Workgroup: Crypto Forum
Internet-Draft:
draft-irtf-cfrg-kangarootwelve-08
Published: 19 August 2022
Intended Status: Informational
Expires: 20 February 2023
Authors: B. Viguier
D. Wong, Ed.
G. Van Assche, Ed.
ABN AMRO Bank $0(1)$ Labs
STMicroelectronics
Q. Dang, Ed. J. Daemen, Ed.
NIST Radboud University
KangarooTwelve

Abstract

This document defines the KangarooTwelve extendable Output Function (XOF), a hash function with output of arbitrary length. It provides an efficient and secure hashing primitive, which is able to exploit the parallelism of the implementation in a scalable way. It uses tree hashing over a round-reduced version of SHAKE128 as underlying primitive.

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 InternetDrafts 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 20 February 2023.

## Copyright Notice

Copyright (c) 2022 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. Specifications
2.1. Inner function $F$
2.2. Tree hashing over $F$
2.3. length_encode( $x$ )
3. Test vectors
4. IANA Considerations
5. Security Considerations
6. References
6.1. Normative References
6.2. Informative References

Appendix A. Pseudocode
A.1. Keccak-p[1600,n_r=12]
A.2. KangarooTwelve

Authors' Addresses

## 1. Introduction

This document defines the KangarooTwelve eXtendable Output Function (XOF) [K12], i.e. a generalization of a hash function that can return an output of arbitrary length. KangarooTwelve is based on a Keccak-p permutation specified in [FIPS202] and has a higher speed than SHAKE and SHA-3.

The SHA-3 functions process data in a serial manner and are unable to optimally exploit parallelism available in modern CPU architectures. Similar to ParallelHash [SP800-185], KangarooTwelve splits the input message into fragments to exploit available parallelism. It then applies an inner hash function $F$ on each of them separately before applying $F$ again on the concatenation of the digests. It makes use of Sakura coding for ensuring soundness of the tree hashing mode [SAKURA]. The inner hash function $F$ is a sponge function and uses a round-reduced version of the permutation Keccakf used in SHA-3, making it faster than ParallelHash. Its security builds up on the scrutiny that Keccak has received since its publication [KECCAK_CRYPTANALYSIS].

With respect to [FIPS202] and [SP800-185] functions, KangarooTwelve features the following advantages:
*Unlike SHA3-224, SHA3-256, SHA3-384, SHA3-512, KangarooTwelve has 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 splits the input message into fragments to exploit available parallelism.
*Unlike ParallelHash, KangarooTwelve does not have overhead when processing short messages.
*The Keccak-f permutation in KangarooTwelve has half the number of rounds of the one used in SHA3, making it faster than any function defined in [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, © 0 B 80`. |s| denotes the length of a byte string `s`. For example, |`FF FF`| \(=2\). \(` 00 `\wedge b\) denotes a byte string consisting of the concatenation of \(b\) bytes`00`. For example, `00`^7 = `00 $000000000000 `$.
`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`. \(\mathbf{s}[\mathrm{n}:]\) denotes the selection of bytes from n to the end of a string s . For example, for \(\mathrm{s}=\) `A5 C6 D7`, \(\mathrm{s}[0:]=\) 'A5 C6 D7` and $\mathrm{s}[2:]$ $={ }^{`} \mathrm{D} 7{ }^{\circ}$.

In the following, $x$ and $y$ are byte strings of equal length:
$\mathbf{x}^{\wedge}=\mathbf{y}$ denotes $x$ takes the value $x$ XOR $y$.
$x \& y$ denotes $x$ AND $y$.

In the following, $x$ and $y$ are integers:
$\mathbf{x + = y}$ denotes $x$ takes the value $x+y$.
$\mathbf{x - = y}$ denotes $x$ takes the value $x-y$.
$\mathbf{x}^{* *} \mathbf{y}$ denotes the exponentiation of $x$ by $y$.

## 2. Specifications

KangarooTwelve is an eXtendable Output Function (XOF). It takes as input 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 input, C MUST be the empty string.

