

NIST Special Publication 800 NIST SP 800-185r1 ipd

SHA-3 Derived Functions:

cSHAKE, KMAC, TUPLEHASH, and PARALLELHASH

(Initial Public Draft)

John Kelsey
Ray Perlner
Meltem Sonmez Turan

This publication is available free of charge from:
<https://doi.org/10.6028/NIST.SP.800-185r1.ipd>

12 NIST Special Publication 800
13 NIST SP 800-185r1 ipd

14 SHA-3 Derived Functions:

15 *cSHAKE, KMAC, TUPLEHASH, and PARALLELHASH*

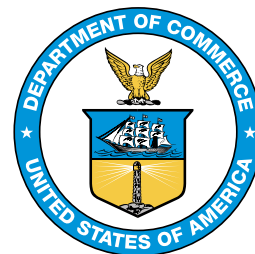
16 (Initial Public Draft)

17 John Kelsey¹, Ray Perlner¹, Meltem Sonmez Turan¹

18 ¹ Computer Security Division, Information Technology Laboratory, NIST, Gaithersburg, Maryland, USA

19 This publication is available free of charge from:
20 <https://doi.org/10.6028/NIST.SP.800-185r1.ipd>

October 2026



22 U.S. Department of Commerce
23 Howard Lutnick, Secretary

24 National Institute of Standards and Technology
25 Arvind Raman, NIST Director and Under Secretary of Commerce for Standards and Technology

26 **NIST Technical Series Publications:** <https://www.nist.gov/nist-research-library/nist-publications>

27 **SP 800 Series**

28 The Special Publication 800-series reports on ITL's research, guidelines, and outreach efforts in information
29 system security, and its collaborative activities with industry, government, and academic organizations.

30 **Authority**

31 This publication has been developed by NIST in accordance with its statutory responsibilities under the
32 Federal Information Security Modernization Act (FISMA) of 2014, 44 U.S.C § 3551 et seq., Public Law (P.L.)
33 113-283. NIST is responsible for developing information security standards and guidelines, including minimum
34 requirements for federal information systems, but such standards and guidelines shall not apply to national
35 security systems without the express approval of appropriate federal officials exercising policy authority over
36 such systems. This guideline is consistent with the requirements of the Office of Management and Budget
37 (OMB) Circular A-130.

38 Nothing in this publication should be taken to contradict the standards and guidelines made mandatory and bind-
39 ing on federal agencies by the Secretary of Commerce under statutory authority. Nor should these guidelines be
40 interpreted as altering or superseding the existing authorities of the Secretary of Commerce, Director of the OMB,
41 or any other federal official. This publication may be used by nongovernmental organizations on a voluntary
42 basis and is not subject to copyright in the United States. Attribution would, however, be appreciated by NIST.

43 **Reports on Computer Systems Technology**

44 The Information Technology Laboratory (ITL) at the National Institute of Standards and Technology
45 (NIST) promotes the U.S. economy and public welfare by providing technical leadership for the Nation's
46 measurement and standards infrastructure. ITL develops tests, test methods, reference data, proof of concept
47 implementations, and technical analyses to advance the development and productive use of information
48 technology. ITL's responsibilities include the development of management, administrative, technical,
49 and physical standards and guidelines for the cost-effective security and privacy of other than national
50 security-related information in federal information systems.

51 **References to Other Publications**

52 There may be references in this publication to other publications currently under development by NIST
53 in accordance with its assigned statutory responsibilities. The information in this publication, including
54 concepts and methodologies, may be used by federal agencies even before the completion of such companion
55 publications. Thus, until each publication is completed, current requirements, guidelines, and procedures,
56 where they exist, remain operative. For planning and transition purposes, federal agencies may wish to closely
57 follow the development of these new publications by NIST.

58 Organizations are encouraged to review all draft publications during public comment periods and provide
59 feedback to NIST. Many NIST cybersecurity publications, other than the ones noted above, are available
60 at <https://csrc.nist.gov/publications>

61 **Non-Endorsement Disclaimer**

62 Certain equipment, instruments, software, or materials, commercial or non-commercial, are identified in
63 this paper in order to specify the experimental procedure adequately. Such identification does not imply
64 recommendation or endorsement of any product or service by NIST, nor does it imply that the materials
65 or equipment identified are necessarily the best available for the purpose.

