NIST Special Publication 800-185

SHA-3 Derived Functions:

cSHAKE, KMAC, TupleHash and ParallelHash

John Kelsey
Shu-jen Chang
Ray Perlner

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-185

C O M P U T E R   S E C U R I T Y

National Institute of
Standards and Technology
U.S. Department of Commerce

NIST Special Publication 800-185

SHA-3 Derived Functions:

cSHAKE, KMAC, TupleHash and ParallelHash

John Kelsey
Shu-jen Chang
Ray Perlner
Computer Security Division
Information Technology Laboratory

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-185

December 2016

U.S. Department of Commerce
Penny Pritzker, Secretary

National Institute of Standards and Technology
Willie May, Under Secretary of Commerce for Standards and Technology and Director

Authority

National Institute of Standards and Technology Special Publication 800-185
Natl. Inst. Stand. Technol. Spec. Publ. 800-185, 32 pages (December 2016)
CODEN: NSPUE2

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-185

Comments on this publication may be submitted to:

National Institute of Standards and Technology
Attn: Computer Security Division, Information Technology Laboratory
100 Bureau Drive (Mail Stop 8930) Gaithersburg, MD 20899-8930
Email: SP800-185@nist.gov

All comments are subject to release under the Freedom of Information Act (FOIA).

Abstract

This Recommendation specifies four types of SHA-3-derived functions: cSHAKE, KMAC, TupleHash, and ParallelHash, each defined for a 128- and 256-bit security strength. cSHAKE is a customizable variant of the SHAKE function, as defined in Federal Information Processing Standard (FIPS) 202. KMAC (for KECCAK Message Authentication Code) is a variable-length message authentication code algorithm based on KECCAK; it can also be used as a pseudorandom function. TupleHash is a variable-length hash function designed to hash tuples of input strings without trivial collisions. ParallelHash is a variable-length hash function that can hash very long messages in parallel.

Keywords

authentication; cryptography; cSHAKE; customizable SHAKE function; hash function; information security; integrity; KECCAK; KMAC; message authentication code; parallel hashing; ParallelHash; PRF; pseudorandom function; SHA-3; SHAKE; tuple hashing; TupleHash.

Acknowledgements

The authors thank the KECCAK team members: Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche, for their feedback. The authors also thank their colleagues for reviewing drafts of this document and contributing to its development.

Table of Contents

List of Appendices

List of Tables

1 Introduction

Federal Information Processing Standard (FIPS) 202, *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions* [1], defines four fixed-length hash functions (SHA3-224, SHA3-256, SHA3-384, and SHA3-512), and two eXtendable Output Functions (XOFs), SHAKE128 and SHAKE256. These SHAKE (Secure Hash Algorithm KECCAK) functions are a new kind of cryptographic primitive; unlike earlier hash functions, they are named for their expected security strength.

FIPS 202 also supports a flexible scheme for domain separation between different functions derived from KECCAK—the algorithm [2] on which the SHA-3 Standard is based. Domain separation ensures that different named functions (such as SHA3-512 and SHAKE128) will be unrelated. cSHAKE—the customizable version of SHAKE—extends this scheme to allow users to customize their use of the function, as described below.

Customization is analogous to strong typing in a programming language; such customization makes it extremely unlikely that computing one function with two different customization strings will yield the same answer. Thus, two cSHAKE computations with different customization strings (for example, a key fingerprint and an email signature) are unrelated: knowing one of these results will give an attacker no information about the other.

This Recommendation defines two cSHAKE variants, cSHAKE128 and cSHAKE256, in Sec. 3, based on the KECCAK[c] sponge function [3] defined in FIPS 202. It then defines three additional SHA-3-derived functions, in Secs. 4 through 6, that provide new functionality not directly available from the more basic functions. They are:

- KMAC128[1] and KMAC256, providing pseudorandom functions (PRFs) and keyed hash functions with variable-length outputs;
- TupleHash128 and TupleHash256, providing functions that hash tuples of input strings unambiguously[2]; and
- ParallelHash128 and ParallelHash256, providing efficient hash functions to hash long messages more quickly by taking advantage of parallelism in the processors.

All four functions defined in this Recommendation—cSHAKE, KMAC, TupleHash, and ParallelHash—have these properties in common:

- They are all derived from the functions specified in FIPS 202.
- All the functions except cSHAKE are defined in terms of cSHAKE.

[1] KMAC stands for KECCAK Message Authentication Code.

[2] TupleHash processes a tuple of one or more input strings, and incorporates the contents of all the strings, the number of strings, and the specific content of each string in the calculation of the resulting hash value. Thus, any change (such as moving bytes from one input string to an adjacent one, or removing an empty string from the input tuple) is extremely likely to lead to a different result.

- All support user-defined customization strings.
- All support variable-length outputs of any bit length. KMAC, TupleHash, and ParallelHash have the additional property that any change in the requested output length completely changes the function. Even with identical inputs otherwise, any of these functions, when called with different requested output lengths, will, in general, yield unrelated outputs.
- All support two security strengths: 128 and 256 bits.

These functions are detailed in the specific sections below. In addition, a method is specified in Appendix B to facilitate using these functions to produce output that is almost uniformly distributed on the integers {0, 1, 2, ..., R–1} for any positive integer R.

2 Glossary

In this document, bits are indicated in the `Courier New` font. Bytes are typically written as two-digit hexadecimal numbers from the ASCII characters 0 through 9 and A through F, preceded by the prefix “0x”. In binary representation, bytes are written with the low-order bit first, while in hexadecimal representation, bytes are written with the high-order digit first. E.g., 0x01 = `10000000` and 0x80 = `00000001`. These bit-ordering conventions follow the conventions established in Sec. B.1 of FIPS 202. Character strings appear in this document in double-quotes. Character strings are interpreted as bit strings whose length is a multiple of 8 bits, consisting of a 0 bit, followed by the 7-bit ASCII representation of each successive character.