### 2.1. Inner function $F$

The inner function $F$ makes use of the permutation Keccakp[1600,n_r=12], i.e., a version of the permutation Keccak-f[1600] used in SHAKE and SHA-3 instances reduced to its last n_r=12 rounds and specified in FIPS 202, sections 3.3 and 3.4 [FIPS202]. KP denotes this permutation.
$F$ is a sponge function calling this permutation KP with a rate of 168 bytes or 1344 bits. It follows that $F$ has a capacity of 1600 $1344=256$ bits or 32 bytes.

The sponge function $F$ takes:
input byte-string of positive length, the input bytes and
outputByteLen positive integer, the length of the output in bytes

First non-multiple of 168-bytes-length inputs are padded with zeroes to the next multiple of 168 bytes while inputs multiple of 168 bytes are kept as is. Then a byte `80` is XORed to the last byte of the padded message and the resulting string is split into a sequence of 168-byte blocks.

Inputs of length 0 bytes do not happen as a result of the tree hashing mode defined in section 2.2 .

As defined by the sponge construction, the process operates on a state and consists of two phases: the absorbing phase that processes the input 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.

The definition of the function $F$ equivalently implements the pad10*1 rule. It assumes an at least one-byte-long input where the last byte is in the `01`-`7F` range, and this is the case in KangarooTwelve. 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].

A pseudocode version is available as follows:

```
F(input, outputByteLen):
    offset = 0
    state = `00`^200
    # === 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
    end
```


### 2.2. Tree hashing over $F$

On top of the sponge function F, KangarooTwelve uses a Sakuracompatible 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 2.3.

$$
S=M| | C| | l e n g t h \_e n c o d e(|C|)
$$

Then, split $S$ into $n$ chunks of 8192 bytes.

$$
\begin{aligned}
& S=S \_0| | \ldots| | S \_(n-1) \\
& \left|S_{-} 0\right|=\ldots=\left|S_{-}(n-2)\right|=8192 \text { bytes } \\
& \left|S_{-}(n-1)\right|<=8192 \text { bytes }
\end{aligned}
$$

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.

$$
\text { CV_i }=\text { F( S_i||`0B`, } 32 \text { ) }
$$

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 F( FinalNode||`07`, L )
KangarooTwelve( M, C, L ) = F( FinalNode||`07`, L )
*Otherwise from F( FinalNode||`06`, L )
KangarooTwelve( M, C, L ) = F( 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 length_encode( $x$ ) is abbreviated as l_e( x ):

||
+--------------
| `03`||`00`^7 |
II

||

||
II
+--------- $F(. .| | ` 0 B `$, 32$) \quad+------------+$
| S_(n-1) |---------------->| CV_(n-1) |
| |
+-------------+
| l_e( n-1 ) |
II


A pseudocode version is provided in Appendix A. 2.
The table below gathers the values of the domain separation bytes used by the tree hash mode:


## 2.3. 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

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

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

As example, length_encode(0) = `00`, length_encode(12) = `0C 01` and length_encode(65538) = `01 0002 03`

A pseudocode version is as follows.
length_encode(x):
S = `00`^0
while $x>0$
$\mathrm{S}=\mathrm{x} \bmod 256 \| \mathrm{S}$
x = x / 256
$S=S| | l e n g t h(S)$
return S
end
3. 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:
$p \operatorname{tn}(17)=$
`00 010203040506070809 0A 0B 0C 0D 0E 0F 10`

