

Crypto Forum
Internet-Draft
Intended status: Informational
Expires: 25 August 2025

B. Viguier
ABN AMRO Bank
D. Wong, Ed.
zkSecurity
G. Van Assche, Ed.
STMicroelectronics
Q. Dang, Ed.
NIST
J. Daemen, Ed.
Radboud University
21 February 2025

KangarooTwelve and TurboSHAKE
draft-irtf-cfrg-kangarootwelve-17

Abstract

This document defines four eXtendable Output Functions (XOF), hash functions with output of arbitrary length, named TurboSHAKE128, TurboSHAKE256, KT128 and KT256.

All four functions provide efficient and secure hashing primitives, and the last two are able to exploit the parallelism of the implementation in a scalable way.

This document is a product of the Crypto Forum Research Group. It builds up on the definitions of the permutations and of the sponge construction in [FIPS 202], and is meant to serve as a stable reference and an implementation guide.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 25 August 2025.

Copyright Notice

Copyright (c) 2025 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	3
1.1. Conventions	4
2. TurboSHAKE	5
2.1. Interface	5
2.2. Specifications	6
3. KangarooTwelve: Tree hashing over TurboSHAKE	8
3.1. Interface	8
3.2. Specification of KT128	8
3.3. length_encode(x)	11
3.4. Specification of KT256	11
4. Message authentication codes	11
5. Test vectors	12
6. IANA Considerations	20
7. Security Considerations	21
8. References	22
8.1. Normative References	22
8.2. Informative References	23
Appendix A. Pseudocode	24
A.1. Keccak-p[1600,n_r=12]	24
A.2. TurboSHAKE128	25
A.3. TurboSHAKE256	26
A.4. KT128	27
A.5. KT256	28
Authors' Addresses	29

1. Introduction

This document defines the TurboSHAKE128, TurboSHAKE256 [TURBOSHAKE], KT128 and KT256 [KT] eXtendable Output Functions (XOF), i.e., a hash function generalization that can return an output of arbitrary length. Both TurboSHAKE128 and TurboSHAKE256 are based on a Keccak-p permutation specified in [FIPS202] and have a higher speed than the SHA-3 and SHAKE functions.

TurboSHAKE is a sponge function family that makes use of Keccak-p[n_r=12,b=1600], a round-reduced version of the permutation used in SHA-3. Similarly to the SHAKE's, it proposes two security strengths: 128 bits for TurboSHAKE128 and 256 bits for TurboSHAKE256. Halving the number of rounds compared to the original SHAKE functions makes TurboSHAKE roughly two times faster.

KangarooTwelve applies tree hashing on top of TurboSHAKE and comprises two functions, KT128 and KT256. Note that [KT] only defined KT128 under the name KangarooTwelve. KT256 is defined in this document.

The SHA-3 and SHAKE functions process data in a serial manner and are strongly limited in exploiting available parallelism in modern CPU architectures. Similar to ParallelHash [SP800-185], KangarooTwelve splits the input message into fragments. It then applies TurboSHAKE on each of them separately before applying TurboSHAKE again on the combination of the first fragment and the digests. More precisely, KT128 uses TurboSHAKE128 and KT256 uses TurboSHAKE256. They make use of Sakura coding for ensuring soundness of the tree hashing mode [SAKURA]. The use of TurboSHAKE in KangarooTwelve makes it faster than ParallelHash.

The security of TurboSHAKE128, TurboSHAKE256, KT128 and KT256 builds on the public scrutiny that Keccak has received since its publication [KECCAK_CRYPTANALYSIS][TURBOSHAKE].

With respect to [FIPS202] and [SP800-185] functions, TurboSHAKE128, TurboSHAKE256, KT128 and KT256 feature the following advantages:

- * Unlike SHA3-224, SHA3-256, SHA3-384, SHA3-512, the TurboSHAKE and KangarooTwelve functions have an extendable output.
- * Unlike any [FIPS202] defined function, similarly to functions defined in [SP800-185], KT128 and KT256 allow the use of a customization string.
- * Unlike any [FIPS202] and [SP800-185] functions but ParallelHash, KT128 and KT256 exploit available parallelism.

- * Unlike ParallelHash, KT128 and KT256 do not have overhead when processing short messages.
- * The permutation in the TurboSHAKE functions has half the number of rounds compared to the one in the SHA-3 and SHAKE functions, making them faster than any function defined in [FIPS202]. The KangarooTwelve functions immediately benefit from the same speedup, improving over [FIPS202] and [SP800-185].

With respect to SHA-256 and SHA-512 and other [FIPS180] functions, TurboSHAKE128, TurboSHAKE256, KT128 and KT256 feature the following advantages:

- * Unlike [FIPS180] functions, the TurboSHAKE and KangarooTwelve functions have an extendable output.
- * The TurboSHAKE functions produce output at the same rate as they process input, whereas SHA-256 and SHA-512, when used in a mask generation function (MGF) construction, produce output half as fast as they process input.
- * Unlike the SHA-256 and SHA-512 functions, TurboSHAKE128, TurboSHAKE256, KT128 and KT256 do not suffer from the length extension weakness.
- * Unlike any [FIPS180] functions, TurboSHAKE128, TurboSHAKE256, KT128 and KT256 use a round function with algebraic degree 2, which makes them more suitable to masking techniques for protections against side-channel attacks.

This document represents the consensus of the Crypto Forum Research Group (CFRG) in the IRTF. It is not an IETF product and is not a standard.

1.1. Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

The following notations are used throughout the document:

`'...'` denotes a string of bytes given in hexadecimal. For example, `'0B 80'`.

`|s|` denotes the length of a byte string `'s'`. For example, `('|FF FF|`

= 2.

`'00'^b` denotes a byte string consisting of the concatenation of `b` bytes `'00'`. For example, `'00'^7` = `'00 00 00 00 00 00 00'`.

`'00'^0` denotes the empty byte-string.

`a||b` denotes the concatenation of two strings `a` and `b`. For example, `'10' || 'F1'` = `'10 F1'`

`s[n:m]` denotes the selection of bytes from `n` (inclusive) to `m` (exclusive) of a string `s`. The indexing of a byte-string starts at 0. For example, for `s = 'A5 C6 D7'`, `s[0:1]` = `'A5'` and `s[1:3]` = `'C6 D7'`.

`s[n:]` denotes the selection of bytes from `n` to the end of a string `s`. For example, for `s = 'A5 C6 D7'`, `s[0:]` = `'A5 C6 D7'` and `s[2:]` = `'D7'`.

In the following, `x` and `y` are byte strings of equal length:

`x^=y` denotes `x` takes the value `x XOR y`.

`x & y` denotes `x AND y`.

In the following, `x` and `y` are integers:

`x+=y` denotes `x` takes the value `x + y`.

`x-=y` denotes `x` takes the value `x - y`.

`x**y` denotes the exponentiation of `x` by `y`.

`x mod y` denotes the remainder of the division of `x` by `y`.

`x / y` denotes the integer dividend of the division of `x` by `y`.

2. TurboSHAKE

2.1. Interface

TurboSHAKE is a family of eXtendable Output Functions (XOF). Internally, it makes use of the sponge construction, parameterized by two integers, the rate and the capacity, that sum to the permutation width (here, 1600 bits). The rate gives the number of bits processed or produced per call to the permutation, whereas the capacity determines the security level, see [FIPS202] for more details. This document focuses on only two instances, namely, TurboSHAKE128 and

TurboSHAKE256. (Note that the original definition includes a wider range of instances parameterized by their capacity [TURBOSHAKE].)

