher algorithm using the authenticated encryption with additional data (AEAD) approach. This algorithm can be used to encrypt and decrypt arbitrary user data. The cipher algorithm uses the cSHAKE algorithm to generate a key stream which is XORed with either the plaintext (encryption) or ciphertext (decryption) data. The cSHAKE is initialized with the user-provided key and the user-provided IV. In addition, a second cSHAKE instance is initialized which calculates a keyed-message digest of the ciphertext to create a message authentication tag. This message authentication tag is used during decryption to verify the integrity of the ciphertext.

2.1 Introduction

This specification defines a symmetric stream cipher algorithm using the authenticated encryption with additional data (AEAD) approach. This algorithm can be used to encrypt and decrypt arbitrary user data.

The base of the algorithm is the generation of a key stream using cSHAKE which is XORed with the plaintext for the encryption operation, or with the ciphertext for the decryption operation.

The algorithm also applies an Encrypt-Then-MAC by calculating a message authentication tag using cSHAKE over the ciphertext. During decryption, this calculated message authentication tag is compared with the message authentication tag obtained during the encryption operation. If both values show a mismatch, the authentication fails and the decryption operation is terminated. Only when both message authentication tags are identical the decryption operation completes successfully and returns the decrypted message.

The key along with the IV are used to initialize the cSHAKE algorithm for generating the key stream. The first output block from the cSHAKE is used to initialize the authenticating cSHAKE instance used to calculate the message authentication tag.

The size of the key is defined to be 256 bits when using cSHAKE-256. The size of the IV can be selected by the caller. The algorithm supports any IV size, including having no IV.

As part of the authentication, the algorithm allows the addition of additional authenticated data (AAD) of arbitrary size. This AAD is inserted into the authentication cSHAKE instance during calculating the message authentication tag.

2.2 cSHAKE-based AEAD Algorithm

2.2.1 Notation

The cSHAKE-hash denotes the cSHAKE256 function [SP800-185]. The cSHAKE-hash has 4 arguments: the main input bit string `X`, the requested output length `L` in bits, a function-name bit string, and an optional customization bit string `S`.

The inputs to the cSHAKE-hash function are specified with references to these parameters.

2.2.2 Common Processing of Data

```
cSHAKE-Crypt(key, IV, input data) -> output data, auth key
```

Inputs:

- key: The caller-provided key of size 256 bits
- IV: The caller-provided initialization vector. The IV can have any size including an empty bit-string. See cSHAKE AEAD Cryptographic Aspects for a discussion of the IV, however.
- input data: The caller-provided input data - in case of encryption, the caller provides the plaintext data, in case of decryption, caller provides the ciphertext data.

Outputs:

- output data: The resulting data - in case of encryption, the ciphertext is produced, in case of decryption, the plaintext is returned.
- auth key: The key that is used for the cSHAKE operation calculating the message authentication tag.

The common processing of data is performed as follows:

```
input length = size of input data in bits

KS = cSHAKE(N = "cSHAKE-AEAD crypt",
            X = IV,
            L = 256 bits + input length,
            S = key)

auth key = 256 left-most bits of KS

KS crypt = all right-most bits of KS starting with the 256th bit

output data = input data XOR KS crypt
```

2.2.3 Calculating of Message Authentication Tag

```
cSHAKE-Auth(auth key, AAD, ciphertext, taglen) -> tag
```

Input:

- auth key: The key that is used for the cSHAKE operation calculating the message authentication tag.
- AAD: The caller-provided additional authenticated data. The AAD can have any size including an empty bit-string.
- ciphertext: The ciphertext obtained from the encryption operation or provided to the decryption operation.
- taglen: The length of the message authentication tag to be generated.

Output:

- tag: The message authentication tag that can be exchanged with the recipient over insecure channels.

The calculation of the message authentication tag is performed as follows:

```
tag = cSHAKE(N = "cSHAKE-AEAD auth",
             X = ciphertext || AAD,
             L = taglen,
             S = auth key)
```

Note, the implementation must ensure that after generating the message authentication tag, the cSHAKE state is re-initialized with the authentication key to allow multiple successive encryption or decryption operations. Each ciphertext used with an encryption / decryption operation must be be processed with the given cSHAKE invocation. This is required as otherwise the re-invocation of the encryption / decryption operation causes the cSHAKE state to be used in an undefined way, i.e. invoking the Keccak absorb function after the Keccak squeeze operation was invoked.

2.2.4 Encryption Operation

```
cSHAKE-Encrypt(key, IV, plaintext, AAD, taglen) -> ciphertext, tag
```

Input:

- key: The caller-provided key of size 256 bits
- IV: The caller-provided initialization vector. The IV can have any size including an empty bit-string.
- plaintext: The caller-provided plaintext data.
- AAD: The caller-provided additional authenticated data.
- taglen: The length of the message authentication tag to be generated.

Output:

- ciphertext: The ciphertext that can exchanged with the recipient over insecure channels.
- tag: The message authentication tag that can be exchanged with the recipient over insecure channels.

The encryption operation is performed as follows:

```
ciphertext, auth key = cSHAKE-Crypt(key, IV, plaintext)

tag = cSHAKE-Auth(auth key, AAD, ciphertext, taglen)
```

2.2.5 Decryption Operation

```
cSHAKE-Decrypt(key, IV, ciphertext, AAD, tag) ->
    plaintext, authentication result
```

Input:

- key: The caller-provided key of size 256 bits
- IV: The caller-provided initialization vector. The IV can have any size including an empty bit-string.
- ciphertext: The ciphertext that was received from the send over insecure channels.
- AAD: The caller-provided additional authenticated data.
- tag: The message authentication tag that was received from the send over insecure channels.