2.1 Terms and Acronyms

Bit	A binary digit: `0` or `1`.
CMAC	Cipher-based Message Authentication Code.
cSHAKE	The customizable SHAKE function.
Domain Separation	For a function, a partitioning of the inputs to different application domains so that no input is assigned to more than one domain.
eXtendable-Output Function (XOF)	A function on bit strings in which the output can be extended to any desired length.
FIPS	Federal Information Processing Standard.
Hash Function	A function on bit strings in which the length of the output is fixed. The output often serves as a condensed representation of the input.
HMAC	Keyed-Hash Message Authentication Code.
KECCAK	The family of all sponge functions with a KECCAK-*f* permutation as the underlying function and multi-rate padding as the padding rule. KECCAK was originally specified in [2], and standardized in FIPS 202.
KMAC	KECCAK Message Authentication Code.
MAC	Message Authentication Code.
NIST	National Institute of Standards and Technology.
PRF	See *Pseudorandom Function*.

Pseudorandom Function (PRF)	A function that can be used to generate output from a random seed such that the output is computationally indistinguishable from truly random output.
Rate	In the sponge construction, the number of input bits processed per invocation of the underlying function.
SHA-3	Secure Hash Algorithm-3.
Sponge Construction	The method originally specified in [3] for defining a function from the following: 1) an underlying function on bit strings of a fixed length, 2) a padding rule, and 3) a rate. Both the input and the output of the resulting function are bit strings that can be arbitrarily long.
Sponge Function	A function that is defined according to the sponge construction, possibly specialized to a fixed output length.
String	A sequence of bits.
XOF	See *eXtendable-Output Function*.

2.2 Basic Operations

$\lceil x \rceil$	For a real number x, $\lceil x \rceil$ is the least integer that is not strictly less than x. For example, $\lceil 3.2 \rceil = 4$, $\lceil -3.2 \rceil = -3$, and $\lceil 6 \rceil = 6$.
$\mathtt{0}^s$	For a positive integer s, $\mathtt{0}^s$ is the string that consists of s consecutive 0 bits.
a mod b	The modulo operation of integers a and b. "a mod b" returns the remainder after dividing a by b.
$\text{enc}_8(i)$	For an integer i ranging from 0 to 255, $\text{enc}_8(i)$ is the byte encoding of i, with bit 0 being the low-order bit of the byte.
len(X)	For a bit string X, len(X) is the length of X in bits.
$X \parallel Y$	For strings X and Y, $X \parallel Y$ is the concatenation of X and Y. For example, $\mathtt{11001} \parallel \mathtt{010} = \mathtt{11001010}$.

2.3 Other Internal Functions

This section describes the string encoding, padding and substring functions used in the definition of the SHA-3-derived functions.

2.3.1 Integer to Byte String Encoding

Two internal functions, *left_encode* and *right_encode*, are defined to encode integers as byte strings. Both functions can encode integers up to an extremely large maximum, $2^{2040}–1$.

left_encode(x) encodes the integer x as a byte string in a way that can be unambiguously parsed from the beginning of the string by inserting the length of the byte string before the byte string representation of x.

right_encode(x) encodes the integer x as a byte string in a way that can be unambiguously parsed from the end of the string by inserting the length of the byte string after the byte string representation of x.

Using the function $enc_8()$ to encode the individual bytes, these two functions are defined as follows:

right_encode(*x*):
Validity Conditions: $0 \leq x < 2^{2040}$

```
1. Let $n$ be the smallest positive integer for which $2^{8n} > x$.
2. Let $x_1, x_2, \ldots, x_n$ be the base-256 encoding of $x$ satisfying:
       $x = \sum 2^{8(n-i)} x_i$, for $i = 1$ to $n$.
3. Let $O_i = \text{enc}_8(x_i)$, for $i = 1$ to $n$.
4. Let $O_{n+1} = \text{enc}_8(n)$.
5. Return $O = O_1 \| O_2 \| \ldots \| O_n \| O_{n+1}$.
```

left_encode(*x*):
Validity Conditions: $0 \leq x < 2^{2040}$

```
1. Let $n$ be the smallest positive integer for which $2^{8n} > x$.
2. Let $x_1, x_2, \ldots, x_n$ be the base-256 encoding of $x$ satisfying:
       $x = \sum 2^{8(n-i)} x_i$, for $i = 1$ to $n$.
3. Let $O_i = \text{enc}_8(x_i)$, for $i = 1$ to $n$.
4. Let $O_0 = \text{enc}_8(n)$.
5. Return $O = O_0 \| O_1 \| \ldots \| O_{n-1} \| O_n$.
```

As an example, right_encode(0) will yield `00000000 10000000`, and left_encode(0) will yield `10000000 00000000`.

2.3.2 String Encoding

The encode_string function is used to encode bit strings in a way that may be parsed unambiguously from the beginning of the string, *S*. The function is defined as follows:

encode_string(*S*):
Validity Conditions: $0 \leq len(S) < 2^{2040}$

```
1. Return left_encode(len($S$)) || $S$.
```

As an example, encode_string(*S*) where *S* is the empty string "" will yield `10000000 00000000`.

Note that if the bit string *S* is not byte-oriented (i.e., len(*S*) is not a multiple of 8), the bit string returned from encode_string(*S*) is also not byte-oriented. However, if len(*S*) is a multiple of 8, then the length of the output of encode_string(*S*) will also be a multiple of 8.

2.3.3 Padding

The bytepad(*X*, *w*) function prepends an encoding of the integer *w* to an input string *X*, then pads the result with zeros until it is a byte string whose length in bytes is a multiple of *w*. In general, bytepad is intended to be used on encoded strings—the byte string bytepad(encode_string(*S*), *w*) can be parsed unambiguously from its beginning, whereas bytepad does not provide unambiguous padding for all input strings.

The definition of bytepad() is as follows:

bytepad(*X*, *w*):
Validity Conditions: w > 0

```
1. $z$ = left_encode($w$) || $X$.
2. while len($z$) mod 8 ≠ 0:
       $z = z$ || 0
3. while (len($z$)/8) mod $w$ ≠ 0:
       z = z || 00000000
4. return $z$.
```

2.3.4 Substrings

Let parameters *a* and *b* be non-negative integers that denote a specific position in a bit string *X*. Informally, the substring(*X*, *a*, *b*) function returns a substring from the bit string *X* containing the values at bit positions *a*, *a*+1, ..., *b*−1, inclusive. More precisely, the substring function operates as defined below. Note that all bit positions in the input and output strings are indexed from zero. Thus, the first bit in a string is in position 0, and the last bit in an *n*-bit string is in position *n*−1.

substring(*X*, *a*, *b*):

```
1. If $a \geq b$ or $a \geq$ len($X$):
       return the empty string.
2. Else if $b \leq$ len($X$):
       return the bits of $X$ from position $a$ to position $b$−1, inclusive.
3. Else:
       return the bits of $X$ from position $a$ to position len($X$)−1, inclusive.
```

3 cSHAKE

3.1 Overview

The two variants of cSHAKE—cSHAKE128 and cSHAKE256—are defined in terms of the SHAKE and KECCAK[*c*] functions specified in FIPS 202. cSHAKE128 provides a 128-bit security strength, while cSHAKE256 provides a 256-bit security strength.

