Workgroup: Crypto Forum
Internet-Draft:
draft-irtf-cfrg-kangarootwelve-11
Published: 20 June 2023
Intended Status: Informational
Expires: 22 December 2023
Authors: B. Viguier, ABN AMRO Bank
D. Wong, Ed., O(1) Labs
G. Van Assche, Ed., STMicroelectronics
Q. Dang, Ed., NIST
J. Daemen, Ed., Radboud University

KangarooTwelve and TurboSHAKE

Abstract

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

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

This document 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 22 December 2023.

Copyright Notice

Copyright (c) 2023 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
  1.1. Conventions
2. TurboSHAKE
  2.1. Interface
  2.2. Specifications
3. KangarooTwelve: Tree hashing over TurboSHAKE128
  3.1. Interface
  3.2. Specification
  3.3. length_encode(x)
4. Message authentication codes
5. Test vectors
6. IANA Considerations
7. Security Considerations
8. References
  8.1. Normative References
  8.2. Informative References
Appendix A. Pseudocode
  A.1. Keccak-p[1600,n_r=12]
  A.2. TurboSHAKE128
  A.3. TurboSHAKE256
  A.4. KangarooTwelve
Authors' Addresses

1. Introduction

This document defines the TurboSHAKE128, TurboSHAKE256 [TURBOSHAKE] and KangarooTwelve [K12] 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 twice faster.

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 TurboSHAKE128 on each of them separately before applying TurboSHAKE128 again on the combination of the first fragment and the digests. It makes use of Sakura coding for ensuring soundness of the tree hashing mode [SAKURA]. The use of TurboSHAKE128 in KangarooTwelve makes it faster than ParallelHash.

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

With respect to [FIPS202] and [SP800-185] functions, TurboSHAKE128, TurboSHAKE256 and KangarooTwelve 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], KangarooTwelve allows the use of a customization string.

* Unlike any [FIPS202] and [SP800-185] functions but ParallelHash, KangarooTwelve exploits available parallelism.

* Unlike ParallelHash, KangarooTwelve does 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 it faster than any function defined in [FIPS202]. KangarooTwelve immediately benefits from the same speed up, improving over [FIPS202] and [SP800-185].

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 RFC 2119 [RFC2119].

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.

xy** denotes the exponentiation of x by y.

x mod y denotes reminder 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). This document focuses on only two instances, namely, TurboSHAKE128 and TurboSHAKE256, although the original definition includes a wider range of instances parameterized by their capacity [TURBOSHAKE].

An instance of TurboSHAKE takes as 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, the requested number of output bytes.

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

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.

* Non-multiple of 168-bytes-length M' are padded with zeroes to the next multiple of 168 bytes while M' with length multiple of 168 bytes are kept as is. 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 output is formed by taking the first 168 bytes of the state, repeated as many times as necessary until outputByteLen bytes are obtained, interleaved with the application of KP to the state.

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

The definition of the TurboSHAKE functions equivalently implements the pad10*1 rule. 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 [K12] 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 TurboSHAKE128

3.1. Interface

KangarooTwelve is an eXtendable Output Function (XOF). It takes as 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...)

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.

3.2. Specification

On top of the sponge function TurboSHAKE128, KangarooTwelve 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 || C || 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 SHOULD 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| <= 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, KangarooTwelve output is retrieved:

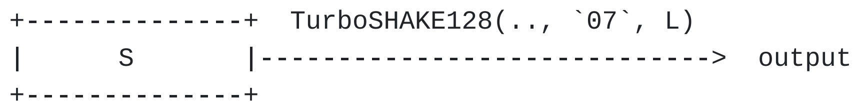
  *If |S| <= 8192 bytes, from TurboSHAKE128(FinalNode, `07`, L)

```
 KangarooTwelve( M, C, L ) = TurboSHAKE128( FinalNode, `07`, L )
```

  *Otherwise from TurboSHAKE128(FinalNode, `06`, L)

```
 KangarooTwelve( M, C, L ) = TurboSHAKE128( FinalNode, `06`, L )
```

The following figure illustrates the computation flow of KangarooTwelve for |S| <= 8192 bytes:



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

```
                               +--------------+
                               |     S_0      |
                               +--------------+
                                     ||
                               +--------------+
                               | `03`||`00`^7 |
                               +--------------+
                                     ||
+---------+  TSHK128(..,`0B`,32)  +--------------+
|   S_1   |---------------------->|     CV_1     |
+---------+                       +--------------+
                                     ||
+---------+  TSHK128(..,`0B`,32)  +--------------+
|   S_2   |---------------------->|     CV_2     |
+---------+                       +--------------+
                                     ||
              ..                     ..
                                     ||
+---------+  TSHK128(..,`0B`,32)  +--------------+
| S_(n-1) |---------------------->|   CV_(n-1)   |
+---------+                       +--------------+
                                     ||
                               +--------------+
                               |  l_e( n-1 )  |
                               +--------------+
                                     ||
                               +--------------+
                               |   `FF FF`    |
                               +--------------+
                                      | TSHK128(.., `06`, L)
                                      +-------------------->  output
```

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) || .. || x_0 || n where

```
x = sum of 256**i * x_i for i from 0 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) || .. || 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
```

4. Message authentication codes

Implementing a MAC with KangarooTwelve SHOULD use a HASH-then-MAC construction. This document recommends a method called HopMAC, defined as follows:

```
HopMAC(Key, M, C, L) = K12(Key, K12(M, C, 32), 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.

In any case, KangarooTwelve 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 KangarooTwelve with the key as the customization string, i.e., MAC = K12(M, Key, L).

5. Test vectors

Test vectors are based on the repetition of the pattern `00 01 .. FA` with a specific length. ptn(n) defines a string by repeating the pattern `00 01 .. 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=`07`, 32):
  `5A 22 3A D3 0B 3B 8C 66 A2 43 04 8C FC ED 43 0F
   54 E7 52 92 87 D1 51 50 B9 73 13 3A DF AC 6A 2F`

TurboSHAKE128(M=`00`^0, D=`07`, 64):
  `5A 22 3A D3 0B 3B 8C 66 A2 43 04 8C FC ED 43 0F
   54 E7 52 92 87 D1 51 50 B9 73 13 3A DF AC 6A 2F
   FE 27 08 E7 30 61 E0 9A 40 00 16 8B A9 C8 CA 18
   13 19 8F 7B BE D4 98 4B 41 85 F2 C2 58 0E E6 23`

TurboSHAKE128(M=`00`^0, D=`07`, 10032), last 32 bytes:
  `75 93 A2 80 20 A3 C4 AE 0D 60 5F D6 1F 5E B5 6E
   CC D2 7C C3 D1 2F F0 9F 78 36 97 72 A4 60 C5 5D`

TurboSHAKE128(M=ptn(1 bytes), D=`07`, 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`

TurboSHAKE128(M=ptn(17 bytes), D=`07`, 32):
  `AC BD 4A A5 75 07 04 3B CE E5 5A D3 F4 85 04 D8
   15 E7 07 FE 82 EE 3D AD 6D 58 52 C8 92 0B 90 5E`

TurboSHAKE128(M=ptn(17**2 bytes), D=`07`, 32):
  `7A 4D E8 B1 D9 27 A6 82 B9 29 61 01 03 F0 E9 64
   55 9B D7 45 42 CF AD 74 0E E3 D9 B0 36 46 9E 0A`

TurboSHAKE128(M=ptn(17**3 bytes), D=`07`, 32):
  `74 52 ED 0E D8 60 AA 8F E8 E7 96 99 EC E3 24 F8
   D9 32 71 46 36 10 DA 76 80 1E BC EE 4F CA FE 42`