An instance of TurboSHAKE takes as input parameters a byte-string *M*, an OPTIONAL byte *D* and a positive integer *L* where

M byte-string, is the Message and

D byte in the range ['01', '02', .. , '7F'], is an OPTIONAL Domain separation byte and

L positive integer, is the requested number of output bytes.

Conceptually, a XOF can be viewed as a hash function with an infinitely long output truncated to *L* bytes. This means that calling a XOF with the same input parameters but two different lengths yields outputs such that the shorter one is a prefix of the longer one. Specifically, if $L_1 < L_2$, then TurboSHAKE(*M*, *D*, *L*₁) is the same as the first *L*₁ bytes of TurboSHAKE(*M*, *D*, *L*₂).

By default, the Domain separation byte is '1F'. For an API that does not support a domain separation byte, *D* MUST be the '1F'.

The TurboSHAKE instance produces output that is a hash of the (*M*, *D*) couple. If *D* is fixed, this becomes a hash of the Message *M*. However, a protocol that requires a number of independent hash functions can choose different values for *D* to implement these. Specifically, for any distinct values *D*₁ and *D*₂, TurboSHAKE(*M*, *D*₁, *L*₁) and TurboSHAKE(*M*, *D*₂, *L*₂) yield independent hashes of *M*.

Note that an implementation MAY propose an incremental input interface where the input string *M* is given in pieces. If so, the output MUST be the same as if the function was called with *M* equal to the concatenation of the different pieces in the order they were given. Independently, an implementation MAY propose an incremental output interface where the output string is requested in pieces of given lengths. When the output is formed by concatenating the pieces in the requested order, it MUST be the same as if the function was called with *L* equal to the sum of the given lengths.

2.2. Specifications

TurboSHAKE makes use of the permutation Keccak-p[1600,n_r=12], i.e., the permutation used in SHAKE and SHA-3 functions reduced to its last *n_r*=12 rounds and specified in FIPS 202, Sections 3.3 and 3.4 [FIPS202]. KP denotes this permutation.

Similarly to SHAKE128, TurboSHAKE128 is a sponge function calling this permutation KP with a rate of 168 bytes or 1344 bits. It follows that TurboSHAKE128 has a capacity of $1600 - 1344 = 256$ bits or 32 bytes. Respectively to SHAKE256, TurboSHAKE256 makes use of a rate of 136 bytes or 1088 bits, and has a capacity of 512 bits or 64 bytes.

	Rate	Capacity
TurboSHAKE128	168 Bytes	32 Bytes
TurboSHAKE256	136 Bytes	64 Bytes

We now describe the operations inside TurboSHAKE128.

- * First the input M' is formed by appending the domain separation byte D to the message M.
- * If the length of M' is not a multiple of 168 bytes then it is padded with zeros at the end to make it a multiple of 168 bytes. If M' is already a multiple of 168 bytes then no padding is added. Then a byte '80' is XORed to the last byte of the padded input M' and the resulting string is split into a sequence of 168-byte blocks.
- * M' never has a length of 0 bytes due to the presence of the domain separation byte.
- * As defined by the sponge construction, the process operates on a state and consists of two phases: the absorbing phase that processes the padded input M' and the squeezing phase that produces the output.
- * In the absorbing phase the state is initialized to all-zero. The message blocks are XORed into the first 168 bytes of the state. Each block absorbed is followed with an application of KP to the state.
- * In the squeezing phase the output is formed by taking the first 168 bytes of the state, applying KP to the state, and repeating as many times as is necessary.

TurboSHAKE256 performs the same steps but makes use of 136-byte blocks with respect to the padding, absorbing, and squeezing phases.

The definition of the TurboSHAKE functions equivalently implements the `pad10*1` rule; see Section 5.1 of [FIPS202] for a definition of `pad10*1`. While `M` can be empty, the `D` byte is always present and is in the `'01'-'7F'` range. This last byte serves as domain separation and integrates the first bit of padding of the `pad10*1` rule (hence it cannot be `'00'`). Additionally, it must leave room for the second bit of padding (hence it cannot have the MSB set to 1), should it be the last byte of the block. For more details, refer to Section 6.1 of [KT] and Section 3 of [TURBOSHAKE].

The pseudocode versions of TurboSHAKE128 and TurboSHAKE256 are provided respectively in Appendix A.2 and Appendix A.3.

3. KangarooTwelve: Tree hashing over TurboSHAKE

3.1. Interface

KangarooTwelve is a family of eXtensible Output Functions (XOF) consisting of the KT128 and KT256 instances. A KangarooTwelve instance takes as input parameters two byte-strings (`M`, `C`) and a positive integer `L` where

`M` byte-string, is the Message and

`C` byte-string, is an OPTIONAL Customization string and

`L` positive integer, the requested number of output bytes.

The Customization string MAY serve as domain separation. It is typically a short string such as a name or an identifier (e.g. URI, ODI...). It can serve the same purpose as TurboSHAKE's `D` input parameter (see Section 2.1), but with a larger range.

By default, the Customization string is the empty string. For an API that does not support a customization string parameter, `C` MUST be the empty string.

Note that an implementation MAY propose an interface with the input and/or output provided incrementally as specified in Section 2.1.

3.2. Specification of KT128

On top of the sponge function TurboSHAKE128, KT128 uses a Sakura-compatible tree hash mode [SAKURA]. First, merge `M` and the OPTIONAL `C` to a single input string `S` in a reversible way. `length_encode(|C|)` gives the length in bytes of `C` as a byte-string. See Section 3.3.

$$S = M \parallel C \parallel \text{length_encode}(|C|)$$

Then, split S into n chunks of 8192 bytes.

```
S = S_0 || .. || S_(n-1)
|S_0| = .. = |S_(n-2)| = 8192 bytes
|S_(n-1)| <= 8192 bytes
```

From S_1 .. $S_{(n-1)}$, compute the 32-byte Chaining Values CV_1 .. $CV_{(n-1)}$. In order to be optimally efficient, this computation MAY exploit the parallelism available on the platform such as SIMD instructions.

```
CV_i = TurboSHAKE128( S_i, '0B', 32 )
```

Compute the final node: $FinalNode$.

- * If $|S| \leq 8192$ bytes, $FinalNode = S$
- * Otherwise compute $FinalNode$ as follows:

```
FinalNode = S_0 || '03 00 00 00 00 00 00 00'
FinalNode = FinalNode || CV_1
..
FinalNode = FinalNode || CV_(n-1)
FinalNode = FinalNode || length_encode(n-1)
FinalNode = FinalNode || 'FF FF'
```

Finally, the KT128 output is retrieved:

- * If $|S| \leq 8192$ bytes, from $TurboSHAKE128(FinalNode, '07', L)$

```
KT128( M, C, L ) = TurboSHAKE128( FinalNode, '07', L )
```