3.2 Parameters

Both cSHAKE functions take four parameters:

- *X* is the main input bit string. It may be of any length[3], including zero.
- *L* is an integer representing the requested output length[4] in bits.
- *N* is a function-name bit string, used by NIST to define functions based on cSHAKE. When no function other than cSHAKE is desired, *N* is set to the empty string.
- *S* is a customization bit string. The user selects this string to define a variant of the function. When no customization is desired, *S* is set to the empty string[5].

An implementation of cSHAKE may reasonably support only input strings and output lengths that are whole bytes; if so, a fractional-byte input string or a request for an output length that is not a multiple of 8 would result in an error.

When *N* and *S* are both empty strings, cSHAKE(*X*, *L*, *N*, *S*) is equivalent to SHAKE as defined in FIPS 202. Thus,

cSHAKE128(*X*, *L*, "", "") = SHAKE128(*X*, *L*) and
cSHAKE256(*X*, *L*, "", "") = SHAKE256(*X*, *L*).

cSHAKE is designed so that for any two instances:

cSHAKE(*X*1, *L*1, *N*1, *S*1) and
cSHAKE(*X*1, *L*1, *N*2, *S*2),

[3] When a string is specified to be "of any length" in this document, a theoretical limit (e.g., $2^{2040}–1$ bits) may apply. This limit is imposed by the integer encoding schemes left_encode and right_encode, defined in Sec. 2.3.1. Beyond this limit, the string cannot be encoded. In the remainder of this document, absurdly large limits like $2^{2040}–1$ bits will often be treated interchangeably with no limit at all.

[4] When the requested output length is zero, i.e., *L*=0, cSHAKE, KMAC, TupleHash, and ParallelHash return the empty string as the output.

[5] In computing languages that support default values for parameters, a natural way to implement this function would be to set the default values for *N* and *S* to empty strings.

unless *N*1 = *N*2 and *S*1 = *S*2, the two instances produce unrelated outputs. Note that this includes the case where *N*1 and *S*1 are empty strings. That is, cSHAKE with any customization is domain-separated from the ordinary SHAKE function specified in FIPS 202.

3.3 Definition

cSHAKE is defined in terms of SHAKE or KECCAK[*c*], as follows: it either returns the result of a call to SHAKE (if *N* and *S* are both empty strings), or returns the result of a call to KECCAK(*c*) with a padded encoding of *N* and *S* concatenated to the input string *X*.

cSHAKE128(*X, L, N, S*):
Validity Conditions: len(N)< 2^{2040} and *len(S)*< 2^{2040}

```
1. If N = "" and S = "":
        return SHAKE128(X, L);
2. Else:
        return KECCAK[256](bytepad(encode_string(N) || encode_string(S), 168) || X || 00, L).
```

cSHAKE256(*X, L, N, S*):
Validity Conditions: len(N)< 2^{2040} and *len(S)*< 2^{2040}

```
1. If N = "" and S = "":
        return SHAKE256(X, L);
2. Else:
        return KECCAK[512](bytepad(encode_string(N) || encode_string(S), 136) || X || 00, L).
```

Note that the numbers 168 and 136 are *rates* (in bytes) of the KECCAK[256] and KECCAK[512] sponge functions, respectively; the characters `00` in the `Courier New` font in these definitions specify two zero bits.

3.4 Using the Function-Name Input

The cSHAKE function includes an input string that may be used to provide a function name (*N*). This is intended for use by NIST in defining SHA-3-derived functions, and should only be set to values defined by NIST[6]. This parameter provides a level of domain separation by function name. Users of cSHAKE should not make up their own names—that kind of customization is the purpose of the customization string *S*, to be discussed in Sec. 3.5. Nonstandard values of *N* could cause interoperability problems with future NIST-defined functions.

[6] NIST will always make the function name *N* a byte-oriented value.

3.5 Using the Customization String

The cSHAKE function also includes an input string (*S*) to allow users to customize their use of the function. For example, someone using cSHAKE128 to compute a key fingerprint (the hash value for a public key) might use:

cSHAKE128(*public_key*, 256, "", "key fingerprint"),

where "key fingerprint" is a customization string *S*.

Later, the same user might decide to customize a different cSHAKE computation for signing an email:

cSHAKE128(*email_contents*, 256, "", "email signature"),

where "email signature" is the customization string *S*.

The customization string is intended to avoid a collision between these two cSHAKE values—it will be very difficult for an attacker to somehow force one computation (the email signature) to yield the same result as the other computation (the key fingerprint) if different values of *S* are used.

The customization string may be of any length less than 2^{2040}; however, implementations may restrict the length of *S* that they will accept.

4 KMAC

4.1 Overview

The KECCAK Message Authentication Code (KMAC) algorithm is a PRF and keyed hash function based on KECCAK. It provides variable-length output, and unlike SHAKE and cSHAKE, altering the requested output length generates a new, unrelated output. KMAC has two variants, KMAC128 and KMAC256, built from cSHAKE128 and cSHAKE256, respectively. The two variants differ somewhat in their technical security properties. Nonetheless, for most applications, both variants can support any security strength up to 256 bits of security, provided that a long enough key is used, as discussed in Sec. 8.4.1.

4.2 Parameters

Both KMAC functions take the following parameters:

- *K* is a key bit string of any length[7], including zero.
- *X* is the main input bit string. It may be of any length, including zero.
- *L* is an integer representing the requested output length[8] in bits.
- *S* is an optional customization bit string of any length, including zero. If no customization is desired, *S* is set to the empty string.