Call for Patent Claims

This public review includes a call for information on essential patent claims (claims whose use would be required for compliance with the recommendations or requirements in this Information Technology Laboratory (ITL) draft publication). Such recommendations and/or requirements may be directly stated in this ITL Publication or by reference to another publication. This call also includes disclosure, where known, of the existence of pending U.S. or foreign patent applications relating to this ITL draft publication and of any relevant unexpired U.S. or foreign patents.

ITL may require from the patent holder, or a party authorized to make assurances on its behalf, in written or electronic form, either:

1. assurance in the form of a general disclaimer to the effect that such party does not hold and does not currently intend holding any essential patent claim(s); or
2. assurance that a license to such essential patent claim(s) will be made available to applicants desiring to utilize the license for the purpose of complying with the recommendations or requirements in this ITL draft publication either:
 - (a) under reasonable terms and conditions that are demonstrably free of any unfair discrimination; or
 - (b) without compensation and under reasonable terms and conditions that are demonstrably free of any unfair discrimination.

Such assurance shall indicate that the patent holder (or third party authorized to make assurances on its behalf) will include in any documents transferring ownership of patents subject to the assurance, provisions sufficient to ensure that the commitments in the assurance are binding on the transferee, and that the transferee will similarly include appropriate provisions in the event of future transfers with the goal of binding each successor-in-interest.

The assurance shall also indicate that it is intended to be binding on successors-in-interest regardless of whether such provisions are included in the relevant transfer documents.

Such statements should be addressed to: SP800-185@list.nist.gov

Publication History

Approved by the NIST Editorial Review Board on YYYY-MM-DD [Will be added upon final publication].
Supersedes NIST <Series XXX> (Month Year), DOI:<Insert doi here> [Will be added upon final publication]

Suggested Citation

John Kelsey, Ray Perlner, Meltem Sonmez Turan (2026). SHA-3 Derived Functions: cSHAKE, KMAC, TUPLEHASH, and PARALLELHASH. (National Institute of Standards and Technology, Gaithersburg, MD) NIST Special Publication 800 (SP) NIST SP 800-185r1 ipd (Initial Public Draft). DOI:10.6028/NIST.SP.800-185r1.ipd

Author Names and ORCID Identifiers

John Kelsey (0000-0002-3427-1744); Ray Perlner (0000-0001-8793-2238); Meltem Sonmez Turan (0000-0002-1950-7130).

Additional Information

Additional information about this publication, including related content, potential updates, and document history, is available at <https://csrc.nist.gov/pubs/sp/800/185/r1/ipd>.

Public Comments Period: October 8, 2026 – December 7, 2026

Public Comments Contact: SP800-185@list.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 128- and 256-bit security strengths. cSHAKE is a customizable variant of the SHAKE function, as defined in Federal Information Processing Standards publication (FIPS) 202. KMAC (KECCAK Message Authentication Code) is a variable-length message authentication code algorithm based on KECCAK (specified in FIPS 202) that 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. This revision adds a streaming mode for XOF computations, which allows applications to process input and request output incrementally when the total input or output length is not known in advance.

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.

Table of Contents

122

123	Abstract	i
124	Acknowledgments	v
125	Note to the Reviewers	v
126	1. Introduction	1
127	2. Preliminaries	4
128	2.1. Terms and Acronyms	4
129	2.2. Basic Operations	6
130	2.3. Internal Functions	6
131	2.3.1. Integer to Byte String Encoding	6
132	2.3.2. String Encoding	7
133	2.3.3. Padding	7
134	2.3.4. Substrings	8
135	3. cSHAKE	9
136	3.1. Overview	9
137	3.2. Parameters	9
138	3.3. cSHAKE Definition	10
139	3.4. Using the Function Name	10
140	3.5. Using the Customization String	10
141	3.6. SHAKE and cSHAKE as XOFs	11
142	4. KMAC	12
143	4.1. Overview	12
144	4.2. Parameters	12
145	4.3. KMAC Specification	12
146	4.4. KMACXOF Specification	13
147	5. TUPLEHASH	14
148	5.1. Overview	14
149	5.2. Parameters	14
150	5.3. TUPLEHASH Specification	14
151	5.4. TUPLEHASHXOF Specification	15
152	6. PARALLELHASH	16
153	6.1. Overview	16
154	6.2. Parameters	16
155	6.3. PARALLELHASH Specification	16
156	6.4. PARALLELHASHXOF Specification	17