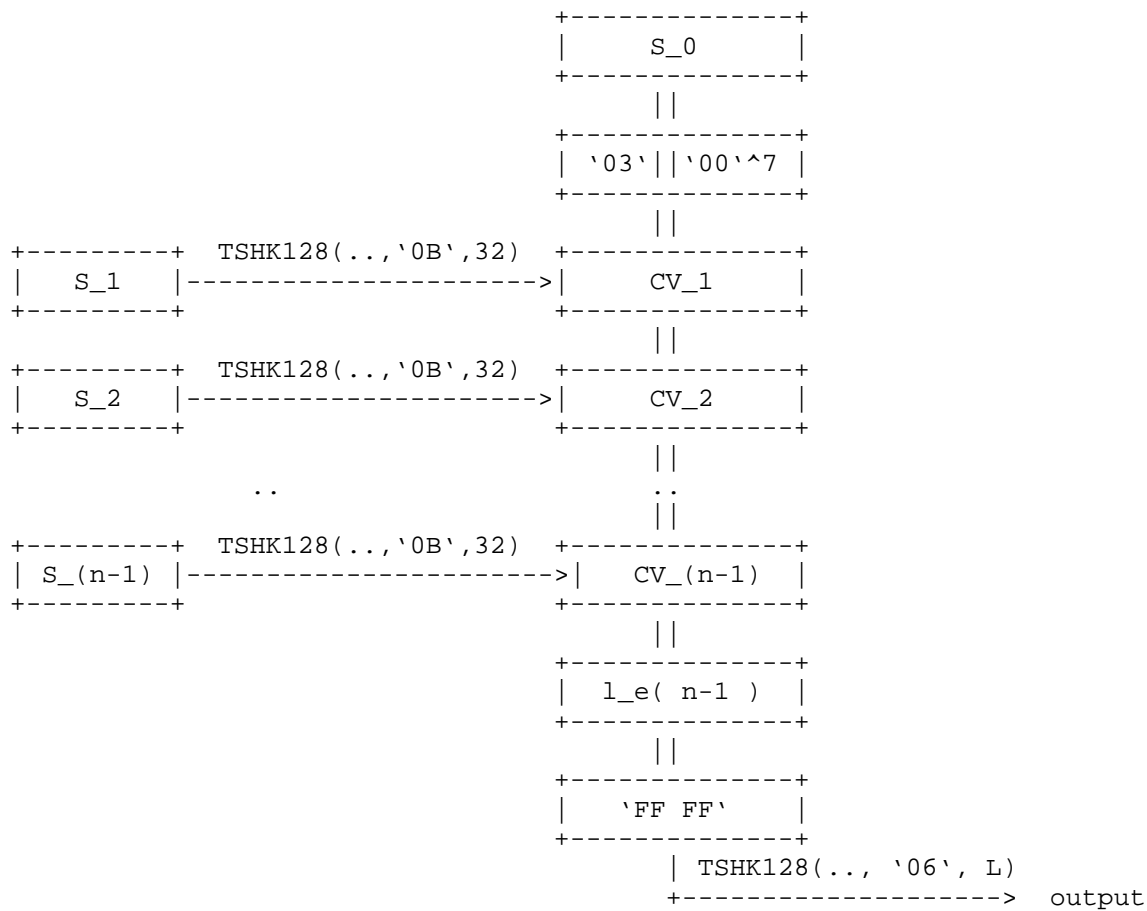
- * Otherwise from $TurboSHAKE128(FinalNode, '06', L)$

```
KT128( M, C, L ) = TurboSHAKE128( FinalNode, '06', L )
```

The following figure illustrates the computation flow of KT128 for $|S| \leq 8192$ bytes:

```
+-----+ TurboSHAKE128(..., '07', L)
|      S      |-----> output
+-----+
```

The following figure illustrates the computation flow of KT128 for $|S| > 8192$ bytes and where $TurboSHAKE128$ and $length_encode(x)$ are abbreviated as respectively $TSHK128$ and $l_e(x)$:



A pseudocode version is provided in Appendix A.4.

The table below gathers the values of the domain separation bytes used by the tree hash mode:

Type	Byte
SingleNode	'07'
IntermediateNode	'0B'
FinalNode	'06'

3.3. length_encode(x)

The function `length_encode` takes as inputs a non-negative integer $x < 256^{**}255$ and outputs a string of bytes $x_{(n-1)} || \dots || x_0 || n$ where

$$x = \text{sum of } 256^{**}i * x_i \text{ for } i \text{ from } 0 \text{ to } n-1$$

and where n is the smallest non-negative integer such that $x < 256^{**}n$. n is also the length of $x_{(n-1)} || \dots || x_0$.

As example, `length_encode(0) = '00'`, `length_encode(12) = '0C 01'` and `length_encode(65538) = '01 00 02 03'`

A pseudocode version is as follows where $\{ b \}$ denotes the byte of numerical value b .

```
length_encode(x):
  S = '00'^0

  while x > 0
    S = { x mod 256 } || S
    x = x / 256

  S = S || { |S| }

  return S
end
```

3.4. Specification of KT256

KT256 is specified exactly like KT128, with two differences:

- * All the calls to TurboSHAKE128 in KT128 are replaced with calls to TurboSHAKE256 in KT256.
- * The chaining values CV_1 to $CV_{(n-1)}$ are 64-byte long in KT256 and are computed as follows:

$$CV_i = \text{TurboSHAKE256}(S_i, '0B', 64)$$

A pseudocode version is provided in Appendix A.5.

4. Message authentication codes

Implementing a MAC with KT128 or KT256 MAY use a hash-then-MAC construction. This document defines and recommends a method called HopMAC:

```
HopMAC128(Key, M, C, L) = KT128(Key, KT128(M, C, 32), L)
HopMAC256(Key, M, C, L) = KT256(Key, KT256(M, C, 64), L)
```

Similarly to HMAC, HopMAC consists of two calls: an inner call compressing the message M and the optional customization string C to a digest, and an outer call computing the tag from the key and the digest.

Unlike HMAC, the inner call to KangarooTwelve in HopMAC is keyless and does not require additional protection against side channel attacks (SCA). Consequently, in an implementation that has to protect the HopMAC key against SCA only the outer call does need protection, and this amounts to a single execution of the underlying permutation (assuming the key length is at most 69 bytes).

In any case, TurboSHAKE128, TurboSHAKE256, KT128 and KT256 MAY be used to compute a MAC with the key reversibly prepended or appended to the input. For instance, one MAY compute a MAC on short messages simply calling KT128 with the key as the customization string, i.e., `MAC = KT128(M, Key, L)`.

5. Test vectors

Test vectors are based on the repetition of the pattern `'00 01 02 .. F9 FA'` with a specific length. `ptn(n)` defines a string by repeating the pattern `'00 01 02 .. F9 FA'` as many times as necessary and truncated to n bytes e.g.

```
Pattern for a length of 17 bytes:
ptn(17) =
'00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10'
```

Pattern for a length of 17**2 bytes:

ptn(17**2) =

```
'00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F
 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F
 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F
 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F
 A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF
 B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF
 C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF
 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF
 E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF
 F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA
 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
 20 21 22 23 24 25'
```

TurboSHAKE128(M='00'^0, D='1F', 32):

```
'1E 41 5F 1C 59 83 AF F2 16 92 17 27 7D 17 BB 53
 8C D9 45 A3 97 DD EC 54 1F 1C E4 1A F2 C1 B7 4C'
```

TurboSHAKE128(M='00'^0, D='1F', 64):

```
'1E 41 5F 1C 59 83 AF F2 16 92 17 27 7D 17 BB 53
 8C D9 45 A3 97 DD EC 54 1F 1C E4 1A F2 C1 B7 4C
 3E 8C CA E2 A4 DA E5 6C 84 A0 4C 23 85 C0 3C 15
 E8 19 3B DF 58 73 73 63 32 16 91 C0 54 62 C8 DF'
```

TurboSHAKE128(M='00'^0, D='1F', 10032), last 32 bytes:

```
'A3 B9 B0 38 59 00 CE 76 1F 22 AE D5 48 E7 54 DA
 10 A5 24 2D 62 E8 C6 58 E3 F3 A9 23 A7 55 56 07'
```