4.3 Definition

KMAC concatenates a padded version of the key *K* with the input *X* and an encoding of the requested output length *L*. The result is then passed to cSHAKE, along with the requested output length *L*, the name *N* ="KMAC" = `11010010 10110010 10000010 11000010`[9], and the optional customization string *S*.

KMAC128(*K, X, L, S*):
Validity Conditions: len(K) $< 2^{2040}$ and $0 \leq L < 2^{2040}$ and *len(S)* $< 2^{2040}$

1. *newX* = bytepad(encode_string(*K*), 168) || *X* || right_encode(*L*).
2. return cSHAKE128(*newX*, *L*, “KMAC”, *S*).

[7] Approved uses of KMAC require the length of *K* to be at least the required security strength, as discussed in Sec. 8.4.1.

[8] Note that there is a limit of $2^{2040}-1$ bits of output from this function unless the function is used as a XOF, as discussed in Sec. 4.3.1.

[9] Note that in binary representation, bytes are written with the low-order bit first in this document, as specified in Sec. 2.

KMAC256(*K*, *X*, *L*, *S*):
Validity Conditions: $len(K) < 2^{2040}$ and $0 \leq L < 2^{2040}$ and $len(S) < 2^{2040}$

```
1. newX = bytepad(encode_string(K), 136) || X || right_encode(L).
2. return cSHAKE256(newX, L, "KMAC", S).
```

Note that the numbers 168 and 136 are *rates* (in bytes) of the KECCAK[256] and KECCAK[512] sponge functions, respectively.

4.3.1 KMAC with Arbitrary-Length Output

Some applications of KMAC may not know the number of output bits they will need until after the outputs begin to be produced. For these applications, KMAC can also be used as a XOF (i.e., the output can be extended to any desired length), which mimics the behavior of cSHAKE.

When used as a XOF, KMAC is computed by setting the encoded output length to 0, as shown in right_encode(0) in Step 1 of the KMACXOF128(*K*, *X*, *L*, *S*) and KMACXOF256(*K*, *X*, *L*, *S*) pseudocodes below. Conceptually, KMAC in XOF mode produces an infinite-length output string, and the caller simply uses as many bits of the output string as are needed. Truncated outputs of KMAC in XOF mode can be computed by the function KMACXOF128(*K*, *X*, *L*, *S*) or KMACXOF256(*K*, *X*, *L*, *S*) given by the following pseudocode:

KMACXOF128(*K*, *X*, *L*, *S*):
Validity Conditions: $len(K) < 2^{2040}$ and $0 \leq L$ and $len(S) < 2^{2040}$

```
1. newX = bytepad(encode_string(K), 168) || X || right_encode(0).
2. return cSHAKE128(newX, L, "KMAC", S).
```

KMACXOF256(*K*, *X*, *L*, *S*):
Validity Conditions: $len(K) < 2^{2040}$ and $0 \leq L$ and $len(S) < 2^{2040}$

```
1. newX = bytepad(encode_string(K), 136) || X || right_encode(0).
2. return cSHAKE256(newX, L, "KMAC", S).
```

5 TupleHash

5.1 Overview

TupleHash is a SHA-3-derived hash function with variable-length output that is designed to simply hash a tuple of input strings, any or all of which may be empty strings, in an unambiguous way. Such a tuple may consist of any number of strings, including zero, and is represented as a sequence of strings or variables in parentheses like (“a”, “b”, “c”,...,“z”) in this document.

TupleHash is designed to provide a generic, misuse-resistant way to combine a sequence of strings for hashing such that, for example, a TupleHash computed on the tuple ("abc" ,"d") will produce a different hash value than a TupleHash computed on the tuple ("ab","cd"), even though all the remaining input parameters are kept the same, and the two resulting concatenated strings, without string encoding, are identical.

TupleHash supports two security strengths: 128 bits and 256 bits. Changing any input to the function, including the requested output length, will almost certainly change the final output.

5.2 Parameters

TupleHash takes the following parameters:

- X is a tuple of zero or more bit strings, any or all of which may be an empty string.
- L is an integer representing the requested output length in bits.
- S is an optional customization bit string of any length, including zero. If no customization is desired, S is set to the empty string.

5.3 Definition

TupleHash encodes the sequence of input strings in an unambiguous way, then encodes the requested output length at the end of the string, and passes the result into cSHAKE, along with the function name (N) of “TupleHash” = `00101010 10101110 00001110 00110110 10100110 00010010 10000110 11001110 00010110`, and the optional customization string S.

If X is a tuple of n bit strings, let $X[i]$ be the ith bit string, numbering from 0. The TupleHash functions are defined in pseudocode as follows:

```
TupleHash128(X, L, S):
Validity Conditions: $0 \leq L < 2^{2040}$ and $len(S) < 2^{2040}$

1. $z$ = "".
2. $n$ = the number of input strings in the tuple $X$.
3. for $i$ = 1 to $n$:
       $z = z \,||\,$ encode_string($X[i]$).
4. $newX = z \,||\,$ right_encode($L$).
```

```
5. return cSHAKE128(newX, L, “TupleHash”, S).
```

TupleHash256(*X*, *L*, *S*):
Validity Conditions: $0 \le L < 2^{2040}$ and $len(S) < 2^{2040}$

```
1. z = "".
2. n = the number of input strings in the tuple X.
3. for i = 1 to n:
       z = z || encode_string(X[i]).
4. newX = z || right_encode(L).
5. return cSHAKE256(newX, L, “TupleHash”, S).
```

5.3.1 TupleHash with Arbitrary-Length Output

Some applications of TupleHash may not know the number of output bits they will need until after the outputs begin to be produced. For these applications, TupleHash can also be used as a XOF (i.e., the output can be extended to any desired length), which mimics the behavior of cSHAKE.