157	7. XOF Access and Streaming Interfaces	19
158	7.1. Interface Model	19
159	7.2. Streaming Access to cSHAKE	20
160	7.3. Streaming Access to SHAKE	22
161	7.4. Streaming Access to KMACXOF	22
162	7.5. Streaming Access to TUPLEHASHXOF	24
163	7.6. Streaming Access to PARALLELHASHXOF	25
164	8. Implementation Considerations	29
165	8.1. Precomputation	29
166	8.2. Limited Implementations	29
167	8.3. Exploiting Parallelism in PARALLELHASH	29
168	9. Security Considerations	31
169	9.1. Claimed Security Strength	31
170	9.2. Security Properties for the Function Name and Customization Strings	31
171	9.2.1. Equivalent Security to SHAKE for Any Legal N and S'	31
172	9.2.2. Domain Separation Using Function Name or Customization Strings	31
173	9.3. Collision and Preimage Resistance	32
174	9.4. Guidelines for Using KMAC Securely	32
175	9.4.1. KMAC Key Length	32
176	9.4.2. KMAC Output Length	33
177	9.5. Security of Sponge Constructions	33
178	References	35
179	Appendix A. KMAC, TUPLEHASH, and PARALLELHASH in Terms of KECCAK$[c]$	37
180	Appendix B. Hashing Into a Range	40
181	Appendix C. Summary of Changes	41

List of Algorithms

1.	<code>left_encode(x)</code>	7
2.	<code>right_encode(x)</code>	7
3.	<code>encode_string(S)</code>	7
4.	<code>bytepad(X, w)</code>	8
5.	<code>substring(X, a, b)</code>	8
6.	<code>cSHAKE$\langle s \rangle$(X, L, N, S)</code>	10
7.	<code>KMAC$\langle s \rangle$(K, X, L, S)</code>	12
8.	<code>KMACXOF$\langle s \rangle$(K, X, L, S)</code>	13
9.	<code>TUPLEHASH$\langle s \rangle$(X, L, S)</code>	15
10.	<code>TUPLEHASHXOF$\langle s \rangle$(X, L, S)</code>	15
11.	<code>PARALLELHASH$\langle s \rangle$(X, B, L, S)</code>	17
12.	<code>PARALLELHASHXOF$\langle s \rangle$(X, B, L, S)</code>	17
13.	Reference procedure for streaming XOF access to <code>cSHAKE$\langle s \rangle$</code>	21
14.	Reference procedure for streaming XOF access to <code>KMACXOF$\langle s \rangle$</code>	23
15.	Reference procedure for streaming XOF access to <code>TUPLEHASHXOF$\langle s \rangle$</code>	25
16.	Reference procedure for streaming XOF access to <code>PARALLELHASHXOF$\langle s \rangle$</code>	27
17.	<code>KMAC$\langle s \rangle$(K, X, L, S)</code> in terms of <code>KECCAK$[c]$</code>	37
18.	<code>KMACXOF$\langle s \rangle$(K, X, L, S)</code> in terms of <code>KECCAK$[c]$</code>	37
19.	<code>TUPLEHASH$\langle s \rangle$(X, L, S)</code> in terms of <code>KECCAK$[c]$</code>	37
20.	<code>TUPLEHASHXOF$\langle s \rangle$(X, L, S)</code> in terms of <code>KECCAK$[c]$</code>	38
21.	<code>PARALLELHASH$\langle s \rangle$(X, B, L, S)</code> in terms of <code>KECCAK$[c]$</code>	38
22.	<code>PARALLELHASHXOF$\langle s \rangle$(X, B, L, S)</code> in terms of <code>KECCAK$[c]$</code>	39
23.	<code>bits_to_integer($b_0, b_1, \dots, b_{n-1}$)</code>	40