TurboSHAKE128(M=ptn(17**4 bytes), D=`07`, 32):
  `CA 5F 1F 3E EA C9 92 CD C2 AB EB CA 0E 21 67 65
   DB F7 79 C3 C1 09 46 05 5A 94 AB 32 72 57 35 22`

TurboSHAKE128(M=ptn(17**5 bytes), D=`07`, 32):
  `E9 88 19 3F B9 11 9F 11 CD 34 46 79 14 E2 A2 6D
   A9 BD F9 6C 8B EF 07 6A EE AD 1A 89 7B 86 63 83`

TurboSHAKE128(M=ptn(17**6 bytes), D=`07`, 32):
  `9C 0F FB 98 7E EE ED AD FA 55 94 89 87 75 6D 09
   0B 67 CC B6 12 36 E3 06 AC 8A 24 DE 1D 0A F7 74`

TurboSHAKE128(M=`00`^0, D=`0B`, 32):
  `8B 03 5A B8 F8 EA 7B 41 02 17 16 74 58 33 2E 46
   F5 4B E4 FF 83 54 BA F3 68 71 04 A6 D2 4B 0E AB`

TurboSHAKE128(M=`00`^0, D=`06`, 32):
  `C7 90 29 30 6B FA 2F 17 83 6A 3D 65 16 D5 56 63
   40 FE A6 EB 1A 11 39 AD 90 0B 41 24 3C 49 4B 37`
```

```
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=`06`, 32):
  `3D 03 98 8B B5 9E 68 18 51 A1 92 F4 29 AE 03 98
   8E 8F 44 4B C0 60 36 A3 F1 A7 D2 CC D7 58 D1 74`

TurboSHAKE128(M=`FF FF FF FF FF FF FF`, D=`06`, 32):
  `05 D9 AE 67 3D 5F 0E 48 BB 2B 57 E8 80 21 A1 A8
   3D 70 BA 85 92 3A A0 4C 12 E8 F6 5B A1 F9 45 95`
```

```
TurboSHAKE256(M=`00`^0, D=`07`, 64):
  `4A 55 5B 06 EC F8 F1 53 8C CF 5C 95 15 D0 D0 49
   70 18 15 63 A6 23 81 C7 F0 C8 07 A6 D1 BD 9E 81
   97 80 4B FD E2 42 8B F7 29 61 EB 52 B4 18 9C 39
   1C EF 6F EE 66 3A 3C 1C E7 8B 88 25 5B C1 AC C3`

TurboSHAKE256(M=`00`^0, D=`07`, 10032), last 32 bytes:
  `40 22 1A D7 34 F3 ED C1 B1 06 BA D5 0A 72 94 93
   15 B3 52 BA 39 AD 98 B5 B3 C2 30 11 63 AD AA D0`

TurboSHAKE256(M=ptn(17 bytes), D=`07`, 64):
  `66 D3 78 DF E4 E9 02 AC 4E B7 8F 7C 2E 5A 14 F0
   2B C1 C8 49 E6 21 BA E6 65 79 6F B3 34 6E 6C 79
   75 70 5B B9 3C 00 F3 CA 8F 83 BC A4 79 F0 69 77
   AB 3A 60 F3 97 96 B1 36 53 8A AA E8 BC AC 85 44`

TurboSHAKE256(M=ptn(17**2 bytes), D=`07`, 64):
  `C5 21 74 AB F2 82 95 E1 5D FB 37 B9 46 AC 36 BD
   3A 6B CC 98 C0 74 FC 25 19 9E 05 30 42 5C C5 ED
   D4 DF D4 3D C3 E7 E6 49 1A 13 17 98 30 C3 C7 50
   C9 23 7E 83 FD 9A 3F EC 46 03 FF 57 E4 22 2E F2`

TurboSHAKE256(M=ptn(17**3 bytes), D=`07`, 64):
  `62 A5 A0 BF F0 64 26 D7 1A 7A 3E 9E 3F 2F D6 E2
   52 FF 3F C1 88 A6 A5 36 EC A4 5A 49 A3 43 7C B3
   BC 3A 0F 81 49 C8 50 E6 E7 F4 74 7A 70 62 7F D2
   30 30 41 C6 C3 36 30 F9 43 AD 92 F8 E1 FF 43 90`

TurboSHAKE256(M=ptn(17**4 bytes), D=`07`, 64):
  `52 3C 06 47 18 2D 89 41 F0 DD 5C 5C 0A B6 2D 4F
   C2 95 61 61 53 96 BB 5B 9A 9D EB 02 2B 80 C5 BF
   2D 83 A3 BB 36 FF C0 4F AC 58 CF 11 49 C6 6D EC
   4A 59 52 6E 51 F2 95 96 D8 24 42 1A 4B 84 B4 4D`

TurboSHAKE256(M=ptn(17**5 bytes), D=`07`, 64):
  `D1 14 A1 C1 A2 08 FF 05 FD 49 D0 9E E0 35 46 5D
   86 54 7E BA D8 E9 AF 4F 8E 87 53 70 57 3D 6B 7B
   B2 0A B9 60 63 5A B5 74 E2 21 95 EF 9D 17 1C 9A
   28 01 04 4B 6E 2E DF 27 2E 23 02 55 4B 3A 77 C9`

TurboSHAKE256(M=ptn(17**6 bytes), D=`07`, 64):
  `1E 51 34 95 D6 16 98 75 B5 94 53 A5 94 E0 8A E2
   71 CA 20 E0 56 43 C8 8A 98 7B 5B 6A B4 23 ED E7
   24 0F 34 F2 B3 35 FA 94 BC 4B 0D 70 E3 1F B6 33
   B0 79 84 43 31 FE A4 2A 9C 4D 79 BB 8C 5F 9E 73`

TurboSHAKE256(M=`00`^0, D=`0B`, 64):
  `C7 49 F7 FB 23 64 4A 02 1D 35 65 3D 1B FD F7 47
   CE CE 5F 97 39 F9 A3 44 AD 16 9F 10 90 6C 68 17
   C8 EE 12 78 4E 42 FF 57 81 4E FC 1C 89 87 89 D5
```

```
   E4 15 DB 49 05 2E A4 3A 09 90 1D 7A 82 A2 14 5C`

TurboSHAKE256(M=`00`^0, D=`06`, 64):
  `FF 23 DC CD 62 16 8F 5A 44 46 52 49 A8 6D C1 0E
   8A AB 4B D2 6A 22 DE BF 23 48 02 0A 83 1C DB E1
   2C DD 36 A7 DD D3 1E 71 C0 1F 7C 97 A0 D4 C3 A0
   CC 1B 21 21 E6 B7 CE AB 38 87 A4 C9 A5 AF 8B 03`

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=`06`, 64):