When used as a XOF, TupleHash is computed by setting the encoded output length to 0, as shown in right_encode(0) in Step 1 of the TupleHashXOF128(*X*, *L*, *S*) and TupleHashXOF256(*X*, *L*, *S*) pseudocodes below. Conceptually, TupleHash in XOF mode produces an infinite-length output string, and the caller simply uses as many bits of the output string as are needed. Truncated outputs of TupleHash in XOF mode can be computed by the function TupleHashXOF128(*X*, *L*, *S*) or TupleHashXOF256(*X*, *L*, *S*) given by the following pseudocode:

TupleHashXOF128(*X*, *L*, *S*):
Validity Conditions: $0 \le L$ and $len(S) < 2^{2040}$

```
1. z = "".
2. n = the number of input strings in the tuple X.
3. for i = 1 to n:
       z = z || encode_string(X[i]).
4. newX = z || right_encode(0).
5. return cSHAKE128(newX, L, “TupleHash”, S).
```

TupleHashXOF256(*X*, *L*, *S*):
Validity Conditions: $0 \le L$ and $len(S) < 2^{2040}$

```
1. z = "".
2. n = the number of input strings in the tuple X.
3. for i = 1 to n:
       z = z || encode_string(X[i]).
4. newX = z || right_encode(0).
5. return cSHAKE256(newX, L, “TupleHash”, S).
```

6 ParallelHash

6.1 Overview

The purpose of ParallelHash[10] is to support the efficient hashing of very long strings, by taking advantage of the parallelism available in modern processors. ParallelHash supports the 128- and 256-bit security strengths, and also provides variable-length output. Changing any input parameter to ParallelHash, even the requested output length, will result in unrelated output. Like the other functions defined in this document, ParallelHash also supports user-selected customization strings.

6.2 Parameters

ParallelHash takes the following parameters:

- X is the main input bit string. It may be of any length[11], including zero.
- B is the block size in bytes for parallel hashing. It may be any integer such that $0 < B < 2^{2040}$.
- L is an integer representing the requested output length in bits.
- S is an optional customization bit string of any length, including zero. If no customization is desired, S is set to the empty string.

6.3 Definition

ParallelHash divides the input bit string X into a sequence of contiguous, non-overlapping blocks, each of length B bytes, and then computes the hash value for each block separately. Finally, these hash values are combined and passed to cSHAKE along with the function name (N) of "ParallelHash" = `00001010 10000110 01001110 10000110 00110110 00110110 10100110 00110110 00010010 10000110 11001110 00010110`, the optional customization string S, and some encoded integer values (as shown below in the pseudocode), to generate the final hash value of the function.

The ParallelHash functions are defined in pseudocode as follows:

ParallelHash128(*X, B, L, S*):
Validity Conditions: $0 < B < 2^{2040}$ and $\lceil len(X)/B \rceil < 2^{2040}$ and
$0 \le L < 2^{2040}$ and $len(S) < 2^{2040}$

```
1.  $n = \lceil (\mathrm{len}(X)/8) / B \rceil$.
```

[10] A *generic parallel hash* mode for other NIST-approved hash functions may be developed in the future. The function here (i.e., ParallelHash) is specifically based on cSHAKE, and thus, on KECCAK.

[11] Where $\lceil len(X)/B \rceil < 2^{2040}$ and B is the block size in bytes as defined in Sec. 6.2. As specified in Footnote 2, NIST will treat such absurdly large limit as interchangeable with having no limit at all.

```
2. $z$ = left_encode($B$).
3. for $i$ = 0 to $n-1$:
       $z = z$ || cSHAKE128(substring($X$, $i*B*8$, $(i+1)*B*8$), 256, "", "").
4. $z = z$ || right_encode($n$) || right_encode($L$).
5. $newX = z$.
6. return cSHAKE128($newX$, $L$, "ParallelHash", $S$).
```

ParallelHash256(*X, B, L, S*):
Validity Conditions: $0 < B < 2^{2040}$ and $\lceil len(X)/B \rceil < 2^{2040}$ and
$0 \le L < 2^{2040}$ and $len(S) < 2^{2040}$

```
1. $n = \lceil (\text{len}(X)/8) / B \rceil$.
2. $z$ = left_encode($B$).
3. for $i$ = 0 to $n-1$:
       $z = z$ || cSHAKE256(substring($X$, $i*B*8$, $(i+1)*B*8$), 512, "", "").
4. $z = z$ || right_encode($n$) || right_encode($L$).
5. $newX = z$.
6. return cSHAKE256($newX$, $L$, "ParallelHash", $S$).
```

6.3.1 ParallelHash with Arbitrary-Length Output

Some applications of ParallelHash may not know the number of output bits they will need until after the outputs begin to be produced. For these applications, ParallelHash can also be used as a XOF (i.e., the output can be extended to any desired length), which mimics the behavior of cSHAKE.

When used as a XOF, ParallelHash is computed by setting the encoded output length to 0, as shown in right_encode(0) in Step 1 of the ParallelHashXOF128(*X*, *B*, *L*, *S*) and ParallelHashXOF256(*X*, *B*, *L*, *S*) pseudocodes below. Conceptually, ParallelHash in XOF mode produces an infinite-length output string, and the caller simply uses as many bits of the output string as are needed. Truncated outputs of ParallelHash in XOF mode can be computed by the function ParallelHashXOF128(*X*, *B*, *L*, *S*) or ParallelHashXOF256(*X*, *B*, *L*, *S*) given by the following pseudocode:

ParallelHashXOF128(*X, B, L, S*):
Validity Conditions: $0 < B < 2^{2040}$ and $\lceil len(X)/B \rceil < 2^{2040}$ and
$0 \le L$ and $len(S) < 2^{2040}$

```
1. $n = \lceil (\text{len}(X)/8) / B \rceil$.
2. $z$ = left_encode($B$).
3. for $i$ = 0 to $n-1$:
       $z = z$ || cSHAKE128(substring($X$, $i*B*8$, $(i+1)*B*8$), 256, "", "").
4. $z = z$ || right_encode($n$) || right_encode(0).
5. $newX = z$.
6. return cSHAKE128($newX$, $L$, "ParallelHash", $S$).
```

ParallelHashXOF256(*X, B, L, S*):

Validity Conditions: $0 < B < 2^{2040}$ and $\lceil len(X)/B \rceil < 2^{2040}$ and $0 \leq L$ and $len(S) < 2^{2040}$

```
1. $n = \lceil (\text{len}(X)/8) / B \rceil$.
2. $z$ = left_encode($B$).
3. for $i = 0$ to $n-1$:
       $z = z$ || cSHAKE256(substring($X$, $i*B*8$, $(i+1)*B*8$), 512, "", "").
4. $z = z$ || right_encode($n$) || right_encode(0).
5. $newX = z$.
6. return cSHAKE256($newX$, $L$, “ParallelHash”, $S$).
```

7 Implementation Considerations

7.1 Precomputation

cSHAKE is defined so that all the calls to the underlying KECCAK-*f* function [1] to accommodate the function name *N* and the customization string *S* will process an integer multiple of *r* bits, where *r* is the rate parameter. An implementation can precompute the result of processing this padded block of *N* and *S* with cSHAKE, and thus, will suffer no performance penalty when reusing the same choices of *N* and *S* in multiple cSHAKE executions. Since TupleHash and ParallelHash are defined in terms of cSHAKE, this same precomputation is available to implementations of those functions as well.