List of Figures

Figure 1.	Relationships among the KECCAK-based functions	3
------------------	--	---

List of Tables

Table 1.	Basic Operations and Notation	6
Table 2.	KMAC parameter choices corresponding to selected MAC algorithms.	33

Acknowledgments

The authors acknowledge Shu-jen Chang, a co-author of the original version of this recommendation, for her significant contributions to the development of NIST SP 800-185. The authors thank KECCAK team members Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche for their feedback and contributions to the development of the original version of this recommendation. The authors also thank their colleagues for reviewing drafts of this document and contributing to its development.

Note to the Reviewers

1. The KECCAK-derived functions in this recommendation have been updated to describe a streaming mode of access that provides full XOF functionality. This involves calls to three functions: INIT, ABSORB, and SQUEEZE.
2. Illustrative pseudocode is provided for the streaming interface, but the correct behavior of all streaming modes is defined in terms of the corresponding single-call interface to the functions.
3. The single-call interfaces for these functions are equivalent to the functions defined in the previous version of this document.
4. cSHAKE, KMAC, and PARALLELHASH use the ABSORB function to incrementally input data into the function so that $\text{ABSORB}(X1)$ followed by $\text{ABSORB}(X2)$ has the same effect as $\text{ABSORB}(X1 \parallel X2)$.
5. TUPLEHASH uses the ABSORB function somewhat differently in that each ABSORB call processes one complete tuple element. NIST invites comment on whether this different use of ABSORB might be confusing.

1. Introduction

Federal Information Processing Standards publication (FIPS) 202, *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions* [1], defines fixed-length hash functions SHA3-224, SHA3-256, SHA3-384, and SHA3-512 as well as eXtendable Output Functions (XOFs) SHAKE128 and SHAKE256. The SHA3 hash functions are named according to their output size, whereas the SHAKE functions are named according to their security strength. For example, SHA3-256 has a 256-bit output and provides 256 bits of preimage resistance but only 128 bits of collision resistance. SHAKE256 provides up to 256 bits of security against both preimage and collision attacks, provided that a sufficiently long output is requested. These primitives are based on KECCAK [2–4], whose underlying design follows a sponge construction [5].

FIPS 202 also supports a flexible scheme for domain separation between different functions derived from KECCAK—the algorithm [4] on which the SHA-3 Standard is based. Domain separation ensures that different named functions, such as SHA3-512 and SHAKE128, pass distinct encoded input strings into KECCAK. As a result, the outputs from different domains are expected to behave as outputs of distinct functions.

This recommendation defines four SHA-3-derived function families: cSHAKE, KMAC, TUPLEHASH, and PARALLELHASH. Each supports a user-selected customization string that can be used to provide domain separation among different applications of the same function. For example, a customization string could identify a particular application, such as computing a key fingerprint. Each function has a version that supports 128-bit security and one that supports 256-bit security, as described below.

The additional derived function families provide functionality that is not directly available from the functions specified in FIPS 202 [1]. The function families are:

- KMAC128 and KMAC256 (see Sec. 4) are keyed hash functions that support variable-length output and can be used as message authentication codes and pseudorandom functions. KMAC stands for KECCAK Message Authentication Code.
- TUPLEHASH128 and TUPLEHASH256 (see Sec. 5) hash tuples of input strings unambiguously by encoding each tuple element and its length so that different tuples, including tuples that differ only by the placement or presence of empty strings, are represented by different inputs to the underlying function.
- PARALLELHASH128 and PARALLELHASH256 (see Sec. 6) are variable-length hash functions that provide a mechanism to take advantage of parallel processing when hashing long messages.

cSHAKE, KMAC, TUPLEHASH, and PARALLELHASH all have the following properties in common:

- They are all derived from the functions specified in FIPS 202 [1].