Pattern for a length of $17 * * 2$ bytes:
ptn(17**2) =
-00 01020304050607080904 0B 0C 0D 0E 0F
 20212223242526272829 2A 2B 2C 2D 2E 2F
 40414243444546474849 4A 4B 4C 4D 4E 4F 50515253545556575859 5A 5B 5C 5D 5E 5F 60616263646566676869 6A 6B 6C 6D 6E 6F 70717273747576777879 7A 7B 7C 7D 7E 7F $808182838485868788898 A 8 B 8 C 8 D 8 E 8 F$ 90919293949596979899 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
00010203040506070809 0A 0B 0C 0D 0E 0F
 202122232425 「

```
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 563642 F7 22 8C 8468 4C 898405 D3 A8
34799158 C0 79 B1 288027 7A 1D 28 E2 FF 6D
KangarooTwelve(M=ptn(1 bytes), C=`00`^0, 32):
2B DA 9245 0E 8B 14 7F 8A 7C B6 29 E7 84 A0 58
EF CA 7C F7 D8 21 8E 02 D3 45 DF AA 6524 4A 1F
KangarooTwelve(M=ptn(17 bytes), C=`00`^0, 32):
6B F7 5F A2 239198 DB 4772 E3 6478 F8 E1 9B
0F 371205 F6 A9 A9 3A 27 3F 51 DF 37122888
KangarooTwelve(M=ptn(17**2 bytes), C=`00`^0, 32):
0C 31 5E BC DE DB F6 1426 DE 7D CF 8F B7 25 D1
E7 4675 D7 F5 32 7A 5067 F3 67 B1 08 EC B6 7C
KangarooTwelve(M=ptn(17**3 bytes), C=`00`^0, 32):
CB 55 2E 2E C7 7D 991070 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):
8701045 E 22205345 FF 4 D DA 0555 5C BB 5C 3A F1 A7 71 C2 B8 9B AE F3 7D B4 3D 9998 B9 FE
KangarooTwelve(M=ptn(17**5 bytes), C=`00`^0, 32):
84 4D 610933 B1 B9 96 3C BD EB 5A E3 B6 B0 5C
C7 CB D6 7C EE DF 88 3E B6 78 A0 A8 E0 371682
KangarooTwelve(M=ptn(17**6 bytes), C=`00`^0, 32):
` 3 C 390782 A8 A4 E8 9F A6 36 7F 72 FE AA F1 32     55 C8 D9 587848 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 246188 BF 7A F6 9A 1330 45 F4 6E E9 84 C5 6E 3C 3328 CA AF 1A A1 A5 83

KangarooTwelve(M=`FF`, C=ptn(41 bytes), 32): D8 48 C5 06 8C ED 73 6F 446215 9B 9867 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 712085 4C 2E 8C 6467 0A C0
1358 CF 4C 1B AF 8944 7A 724234 DC 7C ED 74

KangarooTwelve(M=`FF FF FF FF FF FF FF`, C=ptn(41**3 bytes), 32): 75 D2 F8 6A 2E 64456672 6B 4F BC FC 5657 B9 DB CF 07 0C 7B 0D CA 0645 0A B2 91 D7 44 3B CF

## 4. IANA Considerations

None.

## 5. Security Considerations

This document is meant to serve as a stable reference and an implementation guide for the KangarooTwelve eXtendable Output Function. It relies on the cryptanalysis of Keccak and provides with the same security strength as SHAKE128, i.e., 128 bits of security against all attacks.

To be more precise, KangarooTwelve is made of two layers:
*The inner function $F$. This layer relies on cryptanalysis. KangarooTwelve's F 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 KangarooTwelve's F (provided the number of rounds attacked is not higher than 12).
*The tree hashing over $F$. This layer is a mode on top of $F$ 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 128bit security. So for 128-bit (second) preimage security the output should be at least 128 bits, for 128 -bit of security against multitarget preimage attacks with T targets the output should be at least $128+l^{\prime} \mathrm{g}_{2} 2(\mathrm{~T})$ 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].

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) .

## 6. References

### 6.1. 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.

### 6.2. Informative References

[K12] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G., and R. Van Keer, "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, 2017.
[XKCP]
Bertoni, G., Daemen, J., Peeters, M., Van Assche, G., and R. Van Keer, "eXtended Keccak Code Package", WWW https://github.com/XKCP/XKCP, September 2018.
[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 KangarooTwelve. A standalone Python version is 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. KangarooTwelve

```
KangarooTwelve(inputMessage, customString, outputByteLen):
```

S = inputMessage || customString
S = S || length_encode( |customString| )
if $|S|<=8192$
return $F(S$ || `07`, outputByteLen)
else
\# === Kangaroo hopping ===
FinalNode $=$ S[0:8192] || `03` || `00`^7
offset = 8192
numBlock $=0$
while offset < |S|
blockSize $=\min (|S|-$ offset, 8192)
$C V=F(S[$ offset : offset + blockSize] || `0B`, 32)
FinalNode = FinalNode || CV
numBlock += 1
offset += blockSize
FinalNode = FinalNode || length_encode( numBlock ) || `FF FF`
return F(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