KMAC can precompute the result of processing *N* and *S*, and the result of processing the key *K*. Thus, KMAC128 using a fixed, precomputed customization string and key will process an input string as efficiently as SHAKE128.

7.2 Limited Implementations

The cSHAKE, KMAC, TupleHash, and ParallelHash functions are defined to accept a wide range of possible inputs (including unreasonably long inputs, and inputs involving fractional bytes), and to produce a wide range of possible output lengths. However, it is acceptable for a specific implementation to limit the possible inputs that it will process, and the allowed output lengths that it will produce.

For example, it would be acceptable to limit an implementation of any of these functions to producing no more than 65536 bytes of output, or to producing only whole bytes of output, or to accepting only byte strings (never fractional bytes) as inputs. Additionally, implementations intended for only a specific, limited use may further restrict the sets of inputs they will process. For example, an implementation of TupleHash256 used only to process a 6-tuple of strings, and always using a customization string of "address tuple", would be acceptable.

If it is possible for an implementation of one of these functions to be given a set of inputs that it cannot process, then the implementation shall signal an error condition and refuse to produce an output.

7.3 Exploiting Parallelism in ParallelHash

Specific implementations of ParallelHash are permitted to restrict their implementation to a small subset of the allowed values. For example, it would be acceptable for a particular implementation to only allow a single value of *B* if it were only expected to interoperate with another implementation that similarly restricted *B* to that same value.

ParallelHash can be implemented in a straightforward and reasonably efficient way even when only sequential processing is available. However, a much faster implementation is possible when each of the individual blocks of the message can be handled in parallel. The choice of block size *B* can have a huge impact on the efficiency of ParallelHash in this case. ParallelHash is designed so that any machine that can apply parallel processing can, in principle, benefit from that parallel processing. For example, a machine that can hash four blocks in parallel and a machine that can hash 32 blocks in parallel can each benefit from all the parallel processing ability that is available.

8 Security Considerations

8.1 Claimed Security Strength

cSHAKE, KMAC, TupleHash, and ParallelHash are all defined for two claimed security strengths: 128 bits and 256 bits.

cSHAKE128, KMAC128, TupleHash128, and ParallelHash128 each provide a security strength of 128 bits. This means that, for a given output length L, there is no *generic attack* on one of these functions requiring less than 2^{128} work that does not also exist for any hash function with the same output length. Similarly, cSHAKE256, KMAC256, TupleHash256, and ParallelHash256 each provides a security strength of 256 bits against generic attacks.

8.2 Security Properties for the Function-Name and Customization Strings

8.2.1 Equivalent Security to SHAKE for Any Legal *N* and *S*

For a given choice of the function name N and the customization string S, cSHAKE128(X, L, N, S) has exactly the same security properties as SHAKE128(X, L); cSHAKE256(X, L, N, S) has exactly the same security properties as SHAKE256(X, L). There are no "weak" values for N or S.

8.2.2 Different *N* and *S* Give Unrelated Outputs

Suppose that ($n1$, $s1$) and ($n2$, $s2$) are two name and customization string pairs, and either $n1 \neq n2$, or $s1 \neq s2$. Furthermore, suppose that $q1$ and $q2$ are the lengths of the requested output. Then, cSHAKE(X, $q1$, $n1$, $s1$) and cSHAKE(X, $q2$, $n2$, $s2$) can be treated as unrelated functions of X. That is, they can be treated as if they were two completely different functions, with output lengths $q1$ and $q2$, respectively. This means, for example, that:

- Keys $k1$ and $k2$, where $k1$ = cSHAKE($x1$, $q1$, $n1$, $s1$), and $k2$ = cSHAKE($x2$, $q2$, $n2$, $s2$), and both keys are derived from secret, but related, quantities $x1$ and $x2$, will not be susceptible to related key attacks (with complexity less than the claimed security strength of the cSHAKE function.)
- Finding a collision such that cSHAKE128($x1$, L, $n1$, $s1$) = cSHAKE128($x2$, L, $n2$, $s2$) will require a computational complexity on the order of min($2^{L/2}$, 2^{128}). Similarly, finding a collision such that cSHAKE256($x1$, L, $n1$, $s1$) = cSHAKE256($x2$, L, $n2$, $s2$), will require a computational complexity on the order of min($2^{L/2}$, 2^{256}).

Because KMAC, TupleHash, and ParallelHash are derived from cSHAKE, they inherit these properties. Specifically:

- Each of these functions gives outputs unrelated to the outputs of any of the other functions. There is, for example, no relationship between the outputs of KMAC (for any set of inputs) and TupleHash (for any set of inputs).
- For any of these functions, using a different customization string gives an unrelated output. Thus, if $s1 \neq s2$, ParallelHash(X, B, L, $s1$) and ParallelHash(X, B, L, $s2$) are expected to have no particular relationship.

Except when used in XOF mode, KMAC, TupleHash, and ParallelHash have the additional property that, even with the same customization string, blocksize, key, etc., instances of the functions with different output lengths can be treated as unrelated functions of their remaining inputs. Thus, for example, ParallelHash(X, B, $q1$, S) and ParallelHash(X, B, $q2$, S) can be treated as independent hash functions with input X and output lengths $q1$ and $q2$, respectively.

Note that cSHAKE does not share this property. For $q1< q2$, cSHAKE(X, $q1$, N, S) is a prefix of cSHAKE(X, $q2$, N, S). This is a property that cSHAKE shares with SHAKE and other XOFs. This property is discussed in more detail in Appendix A.2 of FIPS 202.