  `E5 53 8C DD 28 30 2A 2E 81 E4 1F 65 FD 2A 40 52
   01 4D 0C D4 63 DF 67 1D 1E 51 0A 9D 95 C3 7D 71
   35 EF 27 28 43 0A 9E 31 70 04 F8 36 C9 A2 38 EF
   35 37 02 80 D0 3D CE 7F 06 12 F0 31 5B 3C BF 63`

TurboSHAKE256(M=`FF FF FF FF FF FF FF`, D=`06`, 64):
  `B3 8B 8C 15 F4 A6 E8 0C D3 EC 64 5F 99 9F 64 98
   AA D7 A5 9A 48 9C 1D EE 29 70 8B 4F 8A 59 E1 24
   99 A9 6F 89 37 22 56 FE 52 2B 1B 97 47 2A DD 73
   69 15 BD 4D F9 3B 21 FF E5 97 21 7E B3 C2 C6 D9`
```

```
KangarooTwelve(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`

KangarooTwelve(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`

KangarooTwelve(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`

KangarooTwelve(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`

KangarooTwelve(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`

KangarooTwelve(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`

KangarooTwelve(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`

KangarooTwelve(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`

KangarooTwelve(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`

KangarooTwelve(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`

KangarooTwelve(M=`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`

KangarooTwelve(M=`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`
```

```
KangarooTwelve(M=`FF FF FF`, C=ptn(41**2), 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`

KangarooTwelve(M=`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`

KangarooTwelve(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`

KangarooTwelve(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`

KangarooTwelve(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`

KangarooTwelve(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`
```

6. IANA Considerations

None.

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. It relies on the cryptanalysis of Keccak and provides with the same security strength as their respective SHAKE functions.

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

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

* The inner function TurboSHAKE128. This layer relies on cryptanalysis. The TurboSHAKE128 function is exactly

Keccak[r=1344, c=256] (as in SHAKE128) reduced to 12 rounds. Any reduced-round cryptanalysis on Keccak is also a reduced-round cryptanalysis of 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 [K12].

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-bit 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, KangarooTwelve achieves NIST's post-quantum security level 2 [NISTPQ].

As a XOF, KangarooTwelve, TurboSHAKE128 or TurboSHAKE256 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).

Lastly, as KangarooTwelve uses TurboSHAKE128 with three values for D, namely 0x06, 0x07, and 0x0B. Protocols that use both KangarooTwelve and TurboSHAKE128, 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>.

[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.

[K12] 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.

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 [K12]

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 x from 0 to 4
    for y 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. KangarooTwelve

```
KangarooTwelve(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
```

Authors' Addresses

Benoît Viguier
ABN AMRO Bank
Groenelaan 2
Amstelveen

Email: cs.ru.nl@viguier.nl

David Wong (editor)
O(1) Labs

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