TurboSHAKE128(M=ptn(17**0 bytes), D='1F', 32):

```
'55 CE DD 6F 60 AF 7B B2 9A 40 42 AE 83 2E F3 F5
 8D B7 29 9F 89 3E BB 92 47 24 7D 85 69 58 DA A9'
```

TurboSHAKE128(M=ptn(17**1 bytes), D='1F', 32):

```
'9C 97 D0 36 A3 BA C8 19 DB 70 ED E0 CA 55 4E C6
 E4 C2 A1 A4 FF BF D9 EC 26 9C A6 A1 11 16 12 33'
```

TurboSHAKE128(M=ptn(17**2 bytes), D='1F', 32):

```
'96 C7 7C 27 9E 01 26 F7 FC 07 C9 B0 7F 5C DA E1
 E0 BE 60 BD BE 10 62 00 40 E7 5D 72 23 A6 24 D2'
```

```
TurboSHAKE128(M=ptn(17**3 bytes), D='1F', 32):  
  'D4 97 6E B5 6B CF 11 85 20 58 2B 70 9F 73 E1 D6  
    85 3E 00 1F DA F8 0E 1B 13 E0 D0 59 9D 5F B3 72'
```

```
TurboSHAKE128(M=ptn(17**4 bytes), D='1F', 32):  
  'DA 67 C7 03 9E 98 BF 53 0C F7 A3 78 30 C6 66 4E  
    14 CB AB 7F 54 0F 58 40 3B 1B 82 95 13 18 EE 5C'
```

```
TurboSHAKE128(M=ptn(17**5 bytes), D='1F', 32):  
  'B9 7A 90 6F BF 83 EF 7C 81 25 17 AB F3 B2 D0 AE  
    A0 C4 F6 03 18 CE 11 CF 10 39 25 12 7F 59 EE CD'
```

```
TurboSHAKE128(M=ptn(17**6 bytes), D='1F', 32):  
  '35 CD 49 4A DE DE D2 F2 52 39 AF 09 A7 B8 EF 0C  
    4D 1C A4 FE 2D 1A C3 70 FA 63 21 6F E7 B4 C2 B1'
```

```
TurboSHAKE128(M='FF FF FF', D='01', 32):  
  'BF 32 3F 94 04 94 E8 8E E1 C5 40 FE 66 0B E8 A0  
    C9 3F 43 D1 5E C0 06 99 84 62 FA 99 4E ED 5D AB'
```

```
TurboSHAKE128(M='FF', D='06', 32):  
  '8E C9 C6 64 65 ED 0D 4A 6C 35 D1 35 06 71 8D 68  
    7A 25 CB 05 C7 4C CA 1E 42 50 1A BD 83 87 4A 67'
```

```
TurboSHAKE128(M='FF FF FF', D='07', 32):  
  'B6 58 57 60 01 CA D9 B1 E5 F3 99 A9 F7 77 23 BB  
    A0 54 58 04 2D 68 20 6F 72 52 68 2D BA 36 63 ED'
```

```
TurboSHAKE128(M='FF FF FF FF FF FF FF', D='0B', 32):  
  '8D EE AA 1A EC 47 CC EE 56 9F 65 9C 21 DF A8 E1  
    12 DB 3C EE 37 B1 81 78 B2 AC D8 05 B7 99 CC 37'
```

```
TurboSHAKE128(M='FF', D='30', 32):  
  '55 31 22 E2 13 5E 36 3C 32 92 BE D2 C6 42 1F A2  
    32 BA B0 3D AA 07 C7 D6 63 66 03 28 65 06 32 5B'
```

```
TurboSHAKE128(M='FF FF FF', D='7F', 32):  
  '16 27 4C C6 56 D4 4C EF D4 22 39 5D 0F 90 53 BD  
    A6 D2 8E 12 2A BA 15 C7 65 E5 AD 0E 6E AF 26 F9'
```

```
TurboSHAKE256(M='00'^0, D='1F', 64):  
  '36 7A 32 9D AF EA 87 1C 78 02 EC 67 F9 05 AE 13  
    C5 76 95 DC 2C 66 63 C6 10 35 F5 9A 18 F8 E7 DB  
    11 ED C0 E1 2E 91 EA 60 EB 6B 32 DF 06 DD 7F 00  
    2F BA FA BB 6E 13 EC 1C C2 0D 99 55 47 60 0D B0'
```

```
TurboSHAKE256(M='00'^0, D='1F', 10032), last 32 bytes:  
  'AB EF A1 16 30 C6 61 26 92 49 74 26 85 EC 08 2F'
```

20 72 65 DC CF 2F 43 53 4E 9C 61 BA 0C 9D 1D 75`

TurboSHAKE256(M=ptn(17**0 bytes), D=`1F`, 64):

`3E 17 12 F9 28 F8 EA F1 05 46 32 B2 AA 0A 24 6E
D8 B0 C3 78 72 8F 60 BC 97 04 10 15 5C 28 82 0E
90 CC 90 D8 A3 00 6A A2 37 2C 5C 5E A1 76 B0 68
2B F2 2B AE 74 67 AC 94 F7 4D 43 D3 9B 04 82 E2`

TurboSHAKE256(M=ptn(17**1 bytes), D=`1F`, 64):

`B3 BA B0 30 0E 6A 19 1F BE 61 37 93 98 35 92 35
78 79 4E A5 48 43 F5 01 10 90 FA 2F 37 80 A9 E5
CB 22 C5 9D 78 B4 0A 0F BF F9 E6 72 C0 FB E0 97
0B D2 C8 45 09 1C 60 44 D6 87 05 4D A5 D8 E9 C7`

TurboSHAKE256(M=ptn(17**2 bytes), D=`1F`, 64):

`66 B8 10 DB 8E 90 78 04 24 C0 84 73 72 FD C9 57
10 88 2F DE 31 C6 DF 75 BE B9 D4 CD 93 05 CF CA
E3 5E 7B 83 E8 B7 E6 EB 4B 78 60 58 80 11 63 16
FE 2C 07 8A 09 B9 4A D7 B8 21 3C 0A 73 8B 65 C0`

TurboSHAKE256(M=ptn(17**3 bytes), D=`1F`, 64):

`C7 4E BC 91 9A 5B 3B 0D D1 22 81 85 BA 02 D2 9E
F4 42 D6 9D 3D 42 76 A9 3E FE 0B F9 A1 6A 7D C0
CD 4E AB AD AB 8C D7 A5 ED D9 66 95 F5 D3 60 AB
E0 9E 2C 65 11 A3 EC 39 7D A3 B7 6B 9E 16 74 FB`

TurboSHAKE256(M=ptn(17**4 bytes), D=`1F`, 64):

`02 CC 3A 88 97 E6 F4 F6 CC B6 FD 46 63 1B 1F 52
07 B6 6C 6D E9 C7 B5 5B 2D 1A 23 13 4A 17 0A FD
AC 23 4E AB A9 A7 7C FF 88 C1 F0 20 B7 37 24 61
8C 56 87 B3 62 C4 30 B2 48 CD 38 64 7F 84 8A 1D`

TurboSHAKE256(M=ptn(17**5 bytes), D=`1F`, 64):

`AD D5 3B 06 54 3E 58 4B 58 23 F6 26 99 6A EE 50
FE 45 ED 15 F2 02 43 A7 16 54 85 AC B4 AA 76 B4
FF DA 75 CE DF 6D 8C DC 95 C3 32 BD 56 F4 B9 86
B5 8B B1 7D 17 78 BF C1 B1 A9 75 45 CD F4 EC 9F`

TurboSHAKE256(M=ptn(17**6 bytes), D=`1F`, 64):