8.3 Collisions and Preimages

All these functions support variable output lengths. The difficulty of an attacker finding a collision or preimage for any of these functions depends on both the claimed security strength and the output length.

A function like cSHAKE128, with a claimed security strength of 128 bits, may be vulnerable to a collision or preimage attack with 2^{128} work regardless of its output length—a longer output does not, in general, improve its security against these attacks. However, a shorter output can make the function more vulnerable to these attacks. With an output of L bits, a collision attack will require $\min(2^{L/2}, 2^{128})$ work, and a preimage attack will require at least $\min(2^{L}, 2^{128})$ work.

8.4 Guidance for Using KMAC Securely

For maximum flexibility and usefulness, the KMAC functions are defined for arbitrary-sized output lengths and key lengths (up to $2^{2040}–1$ bits). However, not all such output and key lengths are secure.

8.4.1 KMAC Key Length

The input key length is the parameter that is most straightforwardly translated into a security strength. Given a small number of known (MAC, plaintext) pairs, an attacker requires at most $2^{\text{len}(K)}$ operations to find the key K.

Applications of this Recommendation **shall not** select an input key, K, whose length is less than their required security strength. Guidance for cryptographic algorithm and key-size selection is available in [4].

8.4.2 KMAC Output Length

The output length is another important security parameter for KMAC—it determines the probability that an online guessing attack will succeed in forging a MAC tag. In particular, an attacker will need to submit, on average, 2^{L} invalid (message, MAC) pairs for each successful forgery. Since L only affects online attacks, a system that uses KMAC for message authentication can mitigate attacks that exploit a short L by limiting the total number of verification failures allowed under a given key.

When used as a MAC, applications of this Recommendation **shall not** select an output length *L* that is less than 32 bits, and **shall** only select an output length less than 64 bits after a careful risk analysis is performed.

To illustrate the security properties of KMAC for given parameter settings, Table 1 lists a few other approved MAC algorithms along with equivalent settings for KMAC as an example. Note that no truncation of the associated tags (i.e., the 128-bit tag for AES-CMAC, the 256-bit tag for HMAC-SHA256, and the 512-bit tag for HMAC-SHA512) is assumed in the listed algorithms, and that equivalent settings of different MAC algorithms do not result in the same output.

Table 1: Equivalent security settings for KMAC and previously standardized MAC algorithms

Existing MAC Algorithm	KMAC Equivalent
AES-CMAC (*K*, *text*)	KMAC128 (*K*, *text*, 128, *S*)
HMAC-SHA256 (*K*, *text*)	KMAC256 (*K*, *text*, 256, *S*)
HMAC-SHA512 (*K*, *text*)	KMAC256 (*K*, *text*, 512, *S*)

Appendix A—KMAC, TupleHash, and ParallelHash in Terms of KECCAK[*c*]

FIPS 202 specifies the KECCAK[*c*] function, on which the SHA-3 and SHAKE functions are built. KMAC, TupleHash, and ParallelHash are defined in terms of cSHAKE, as specified in Sec. 3. In this appendix, KMAC, TupleHash, ParallelHash and these functions in XOF mode are defined directly in terms of KECCAK[*c*]. These definitions are exactly equivalent to the definitions made in terms of cSHAKE in Secs. 4, 5, and 6.

KMAC128(*K*, *X*, *L*, *S*):
Validity Conditions: $len(K) < 2^{2040}$ and $0 \leq L < 2^{2040}$ and $len(S) < 2^{2040}$

```
1. newX = bytepad(encode_string(K), 168) || X || right_encode(L).
2. T = bytepad(encode_string("KMAC") || encode_string(S), 168).
3. return KECCAK[256](T || newX || 00, L).
```

KMAC256(*K*, *X*, *L*, *S*):
Validity Conditions: $len(K) < 2^{2040}$ and $0 \leq L < 2^{2040}$ and $len(S) < 2^{2040}$

```
1. newX = bytepad(encode_string(K), 136) || X || right_encode(L).
2. T = bytepad(encode_string("KMAC") || encode_string(S), 136).
3. return KECCAK[512](T || newX || 00, L).
```

KMACXOF128(*K*, *X*, *L*, *S*):
Validity Conditions: $len(K) < 2^{2040}$ and $0 \leq L$ and $len(S) < 2^{2040}$

```
1. newX = bytepad(encode_string(K), 168) || X || right_encode(0).
2. T = bytepad(encode_string("KMAC") || encode_string(S), 168).
3. return KECCAK[256](T || newX || 00, L).
```

KMACXOF256(*K*, *X*, *L*, *S*):
Validity Conditions: $len(K) < 2^{2040}$ and $0 \leq L$ and $len(S) < 2^{2040}$

```
1. newX = bytepad(encode_string(K), 136) || X || right_encode(0).
2. T = bytepad(encode_string("KMAC") || encode_string(S), 136).
3. return KECCAK[512](T || newX || 00, L).
```

TupleHash128(*X*, *L*, *S*):
Validity Conditions: $0 \leq L < 2^{2040}$ and $len(S) < 2^{2040}$

```
1. z = "".
2. n = the number of input strings in the tuple X.
3. for i = 1 to n:
       z = z || encode_string(X[i]).
4. newX = z || right_encode(L).
5. T = bytepad(encode_string("TupleHash") || encode_string(S), 168).
6. return KECCAK[256](T || newX || 00, L).
```

TupleHash256(*X, L, S*):
Validity Conditions: $0 \le L < 2^{2040}$ and $len(S) < 2^{2040}$

```
1. z = "".
2. n = the number of input strings in the tuple X.
3. for i = 1 to n:
       z = z || encode_string(X[i]).
4. newX = z || right_encode(L).
5. T = bytepad(encode_string("TupleHash") || encode_string(S), 136).
6. return KECCAK[512](T || newX || 00, L).
```

TupleHashXOF128(*X, L, S*):
Validity Conditions: $0 \le L$ and $len(S) < 2^{2040}$

```
1. z = "".
2. n = the number of input strings in the tuple X.
3. for i = 1 to n:
       z = z || encode_string(X[i]).
4. newX = z || right_encode(0).
5. T = bytepad(encode_string("TupleHash") || encode_string(S), 168).
6. return KECCAK[256](T || newX || 00, L).
```