- All have variants that support 128-bit and 256-bit security strengths. Following the convention of SHAKE in [1], these variants are named by their claimed security strength. Thus, cSHAKE128 is the cSHAKE variant with 128-bit security, and KMAC256 is the KMAC variant with 256-bit security. For conciseness, this recommendation uses the notation $F\langle s \rangle$, where $s \in \{128, 256\}$, to denote the corresponding security-strength variant of a function family F . For example, cSHAKE $\langle 128 \rangle$ denotes cSHAKE128, and cSHAKE $\langle 256 \rangle$ denotes cSHAKE256. The same convention is used for SHAKE, KMAC, TUPLEHASH, and PARALLELHASH.
- All of the functions except cSHAKE are defined in terms of cSHAKE.
- All support user-defined customization strings.
- All can produce outputs of arbitrary requested length.
 - KMAC, TUPLEHASH, and PARALLELHASH incorporate the requested output length into the input to cSHAKE so that even two calls to the same function with identical inputs are domain-separated when they request different outputs.
 - KMAC, TUPLEHASH, and PARALLELHASH also have XOF variants (respectively, KMACXOF, TUPLEHASHXOF, and PARALLELHASHXOF). These are intended for applications in which the functions need to process input data before the requested output length is known. They are domain-separated from any fixed-output length call to KMAC, TUPLEHASH, or PARALLELHASH.
 - cSHAKE, SHAKE, KMACXOF, TUPLEHASHXOF, and PARALLELHASHXOF support interfaces for incremental input processing and incremental XOF output, as specified in Section 7.

XOFs and *variable-length hash functions* are similar constructions in that both allow an application to request output of different lengths. However, they differ in how the requested output length affects the computation. For a XOF, the output can be viewed as an infinitely long output stream. Requesting more output bits simply gets more bits from that string, and so the first ℓ bits of a longer output are identical to the output obtained by requesting ℓ bits. In contrast, for the variable-length hash functions defined in this recommendation, the requested output length is encoded as part of the input to the underlying cSHAKE call. Thus, calls that differ only in the requested output length are domain-separated and expected to produce unrelated outputs. In this recommendation, cSHAKE, SHAKE, KMACXOF, TUPLEHASHXOF, and PARALLELHASHXOF provide XOF behavior, while KMAC, TUPLEHASH, and PARALLELHASH provide variable-length outputs whose fixed-output calls are domain-separated by the requested output length.

The relationships among the function families defined in this recommendation and the Keccak-based functions specified in FIPS 202 are illustrated in Figure 1.

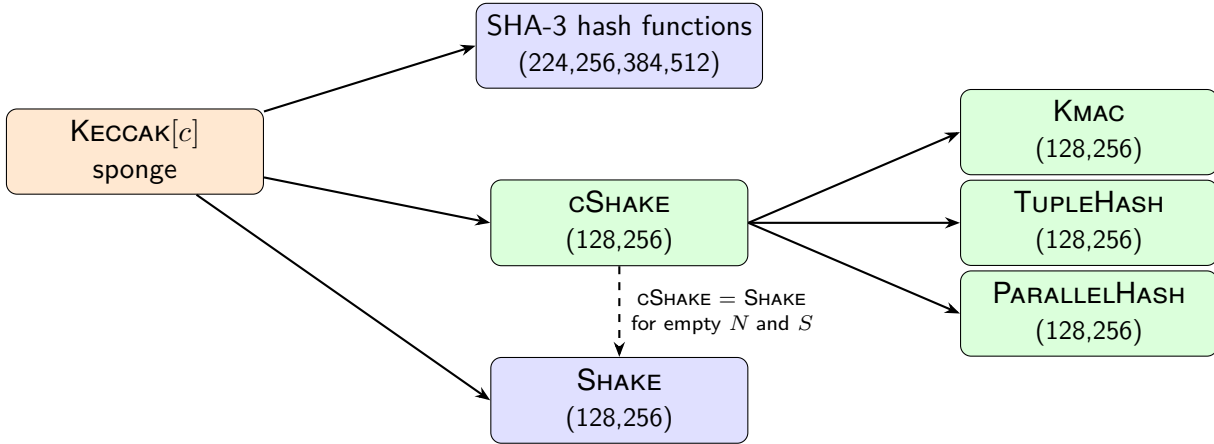


Figure 1: Relationships among the KECCAK-based function families and their variants. Solid arrows indicate derivation; the dashed arrow indicates that cSHAKE is equivalent to SHAKE when N and S are empty.