`9E 11 BC 59 C2 4E 73 99 3C 14 84 EC 66 35 8E F7
1D B7 4A EF D8 4E 12 3F 78 00 BA 9C 48 53 E0 2C
FE 70 1D 9E 6B B7 65 A3 04 F0 DC 34 A4 EE 3B A8
2C 41 0F 0D A7 0E 86 BF BD 90 EA 87 7C 2D 61 04`

TurboSHAKE256(M=`FF FF FF`, D=`01`, 64):

`D2 1C 6F BB F5 87 FA 22 82 F2 9A EA 62 01 75 FB
02 57 41 3A F7 8A 0B 1B 2A 87 41 9C E0 31 D9 33
AE 7A 4D 38 33 27 A8 A1 76 41 A3 4F 8A 1D 10 03`

AD 7D A6 B7 2D BA 84 BB 62 FE F2 8F 62 F1 24 24`

TurboSHAKE256(M=`FF`, D=`06`, 64):

`73 8D 7B 4E 37 D1 8B 7F 22 AD 1B 53 13 E3 57 E3
DD 7D 07 05 6A 26 A3 03 C4 33 FA 35 33 45 52 80
F4 F5 A7 D4 F7 00 EF B4 37 FE 6D 28 14 05 E0 7B
E3 2A 0A 97 2E 22 E6 3A DC 1B 09 0D AE FE 00 4B`

TurboSHAKE256(M=`FF FF FF`, D=`07`, 64):

`18 B3 B5 B7 06 1C 2E 67 C1 75 3A 00 E6 AD 7E D7
BA 1C 90 6C F9 3E FB 70 92 EA F2 7F BE EB B7 55
AE 6E 29 24 93 C1 10 E4 8D 26 00 28 49 2B 8E 09
B5 50 06 12 B8 F2 57 89 85 DE D5 35 7D 00 EC 67`

TurboSHAKE256(M=`FF FF FF FF FF FF FF`, D=`0B`, 64):

`BB 36 76 49 51 EC 97 E9 D8 5F 7E E9 A6 7A 77 18
FC 00 5C F4 25 56 BE 79 CE 12 C0 BD E5 0E 57 36
D6 63 2B 0D 0D FB 20 2D 1B BB 8F FE 3D D7 4C B0
08 34 FA 75 6C B0 34 71 BA B1 3A 1E 2C 16 B3 C0`

TurboSHAKE256(M=`FF`, D=`30`, 64):

`F3 FE 12 87 3D 34 BC BB 2E 60 87 79 D6 B7 0E 7F
86 BE C7 E9 0B F1 13 CB D4 FD D0 C4 E2 F4 62 5E
14 8D D7 EE 1A 52 77 6C F7 7F 24 05 14 D9 CC FC
3B 5D DA B8 EE 25 5E 39 EE 38 90 72 96 2C 11 1A`

TurboSHAKE256(M=`FF FF FF`, D=`7F`, 64):

`AB E5 69 C1 F7 7E C3 40 F0 27 05 E7 D3 7C 9A B7
E1 55 51 6E 4A 6A 15 00 21 D7 0B 6F AC 0B B4 0C
06 9F 9A 98 28 A0 D5 75 CD 99 F9 BA E4 35 AB 1A
CF 7E D9 11 0B A9 7C E0 38 8D 07 4B AC 76 87 76`

KT128(M=`00`^0, C=`00`^0, 32):

`1A C2 D4 50 FC 3B 42 05 D1 9D A7 BF CA 1B 37 51
3C 08 03 57 7A C7 16 7F 06 FE 2C E1 F0 EF 39 E5`

KT128(M=`00`^0, C=`00`^0, 64):

`1A C2 D4 50 FC 3B 42 05 D1 9D A7 BF CA 1B 37 51
3C 08 03 57 7A C7 16 7F 06 FE 2C E1 F0 EF 39 E5
42 69 C0 56 B8 C8 2E 48 27 60 38 B6 D2 92 96 6C
C0 7A 3D 46 45 27 2E 31 FF 38 50 81 39 EB 0A 71`

KT128(M=`00`^0, C=`00`^0, 10032), last 32 bytes:

`E8 DC 56 36 42 F7 22 8C 84 68 4C 89 84 05 D3 A8
34 79 91 58 C0 79 B1 28 80 27 7A 1D 28 E2 FF 6D`

KT128(M=ptn(1 bytes), C=`00`^0, 32):