Output:

- plaintext: The plaintext of the data.
- authentication result: A boolean indicator specifying whether the authentication was successful. If it was unsuccessful the caller shall reject the ciphertext.

The decryption operation is performed as follows:

```
plaintext, auth key = cSHAKE-Crypt(key, IV, ciphertext)

taglen = size of tag

new_tag = cSHAKE-Auth(auth key, AAD, ciphertext, taglen)

if (new_tag == tag)
  authentication result = success
else
  authentication result = failure
```

If the authentication result indicates a failure, the result of the decryption operation SHALL be discarded.

2.3 cSHAKE AEAD Cryptographic Aspects

The cSHAKE AEAD algorithm is a stream cipher which uses the XOR-construction method to perform encryption and decryption. This method is susceptible to attacks when the key stream is identical between different encryption operations. This this case, the key stream can be trivially remove and thus a decryption of the data is possible as follows:

```
ciphertext 1 = plaintext 1 XOR KS

ciphertext 2 = plaintext 2 XOR KS

ciphertext 1 XOR ciphertext 2 =
    (plaintext 1 XOR KS) XOR (plaintext 2 XOR KS) =
    plaintext 1 XOR plaintext 2
```

Thus, the security of the cSHAKE algorithm is based on the property that the key stream KS is unique for different encryption operations. The key stream is derived from the key and the IV using cSHAKE. In common use cases, the key may not be able to be modified. Yet, the IV can be modified. Common protocols allow the generation of a new IV during encryption and transmit the IV to the decryptor. Thus, the IV can be used as a diversifier to for the different encryption operations to obtain a different key stream.

As the cSHAKE algorithm's IV size is unspecified in size, the cSHAKE algorithm can handle any size that may be pre-defined by the use case or protocol consuming the cSHAKE AEAD algorithm.

Considering the avalanche effect of the underlying KECCAK algorithm, even a small IV may result in a completely different key stream rendering the aforementioned attack impossible.

The IV is not required to be a confidentially-protected value. It can be communicated in plaintext to the decryptor. This is due to the fact that the IV is used together with the key to generate the key stream using cSHAKE. An attacker is not able to construct either the key or the key stream by only possessing the IV. Furthermore, the key is defined to possess a cryptographic meaningful entropy (see section 2.3) which implies that the IV does not need to deliver additional entropy to ensure the strength of the cSHAKE AEAD algorithm.

It is permissible that the IV is generated either by a random number generator or using a deterministic construction method. The only requirement is that the probability in generating a key / IV collision is insignificantly low. This implies that considering the IV is only a diversifier for the key stream, and the fact that the IV is not required to be private, the random number generator is not required to possess a cryptographic meaningful strength.

The selection of cSHAKE for generating the keystream is based on the statement in [SP800-185] declaring Keccak is usable as a pseudorandom function.

The approach of Encrypt-Then-MAC is selected based on the analysis of [AUTHENC] table 3 considering on the finding that the MAC algorithm of cSHAKE is strongly unforgeable.

2.4 Comparison with KMAC-based AEAD Algorithm

The cSHAKE cipher is completely identical to the KMAC cipher with the exception that the cSHAKE cipher uses cSHAKE256 and the KMAC cipher uses KMACXOF256 as central functions. The difference of the cSHAKE customization string applied by KMAC compared to cSHAKE is irrelevant to the cryptographic strength of both.

The handling of the key is also very similar:

- The cSHAKE cipher sets the key as part of the N input - the N and X input are concatenated and padded by cSHAKE to bring the entire string into multiples of a cSHAKE block. This data is inserted into the SHAKE algorithm which implies that the insertion triggers as many KECCAK operations as cSHAKE blocks are present based on the input. The cSHAKE DRNG data implies that only one cSHAKE block is present and thus one KECCAK operation is performed.
- The KMAC cipher sets the key compliant to the KMAC definition. KMAC sets two well-defined strings as part of the cSHAKE initialization. The cSHAKE initialization concatenates and pads the input strings to bring the entire string into multiples of a cSHAKE block. This data is inserted into the SHAKE algorithm which implies that the insertion triggers as many KECCAK operations as cSHAKE blocks are present on the input. The KMAC cipher key data implies that only one cSHAKE block is present and thus one KECCAK operation is performed. In addition, KMAC pads the key data into a string that is also multiples of a cSHAKE block in size. Again, this data is inserted into the SHAKE algorithm which again triggers as many KECCAK operations as cSHAKE blocks are present with the key-based input. The KMAC-based AEAD cipher algorithm specification implies again, that only one KECCAK operation is performed.

The rationale shows that for both, the cSHAKE cipher and the KMAC cipher, the key, is inserted into the SHAKE state. The additional data inserted with the KMAC operation does not contain any entropy and only mixes the SHAKE state further without affecting the existing entropy provided with the key or diminish the information inserted with the IV. Therefore, with respect to the security strength, the cSHAKE cipher and the KMAC cipher are considered equal.

Considering that the cSHAKE cipher requires only one KECCAK operation during initialization whereas the KMAC cipher requires two operations, the cSHAKE cipher requires less KECCAK operations for processing the same amount of data.

Thus, the cSHAKE cipher has a higher performance with a equal entropy management comparing to the KMAC cipher.

2.5 Normative References

[AUTHENC] Mihir Bellare and Chanathip Namprempre, Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm

[SP800-185] John Kelsey, Shu-jen Chang, Ray Perlne, NIST Special Publication 800-185 SHA-3 Derived Functions: cSHAKE, CSHAKE, TupleHash and ParallelHash, December 2016