History. NIST Special Publication (SP) 800-185 [6] was first published in 2016 and specified cSHAKE, KMAC, TUPLEHASH, and PARALLELHASH as function families derived from KECCAK-based functions in FIPS 202 [1]. This revision adds explicit interfaces for incremental XOF access and incorporates technical and editorial clarifications based on public comments.

Organization. Section 2 introduces the notation, terminology, and internal functions used throughout this recommendation. Section 3 specifies cSHAKE, which serves as the basis for the other functions. Section 4 specifies KMAC, a keyed function for message authentication, keyed hashing, and pseudorandom-function applications. Section 5 specifies TUPLEHASH, which hashes tuples of bitstrings unambiguously. Section 6 specifies PARALLELHASH, which supports efficient parallel processing of large inputs separated into distinct blocks. Section 7 specifies XOF access and streaming interfaces. Sections 8 and 9 discuss implementation and security considerations, respectively. Appendix A expresses KMAC, TUPLEHASH, and PARALLELHASH directly in terms of the KECCAK[c] function. Appendix B describes a method for mapping function outputs to integers in the range of $\{0, 1, 2, \dots, R - 1\}$ for any positive integer R . Appendix C summarizes the changes made in Revision 1.

2. Preliminaries

In this document, binary constants are indicated in a monospaced font, as with 1001001. Bytes are written as two-digit hexadecimal numbers from the ASCII characters 0 through 9 and A through F, preceded by the prefix 0x, as with 0x49. 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. For example, because bits within a byte are displayed from low order to high order, 0x01 is written as 10000000, while 0x80 is written as 00000001.

These bit-ordering conventions follow the conventions established in FIPS 202 [1]. Character strings appear in this document in double-quotes. Character strings are interpreted as bitstrings whose length is a multiple of 8 bits, consisting of a 0 bit, followed by the 7-bit ASCII representation of each successive character. Equivalently, these character strings are encoded as byte strings in UTF-8.

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.

FIPS Federal Information Processing Standards publication.

Hash Function A deterministic function that maps an input bitstring to an output bitstring and is designed to provide specified cryptographic properties, such as preimage resistance or collision resistance.

Fixed-Length Hash Function A hash function for which the output length is fixed by the function definition. For a given fixed-length hash function, all outputs have the same length.

HMAC Keyed-Hash Message Authentication Code.

KECCAK A family of sponge functions built from KECCAK- f permutation and the multi-rate padding rule. The KECCAK-based functions used in this recommendation are specified in FIPS 202.

KMAC KECCAK Message Authentication Code.

MAC Message Authentication Code.

NIST National Institute of Standards and Technology.

Pseudorandom Function (PRF) A keyed function that, to any party that does not know the key, produces outputs that are computationally indistinguishable from the outputs of a uniformly random function.

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 [7] for defining a function from the following: 1) an underlying function on bitstrings of a fixed length, 2) a padding rule, and 3) a rate. Both the input and the output of the resulting function are bitstrings 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.

Streaming Mode A way of accessing a XOF in which input bits are fed incrementally into the function, and later output bits are requested incrementally from the function. In streaming mode, the complete input need not be known until output bits are requested, and the total number of output bits need not be known when output generation begins.

Variable-Length Hash Function A hash function that takes an input string and a requested output length and returns an output string of that length. Different requested output lengths are domain-separated, so outputs generated from the same input with different requested lengths are expected to be unrelated. In this recommendation, this domain separation is achieved by encoding the requested output length as part of the input to the underlying cSHAKE call.

eXtendable-Output Function (XOF) A function that takes an input string and can produce an output string of any requested length. The output is independent of the requested output length in the sense that, for the same input, the shorter output is always a prefix of the longer output.