`2B DA 92 45 0E 8B 14 7F 8A 7C B6 29 E7 84 A0 58`


```
EF CA 7C F7 D8 21 8E 02 D3 45 DF AA 65 24 4A 1F`

KT128(M=ptn(17 bytes), C=`00`^0, 32):
`6B F7 5F A2 23 91 98 DB 47 72 E3 64 78 F8 E1 9B
 0F 37 12 05 F6 A9 A9 3A 27 3F 51 DF 37 12 28 88`

KT128(M=ptn(17**2 bytes), C=`00`^0, 32):
`0C 31 5E BC DE DB F6 14 26 DE 7D CF 8F B7 25 D1
 E7 46 75 D7 F5 32 7A 50 67 F3 67 B1 08 EC B6 7C`

KT128(M=ptn(17**3 bytes), C=`00`^0, 32):
`CB 55 2E 2E C7 7D 99 10 70 1D 57 8B 45 7D DF 77
 2C 12 E3 22 E4 EE 7F E4 17 F9 2C 75 8F 0D 59 D0`

KT128(M=ptn(17**4 bytes), C=`00`^0, 32):
`87 01 04 5E 22 20 53 45 FF 4D DA 05 55 5C BB 5C
 3A F1 A7 71 C2 B8 9B AE F3 7D B4 3D 99 98 B9 FE`

KT128(M=ptn(17**5 bytes), C=`00`^0, 32):
`84 4D 61 09 33 B1 B9 96 3C BD EB 5A E3 B6 B0 5C
 C7 CB D6 7C EE DF 88 3E B6 78 A0 A8 E0 37 16 82`

KT128(M=ptn(17**6 bytes), C=`00`^0, 32):
`3C 39 07 82 A8 A4 E8 9F A6 36 7F 72 FE AA F1 32
 55 C8 D9 58 78 48 1D 3C D8 CE 85 F5 8E 88 0A F8`

KT128(`00`^0, C=ptn(1 bytes), 32):
`FA B6 58 DB 63 E9 4A 24 61 88 BF 7A F6 9A 13 30
 45 F4 6E E9 84 C5 6E 3C 33 28 CA AF 1A A1 A5 83`

KT128(`FF`, C=ptn(41 bytes), 32):
`D8 48 C5 06 8C ED 73 6F 44 62 15 9B 98 67 FD 4C
 20 B8 08 AC C3 D5 BC 48 E0 B0 6B A0 A3 76 2E C4`

KT128(`FF FF FF`, C=ptn(41**2 bytes), 32):
`C3 89 E5 00 9A E5 71 20 85 4C 2E 8C 64 67 0A C0
 13 58 CF 4C 1B AF 89 44 7A 72 42 34 DC 7C ED 74`

KT128(`FF FF FF FF FF FF FF`, C=ptn(41**3 bytes), 32):
`75 D2 F8 6A 2E 64 45 66 72 6B 4F BC FC 56 57 B9
 DB CF 07 0C 7B 0D CA 06 45 0A B2 91 D7 44 3B CF`

KT128(M=ptn(8191 bytes), C=`00`^0, 32):
`1B 57 76 36 F7 23 64 3E 99 0C C7 D6 A6 59 83 74
 36 FD 6A 10 36 26 60 0E B8 30 1C D1 DB E5 53 D6`

KT128(M=ptn(8192 bytes), C=`00`^0, 32):
`48 F2 56 F6 77 2F 9E DF B6 A8 B6 61 EC 92 DC 93
```

```
B9 5E BD 05 A0 8A 17 B3 9A E3 49 08 70 C9 26 C3`

KT128(M=ptn(8192 bytes), C=ptn(8189 bytes), 32):
`3E D1 2F 70 FB 05 DD B5 86 89 51 0A B3 E4 D2 3C
  6C 60 33 84 9A A0 1E 1D 8C 22 0A 29 7F ED CD 0B`

KT128(M=ptn(8192 bytes), C=ptn(8190 bytes), 32):
`6A 7C 1B 6A 5C D0 D8 C9 CA 94 3A 4A 21 6C C6 46
  04 55 9A 2E A4 5F 78 57 0A 15 25 3D 67 BA 00 AE`

KT256(M=`00`^0, C=`00`^0, 64):
`B2 3D 2E 9C EA 9F 49 04 E0 2B EC 06 81 7F C1 0C
  E3 8C E8 E9 3E F4 C8 9E 65 37 07 6A F8 64 64 04
  E3 E8 B6 81 07 B8 83 3A 5D 30 49 0A A3 34 82 35
  3F D4 AD C7 14 8E CB 78 28 55 00 3A AE BD E4 A9`

KT256(M=`00`^0, C=`00`^0, 128):
`B2 3D 2E 9C EA 9F 49 04 E0 2B EC 06 81 7F C1 0C
  E3 8C E8 E9 3E F4 C8 9E 65 37 07 6A F8 64 64 04
  E3 E8 B6 81 07 B8 83 3A 5D 30 49 0A A3 34 82 35
  3F D4 AD C7 14 8E CB 78 28 55 00 3A AE BD E4 A9
  B0 92 53 19 D8 EA 1E 12 1A 60 98 21 EC 19 EF EA
  89 E6 D0 8D AE E1 66 2B 69 C8 40 28 9F 18 8B A8
  60 F5 57 60 B6 1F 82 11 4C 03 0C 97 E5 17 84 49
  60 8C CD 2C D2 D9 19 FC 78 29 FF 69 93 1A C4 D0`

KT256(M=`00`^0, C=`00`^0, 10064), last 64 bytes:
`AD 4A 1D 71 8C F9 50 50 67 09 A4 C3 33 96 13 9B
  44 49 04 1F C7 9A 05 D6 8D A3 5F 1E 45 35 22 E0
  56 C6 4F E9 49 58 E7 08 5F 29 64 88 82 59 B9 93
  27 52 F3 CC D8 55 28 8E FE E5 FC BB 8B 56 30 69`

KT256(M=ptn(1 bytes), C=`00`^0, 64):
`0D 00 5A 19 40 85 36 02 17 12 8C F1 7F 91 E1 F7
  13 14 EF A5 56 45 39 D4 44 91 2E 34 37 EF A1 7F
  82 DB 6F 6F FE 76 E7 81 EA A0 68 BC E0 1F 2B BF
  81 EA CB 98 3D 72 30 F2 FB 02 83 4A 21 B1 DD D0`

KT256(M=ptn(17 bytes), C=`00`^0, 64):
`1B A3 C0 2B 1F C5 14 47 4F 06 C8 97 99 78 A9 05
  6C 84 83 F4 A1 B6 3D 0D CC EF E3 A2 8A 2F 32 3E
  1C DC CA 40 EB F0 06 AC 76 EF 03 97 15 23 46 83
  7B 12 77 D3 E7 FA A9 C9 65 3B 19 07 50 98 52 7B`

KT256(M=ptn(17*2 bytes), C=`00`^0, 64):
`DE 8C CB C6 3E 0F 13 3E BB 44 16 81 4D 4C 66 F6
  91 BB F8 B6 A6 1E C0 A7 70 0F 83 6B 08 6C B0 29
  D5 4F 12 AC 71 59 47 2C 72 DB 11 8C 35 B4 E6 AA
```

21 3C 65 62 CA AA 9D CC 51 89 59 E6 9B 10 F3 BA`

KT256(M=ptn(17**3 bytes), C=`00`^0, 64):

`64 7E FB 49 FE 9D 71 75 00 17 1B 41 E7 F1 1B D4
91 54 44 43 20 99 97 CE 1C 25 30 D1 5E B1 FF BB
59 89 35 EF 95 45 28 FF C1 52 B1 E4 D7 31 EE 26
83 68 06 74 36 5C D1 91 D5 62 BA E7 53 B8 4A A5`

KT256(M=ptn(17**4 bytes), C=`00`^0, 64):

`B0 62 75 D2 84 CD 1C F2 05 BC BE 57 DC CD 3E C1
FF 66 86 E3 ED 15 77 63 83 E1 F2 FA 3C 6A C8 F0
8B F8 A1 62 82 9D B1 A4 4B 2A 43 FF 83 DD 89 C3
CF 1C EB 61 ED E6 59 76 6D 5C CF 81 7A 62 BA 8D`

KT256(M=ptn(17**5 bytes), C=`00`^0, 64):

`94 73 83 1D 76 A4 C7 BF 77 AC E4 5B 59 F1 45 8B
16 73 D6 4B CD 87 7A 7C 66 B2 66 4A A6 DD 14 9E
60 EA B7 1B 5C 2B AB 85 8C 07 4D ED 81 DD CE 2B
40 22 B5 21 59 35 C0 D4 D1 9B F5 11 AE EB 07 72`

KT256(M=ptn(17**6 bytes), C=`00`^0, 64):

`06 52 B7 40 D7 8C 5E 1F 7C 8D CC 17 77 09 73 82
76 8B 7F F3 8F 9A 7A 20 F2 9F 41 3B B1 B3 04 5B
31 A5 57 8F 56 8F 91 1E 09 CF 44 74 6D A8 42 24
A5 26 6E 96 A4 A5 35 E8 71 32 4E 4F 9C 70 04 DA`

KT256(`00`^0, C=ptn(1 bytes), 64):

`92 80 F5 CC 39 B5 4A 5A 59 4E C6 3D E0 BB 99 37
1E 46 09 D4 4B F8 45 C2 F5 B8 C3 16 D7 2B 15 98
11 F7 48 F2 3E 3F AB BE 5C 32 26 EC 96 C6 21 86
DF 2D 33 E9 DF 74 C5 06 9C EE CB B4 DD 10 EF F6`

KT256(`FF`, C=ptn(41 bytes), 64):

`47 EF 96 DD 61 6F 20 09 37 AA 78 47 E3 4E C2 FE
AE 80 87 E3 76 1D C0 F8 C1 A1 54 F5 1D C9 CC F8
45 D7 AD BC E5 7F F6 4B 63 97 22 C6 A1 67 2E 3B
F5 37 2D 87 E0 0A FF 89 BE 97 24 07 56 99 88 53`

KT256(`FF FF FF`, C=ptn(41**2 bytes), 64):

`3B 48 66 7A 50 51 C5 96 6C 53 C5 D4 2B 95 DE 45
1E 05 58 4E 78 06 E2 FB 76 5E DA 95 90 74 17 2C
B4 38 A9 E9 1D DE 33 7C 98 E9 C4 1B ED 94 C4 E0
AE F4 31 D0 B6 4E F2 32 4F 79 32 CA A6 F5 49 69`

KT256(`FF FF FF FF FF FF FF`, C=ptn(41**3 bytes), 64):

