Internet Research Task Force (IRTF)   B. Viguier
Internet-Draft   Radboud University
Intended status: Informational   March 19, 2018
Expires: September 20, 2018

KangarooTwelve
draft-viguier-kangarootwelve-02

Abstract

This document defines the KangarooTwelve eXtendable Output Function (XOF), a hash function with arbitrary output 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.

Table of Contents

1. Introduction

This document defines the KangarooTwelve eXtendable Output Function (XOF) [K12], i.e. a generalization of a hash function that can return arbitrary output 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 in 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 Keccak-f 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 in 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, `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 to m exclusive of a string s. 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 x multiplied by itself y times.

2. Specifications

KangarooTwelve is an eXtendable Output Function (XOF). It takes as an input a couple of byte-strings (M, C) and a positive integer L where

M byte-string, is the Message and

C byte-string, is a OPTIONAL Customization string and

L positive integer, the number of output bytes requested.

The Customization string MAY serves 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 does that 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 Keccak-p[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, the input bytes and

outputByteLen positive integer, the Length of the output in bytes

First the message is padded with zeroes to the closest multiple of 168 bytes. 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.

As defined by the sponge construction, the process operates on a state and consists of two phases.

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.

This definition of the sponge construction assumes a at least one-byte-long input where the last byte is in the `01`-`7F` range. This is the case in KangarooTwelve.

A pseudo-code version is available as follows:

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

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

  # === Absorb last block and treatment of padding ===
  LastBlockLength = |input| - offset
  state ^= inputBytes[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 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 2.3.

```
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-bytes Chaining Values CV_1 .. CV_n-1. This computation SHOULD exploit the parallelism available on the platform in order to be optimally efficient.

```
CV_i    = F( S_i||`0B`, 32 )
```

Compute the final node: FinalNode.

o If |S| <= 8192 bytes, FinalNode = S

o Otherwise compute FinalNode as follow:

```
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:

o If |S| <= 8192 bytes, from F(FinalNode||`07`, L)

```
KangarooTwelve( M, C, L ) = F( FinalNode||`07`, L )
```

o 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:

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

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

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

We provide a pseudo code version in Appendix A.2.

In the table below are gathered the values of the domain separation bytes used by the tree hash mode:

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

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 = sum from i=0..n-1 of 256**i * x_i

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 pseudo code version is as follow.

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

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

  S = S || length(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:
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`

 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`
```

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 [KECCAK_CRYPTANALYSIS] and provides with the same security strength as SHAKE128, i.e., 128 bits of security against all attacks

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.

6. References

6.1. Normative References

[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.

[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>.

[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 http://eprint.iacr.org/2016/770.pdf, August 2016.

[KCP] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G., and R. Van Keer, "Keccak Code Package", WWW https://github.com/KeccakTeam/KeccakCodePackage, December 2017.

[KECCAK_CRYPTANALYSIS] Keccak Team, "Summary of Third-party cryptanalysis of Keccak", WWW https://www.keccak.team/third_party.html, 2017.

[SAKURA] Bertoni, G., Daemen, J., Peeters, M., and G. Van Assche, "Sakura: a flexible coding for tree hashing", WWW http://eprint.iacr.org/2013/231.pdf, April 2013.

Appendix A. Pseudo code

The sub-sections of this appendix contain pseudo code definitions of KangarooTwelve. A standalone Python version is also available in the Keccak Code Package [KCP] 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)
          CV = 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
```

Author's Address

Benoit Viguier
Radboud University
Toernooiveld 212
Nijmegen
The Netherlands

EMail: b.viguier@cs.ru.nl