TupleHashXOF256(*X, L, S*):
Validity Conditions: $0 \le L$ and $len(S) < 2^{2040}$

```
1. z = "".
2. n = the number of input strings in the tuple X.
3. for i = 1 to n:
       z = z || encode_string(X[i]).
4. newX = z || right_encode(0).
5. T = bytepad(encode_string("TupleHash") || encode_string(S), 136).
6. return KECCAK[512](T || newX || 00, L).
```

ParallelHash128(*X, B, L, S*):
Validity Conditions: $0 < B < 2^{2040}$ and $\lceil len(X)/B \rceil < 2^{2040}$ and
$0 \le L < 2^{2040}$ and $len(S) < 2^{2040}$

```
1. n = ⌈ (len(X)/8) / B ⌉.
2. z = left_encode(B).
3. for i = 0 to n−1:
       z = z || KECCAK[256]( substring(X, i*B*8, (i+1)*B*8) || 1111, 256).
4. z = z || right_encode(n) || right_encode(L).
5. newX = z.
6. T = bytepad(encode_string("ParallelHash") || encode_string(S), 168).
7. return KECCAK[256](T || newX || 00, L).
```

ParallelHash256(*X, B, L, S*):

Validity Conditions: $0 < B < 2^{2040}$ and $\lceil len(X)/B \rceil < 2^{2040}$ and
$0 \leq L < 2^{2040}$ and $len(S) < 2^{2040}$

```
1. n = ⌈ (len(X)/8) / B ⌉.
2. z = left_encode(B).
3. for i = 0 to n−1:
       z = z || KECCAK[512]( substring(X, i*B*8, (i+1)*B*8) || 1111, 512).
4. z = z || right_encode(n) || right_encode(L).
5. newX = z.
6. T = bytepad(encode_string("ParallelHash") || encode_string(S), 136).
7. return KECCAK[512](T || newX || 00, L).
```

ParallelHashXOF128(*X*, *B*, *L*, *S*):
Validity Conditions: $0 < B < 2^{2040}$ and $\lceil len(X)/B \rceil < 2^{2040}$ and
$0 \leq L$ and $len(S) < 2^{2040}$

```
1. n = ⌈ (len(X)/8) / B ⌉.
2. z = left_encode(B).
3. for i = 0 to n−1:
       z = z || KECCAK[256]( substring(X, i*B*8, (i+1)*B*8) || 1111, 256).
4. z = z || right_encode(n) || right_encode(0).
5. newX = z.
6. T = bytepad(encode_string("ParallelHash") || encode_string(S), 168).
7. return KECCAK[256](T || newX || 00, L).
```

ParallelHashXOF256(*X*, *B*, *L*, *S*):
Validity Conditions: $0 < B < 2^{2040}$ and $\lceil len(X)/B \rceil < 2^{2040}$ and
$0 \leq L$ and $len(S) < 2^{2040}$

```
1. n = ⌈ (len(X)/8) / B ⌉.
2. z = left_encode(B).
3. for i = 0 to n−1:
       z = z || KECCAK[512]( substring(X, i*B*8, (i+1)*B*8) || 1111, 512).
4. z = z || right_encode(n) || right_encode(0).
5. newX = z.
6. T = bytepad(encode_string("ParallelHash") || encode_string(S), 136).
7. return KECCAK[512](T || newX || 00, L).
```

Appendix B—Hashing into a Range (Informative)

XOFs, PRFs, and hash functions with variable-length output like cSHAKE, KMAC, TupleHash, and ParallelHash can easily be used to generate an integer X within the range $0 \leq X < R$, denoted as $0..R-1$ in this document, for any positive integer R. The following method will produce outputs that are extremely close to a uniform distribution over that range, assuming that the above functions approximate a uniform random variable.

In order to hash into an integer in the range $0..R-1$, do the following:

1. Let $k = \lceil \lg(R) \rceil + 128$.
2. Call the hash function with a requested length of at least k bits. Let the resulting bit string be Z.
3. Let N = *bits_to_integer*(Z) mod R, where the *bits_to_integer* function is defined below.

At the end of this process, the variable N contains an integer that is extremely close to being uniformly distributed in the range $0..R-1$. For any possible output value t such that $0 \leq t < R$, the following statement is true[12].

$|\mathrm{Prob}(N=t) - 1/R| \leq 2^{-128}/R$.

In other words, the output of this process will have a very small bias. No value will be very much more or less likely to appear as the result of this process than it would have been, had an integer been selected uniformly at random from the integers between 0 and $R-1$, inclusive.

This technique can be applied to SHAKE, cSHAKE, KMAC, TupleHash, or ParallelHash whenever an integer within a specific range is needed, so long as it is acceptable for the resulting integer to have this very small deviation from the uniform distribution on the integers $\{0, 1, ..., R-1\}$.

The *bits_to_integer* function converts a bit string to an integer as follows:

bits_to_integer ($b_1, b_2, \ldots, b_n$):

1. Let ($b_1, b_2, \ldots, b_n$) be the bits of a bit string from the most significant to the least significant bits.
2. $x = \sum_{i=1}^{n} 2^{(n-i)} b_i$.
3. Return (x).

[12] In fact, the bound is slightly tighter than this. If w = the length of the bitstring Z in bits ($w \geq \lceil \lg(R) \rceil + 128$), then $|\mathrm{Prob}(N=t) - 1/R| \leq 2^{-w}$.

Appendix C—References

[1] National Institute of Standards and Technology, *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*, Federal Information Processing Standards (FIPS) Publication 202, August 2015, 37 pp. http://dx.doi.org/10.6028/NIST.FIPS.202.

[2] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, *The KECCAK reference, version 3.0*, January 14, 2011, 69 pp. http://keccak.noekeon.org/Keccak-reference-3.0.pdf [accessed 12/21/2016].

[3] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, *Cryptographic sponge functions, version 0.1*, January 14, 2011, 93 pp. http://sponge.noekeon.org/CSF-0.1.pdf [accessed 12/21/2016].

[4] E. Barker, *Recommendation for Key Management, Part 1: General*, NIST Special Publication (SP) 800-57 Part 1 Revision 4, National Institute of Standards and Technology, January 2016, 160 pp. http://dx.doi.org/10.6028/NIST.SP.800-57pt1r4.