`E0 91 1C C0 00 25 E1 54 08 31 E2 66 D9 4A DD 9B
98 71 21 42 B8 0D 26 29 E6 43 AA C4 EF AF 5A 3A
30 A8 8C BF 4A C2 A9 1A 24 32 74 30 54 FB CC 98`

```
97 67 0E 86 BA 8C EC 2F C2 AC E9 C9 66 36 97 24`
```

```
KT256(M=ptn(8191 bytes), C=`00`^0, 64):
```

```
`30 81 43 4D 93 A4 10 8D 8D 8A 33 05 B8 96 82 CE
BE DC 7C A4 EA 8A 3C E8 69 FB B7 3C BE 4A 58 EE
F6 F2 4D E3 8F FC 17 05 14 C7 0E 7A B2 D0 1F 03
81 26 16 E8 63 D7 69 AF B3 75 31 93 BA 04 5B 20`
```

```
KT256(M=ptn(8192 bytes), C=`00`^0, 64):
```

```
`C6 EE 8E 2A D3 20 0C 01 8A C8 7A AA 03 1C DA C2
21 21 B4 12 D0 7D C6 E0 DC CB B5 34 23 74 7E 9A
1C 18 83 4D 99 DF 59 6C F0 CF 4B 8D FA FB 7B F0
2D 13 9D 0C 90 35 72 5A DC 1A 01 B7 23 0A 41 FA`
```

```
KT256(M=ptn(8192 bytes), C=ptn(8189 bytes), 64):
```

```
`74 E4 78 79 F1 0A 9C 5D 11 BD 2D A7 E1 94 FE 57
E8 63 78 BF 3C 3F 74 48 EF F3 C5 76 A0 F1 8C 5C
AA E0 99 99 79 51 20 90 A7 F3 48 AF 42 60 D4 DE
3C 37 F1 EC AF 8D 2C 2C 96 C1 D1 6C 64 B1 24 96`
```

```
KT256(M=ptn(8192 bytes), C=ptn(8190 bytes), 64):
```

```
`F4 B5 90 8B 92 9F FE 01 E0 F7 9E C2 F2 12 43 D4
1A 39 6B 2E 73 03 A6 AF 1D 63 99 CD 6C 7A 0A 2D
D7 C4 F6 07 E8 27 7F 9C 9B 1C B4 AB 9D DC 59 D4
B9 2D 1F C7 55 84 41 F1 83 2C 32 79 A4 24 1B 8B`
```

6. IANA Considerations

In the Named Information Hash Algorithm Registry, k12-256 refers to the hash function obtained by evaluating KT128 on the input message with default C (the empty string) and L = 32 bytes (256 bits). Similarly, k12-512 refers to the hash function obtained by evaluating KT256 on the input message with default C (the empty string) and L = 64 bytes (512 bits).

In the COSE Algorithms registry, the following entries are assigned to TurboSHAKE and KangarooTwelve:

Name	Value	Description	Capabilities
TurboSHAKE128	-261	TurboSHAKE128 XOF	[kty]
TurboSHAKE256	-262	TurboSHAKE256 XOF	[kty]
KT128	-263	KT128 XOF	[kty]
KT256	-264	KT256 XOF	[kty]

7. Security Considerations

This document is meant to serve as a stable reference and an implementation guide for the KangarooTwelve and TurboSHAKE extendable Output Functions. The security assurance of these functions relies on the cryptanalysis of reduced-round versions of Keccak and they have the same claimed security strength as their corresponding SHAKE functions.

	security claim
TurboSHAKE128	128 bits (same as SHAKE128)
KT128	128 bits (same as SHAKE128)
TurboSHAKE256	256 bits (same as SHAKE256)
KT256	256 bits (same as SHAKE256)

To be more precise, KT128 is made of two layers:

- * The inner function TurboSHAKE128. The security assurance of this layer relies on cryptanalysis. The TurboSHAKE128 function is exactly Keccak[r=1344, c=256] (as in SHAKE128) reduced to 12 rounds. Any cryptanalysis of reduced-round Keccak is also cryptanalysis of reduced-round TurboSHAKE128 (provided the number of rounds attacked is not higher than 12).
- * The tree hashing over TurboSHAKE128. This layer is a mode on top of TurboSHAKE128 that does not introduce any vulnerability thanks to the use of Sakura coding proven secure in [SAKURA].

This reasoning is detailed and formalized in [KT].

KT256 is structured as KT128, except that it uses TurboSHAKE256 as inner function. The TurboSHAKE256 function is exactly Keccak[r=1088, c=512] (as in SHAKE256) reduced to 12 rounds, and the same reasoning on cryptanalysis applies.

TurboSHAKE128 and KT128 aim at 128-bit security. To achieve 128-bit security strength, the output L MUST be chosen long enough so that there are no generic attacks that violate 128-bit security. So for 128-bit (second) preimage security the output should be at least 128 bits, for 128 bits of security against multi-target preimage attacks with T targets the output should be at least $128 + \log_2(T)$ bits and for 128-bit collision security the output should be at least 256 bits. Furthermore, when the output length is at least 256 bits, TurboSHAKE128 and KT128 achieve NIST's post-quantum security level 2 [NISTPQ].

Similarly, TurboSHAKE256 and KT256 aim at 256-bit security. To achieve 256-bit security strength, the output L MUST be chosen long enough so that there are no generic attacks that violate 256-bit security. So for 256-bit (second) preimage security the output should be at least 256 bits, for 256 bits of security against multi-target preimage attacks with T targets the output should be at least $256 + \log_2(T)$ bits and for 256-bit collision security the output should be at least 512 bits. Furthermore, when the output length is at least 512 bits, TurboSHAKE256 and KT256 achieve NIST's post-quantum security level 5 [NISTPQ].

Unlike the SHA-256 and SHA-512 functions, TurboSHAKE128, TurboSHAKE256, KT128 and KT256 do not suffer from the length extension weakness, and therefore do not require the use of the HMAC construction for instance when used for MAC computation [FIPS198]. Also, they can naturally be used as a key derivation function. The input must be an injective encoding of secret and diversification material, and the output can be taken as the derived key(s). The input does not need to be uniformly distributed, e.g., it can be a shared secret produced by the Diffie-Hellman or ECDH protocol, but it needs to have sufficient min-entropy.

Lastly, as KT128 and KT256 use TurboSHAKE with three values for D, namely 0x06, 0x07, and 0x0B. Protocols that use both KT128 and TurboSHAKE128, or both KT256 and TurboSHAKE256, SHOULD avoid using these three values for D.

8. References

8.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [FIPS202] National Institute of Standards and Technology, "FIPS PUB 202 - SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", WWW <http://dx.doi.org/10.6028/NIST.FIPS.202>, August 2015.
- [SP800-185] National Institute of Standards and Technology, "NIST Special Publication 800-185 SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash", WWW <https://doi.org/10.6028/NIST.SP.800-185>, December 2016.

8.2. Informative References

- [TURBOSHAKE] Bertoni, G., Daemen, J., Hoffert, S., Peeters, M., Van Assche, G., Van Keer, R., and B. Viguier, "TurboSHAKE", WWW <http://eprint.iacr.org/2023/342>, March 2023.
- [KT] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G., Van Keer, R., and B. Viguier, "KangarooTwelve: fast hashing based on Keccak-p", WWW https://link.springer.com/chapter/10.1007/978-3-319-93387-0_21, WWW <http://eprint.iacr.org/2016/770.pdf>, July 2018.
- [SAKURA] Bertoni, G., Daemen, J., Peeters, M., and G. Van Assche, "Sakura: a flexible coding for tree hashing", WWW https://link.springer.com/chapter/10.1007/978-3-319-07536-5_14, WWW <http://eprint.iacr.org/2013/231.pdf>, June 2014.
- [KECCAK_CRYPTANALYSIS] Keccak Team, "Summary of Third-party cryptanalysis of Keccak", WWW https://www.keccak.team/third_party.html, 2022.
- [XKCP] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G., and R. Van Keer, "eXtended Keccak Code Package", WWW <https://github.com/XKCP/XKCP>, December 2022.

- [NISTPQ] National Institute of Standards and Technology,
"Submission Requirements and Evaluation Criteria for the
Post-Quantum Cryptography Standardization Process", WWW
<https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf>, December 2016.
- [FIPS180] National Institute of Standards and Technology (NIST),
"Secure Hash Standard (SHS)", FIPS PUB 180-4,
WWW <https://doi.org/10.6028/NIST.FIPS.180-4>, August 2015.
- [FIPS198] National Institute of Standards and Technology (NIST),
"The Keyed-Hash Message Authentication Code (HMAC)", FIPS
PUB 198-1, WWW <https://doi.org/10.6028/NIST.FIPS.198-1>,
July 2008.

Appendix A. Pseudocode

The sub-sections of this appendix contain pseudocode definitions of TurboSHAKE128, TurboSHAKE256 and KangarooTwelve. Standalone Python versions are also available in the Keccak Code Package [XKCP] and in [KT]

A.1. Keccak-p[1600,n_r=12]

```
KP(state):
  RC[0] = '8B 80 00 80 00 00 00 00'
  RC[1] = '8B 00 00 00 00 00 00 80'
  RC[2] = '89 80 00 00 00 00 00 80'
  RC[3] = '03 80 00 00 00 00 00 80'
  RC[4] = '02 80 00 00 00 00 00 80'
  RC[5] = '80 00 00 00 00 00 00 80'
  RC[6] = '0A 80 00 00 00 00 00 00'
  RC[7] = '0A 00 00 80 00 00 00 80'
  RC[8] = '81 80 00 80 00 00 00 80'
  RC[9] = '80 80 00 00 00 00 00 80'
  RC[10] = '01 00 00 80 00 00 00 00'
  RC[11] = '08 80 00 80 00 00 00 80'

  for x from 0 to 4
    for y from 0 to 4
      lanes[x][y] = state[8*(x+5*y):8*(x+5*y)+8]

  for round from 0 to 11
    # theta
    for x from 0 to 4
      C[x] = lanes[x][0]
      C[x] ^= lanes[x][1]
```



```

    C[x] ^= lanes[x][2]
    C[x] ^= lanes[x][3]
    C[x] ^= lanes[x][4]
  for x from 0 to 4
    D[x] = C[(x+4) mod 5] ^ ROL64(C[(x+1) mod 5], 1)
  for y from 0 to 4
    for x from 0 to 4
      lanes[x][y] = lanes[x][y]^D[x]

  # rho and pi
  (x, y) = (1, 0)
  current = lanes[x][y]
  for t from 0 to 23
    (x, y) = (y, (2*x+3*y) mod 5)
    (current, lanes[x][y]) =
      (lanes[x][y], ROL64(current, (t+1)*(t+2)/2))

  # chi
  for y from 0 to 4
    for x from 0 to 4
      T[x] = lanes[x][y]
    for x from 0 to 4
      lanes[x][y] = T[x] ^ ((not T[(x+1) mod 5]) & T[(x+2) mod 5])

  # iota
  lanes[0][0] ^= RC[round]

state = `00`^0
for y from 0 to 4
  for x from 0 to 4
    state = state || lanes[x][y]

return state
end

```

where ROL64(x, y) is a rotation of the 'x' 64-bit word toward the bits with higher indexes by 'y' positions. The 8-bytes byte-string x is interpreted as a 64-bit word in little-endian format.

A.2. TurboSHAKE128

```
TurboSHAKE128(message, separationByte, outputByteLen):
    offset = 0
    state = '00'^200
    input = message || separationByte

    # === Absorb complete blocks ===
    while offset < |input| - 168
        state ^= input[offset : offset + 168] || '00'^32
        state = KP(state)
        offset += 168

    # === Absorb last block and treatment of padding ===
    LastBlockLength = |input| - offset
    state ^= input[offset:] || '00'^(200-LastBlockLength)
    state ^= '00'^167 || '80' || '00'^32
    state = KP(state)

    # === Squeeze ===
    output = '00'^0
    while outputByteLen > 168
        output = output || state[0:168]
        outputByteLen -= 168
        state = KP(state)

    output = output || state[0:outputByteLen]

    return output
```

A.3. TurboSHAKE256

```
TurboSHAKE256(message, separationByte, outputByteLen):
    offset = 0
    state = '00'^200
    input = message || separationByte

    # === Absorb complete blocks ===
    while offset < |input| - 136
        state ^= input[offset : offset + 136] || '00'^64
        state = KP(state)
        offset += 136

    # === Absorb last block and treatment of padding ===
    LastBlockLength = |input| - offset
    state ^= input[offset:] || '00'^(200-LastBlockLength)
    state ^= '00'^135 || '80' || '00'^64
    state = KP(state)

    # === Squeeze ===
    output = '00'^0
    while outputByteLen > 136
        output = output || state[0:136]
        outputByteLen -= 136
        state = KP(state)

    output = output || state[0:outputByteLen]

    return output
```

A.4. KT128

```
KT128(inputMessage, customString, outputByteLen):
  S = inputMessage || customString
  S = S || length_encode( |customString| )

  if |S| <= 8192
    return TurboSHAKE128(S, '07', outputByteLen)
  else
    # === Kangaroo hopping ===
    FinalNode = S[0:8192] || '03' || '00'^7
    offset = 8192
    numBlock = 0
    while offset < |S|
      blockSize = min( |S| - offset, 8192)
      CV = TurboSHAKE128(S[offset : offset + blockSize], '0B', 32)
      FinalNode = FinalNode || CV
      numBlock += 1
      offset += blockSize

    FinalNode = FinalNode || length_encode( numBlock ) || 'FF FF'

    return TurboSHAKE128(FinalNode, '06', outputByteLen)
  end
```

A.5. KT256

```
KT256(inputMessage, customString, outputByteLen):
  S = inputMessage || customString
  S = S || length_encode( |customString| )

  if |S| <= 8192
    return TurboSHAKE256(S, '07', outputByteLen)
  else
    # === Kangaroo hopping ===
    FinalNode = S[0:8192] || '03' || '00'^7
    offset = 8192
    numBlock = 0
    while offset < |S|
      blockSize = min( |S| - offset, 8192)
      CV = TurboSHAKE256(S[offset : offset + blockSize], '0B', 64)
      FinalNode = FinalNode || CV
      numBlock += 1
      offset += blockSize

    FinalNode = FinalNode || length_encode( numBlock ) || 'FF FF'

    return TurboSHAKE256(FinalNode, '06', outputByteLen)
  end
```

Authors' Addresses

Benoît Viguiier
ABN AMRO Bank
Groenelaan 2
Amstelveen
Email: cs.ru.nl@viguier.nl

David Wong (editor)
zkSecurity
Email: davidwong.crypto@gmail.com

Gilles Van Assche (editor)
STMicroelectronics
Email: gilles.vanassche@st.com

Quynh Dang (editor)
National Institute of Standards and Technology
Email: quynh.dang@nist.gov

Joan Daemen (editor)
Radboud University
Email: joan@cs.ru.nl