2.2. Basic Operations

Table 1. Basic Operations and Notation

Notation	Description
$\lceil x \rceil$	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 $\lceil 6 \rceil = 6$.
0^s	For a non-negative integer s , 0^s is the string that consists of s consecutive 0 bits. In particular, 0^0 is the empty string.
$a \bmod b$	For integers a and b , with $b > 0$, $a \bmod b$ is the unique integer r such that $0 \leq r < b$ and $a = qb + r$ for some integer q .
$\text{enc}_8(i)$	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.
$\text{len}(X)$	For a bitstring X , $\text{len}(X)$ 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, $11001 \parallel 010 = 11001010$.

2.3. 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 `enc8()` to encode the individual bytes, these two functions are defined as follows:

Algorithm 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 $x_1, x_2, \dots, x_n$ be the base-256 encoding of x satisfying $x = \sum_{i=1}^n 2^{8(n-i)} x_i$.
 - 3: Let $O_i = \text{enc}_8(x_i)$, for $i = 1$ to n .
 - 4: Let $O_0 = \text{enc}_8(n)$.
 - 5: **return** $O = O_0 \parallel O_1 \parallel \dots \parallel O_{n-1} \parallel O_n$.
-

Algorithm 2 `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 $x_1, x_2, \dots, x_n$ be the base-256 encoding of x satisfying $x = \sum_{i=1}^n 2^{8(n-i)} x_i$.
 - 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** $O = O_1 \parallel O_2 \parallel \dots \parallel O_n \parallel O_{n+1}$.
-

346 As an example, `right_encode(0)` will yield 00000000 10000000, and `left_encode(0)` will yield
347 10000000 00000000.

348 **2.3.2. String Encoding**

349 The `encode_string` function encodes bitstring S so that S and its length can be parsed unambigu-
350 ously from the beginning of the encoded string. The function is defined as follows:

Algorithm 3 `encode_string( $S$ )`

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

- 1: **return** `left_encode(len( $S$ ))` $\parallel$ S .
-

351 As an example, when S is the empty string "", `encode_string( $S$ )` yields 10000000 00000000.

352 If the bitstring S is not byte-oriented (i.e., $\text{len}(S)$ is not a multiple of 8), the bitstring returned
353 from `encode_string( $S$ )` is also not byte-oriented. However, if $\text{len}(S)$ is a multiple of 8, then the
354 length of the output of `encode_string( $S$ )` will also be a multiple of 8.

355 **2.3.3. Padding**

356 The `bytepad( $X, w$ )` function prepends an encoding of the integer w to an input string X and then
357 pads the result with zeros until it is a byte string whose length in bytes is a multiple of w . In general,
358 `bytepad` is intended to be used on encoded strings. The byte string `bytepad(encode_string( $S$ ),  $w$ )`

359 can be parsed unambiguously from the left, whereas bytepad does not provide unambiguous
360 padding for all input strings.

361 The definition of bytepad() is as follows:

Algorithm 4 bytepad(X, w)

Validity Conditions: $0 < w < 2^{2040}$

```

1:  $z = \text{left\_encode}(w) \parallel X$ .
2: while  $\text{len}(z) \bmod 8 \neq 0$  do
3:    $z = z \parallel 0$ 
4: while  $(\text{len}(z)/8) \bmod w \neq 0$  do
5:    $z = z \parallel 00000000$ 
6: return  $z$ .
```

362 **2.3.4. Substrings**

363 Let parameters a and b be non-negative integers. Informally, the substring(X, a, b) function
364 returns a substring from the bitstring X containing the values at bit positions $a, a + 1, \dots, b - 1$,
365 inclusive. More precisely, the substring function operates as defined below. All bit positions in the
366 input and output strings are indexed from zero. Thus, the first bit in a string is in position 0, and
367 the last bit in an n -bit string is in position $n - 1$.

Algorithm 5 substring(X, a, b)

```

1: if  $a \geq b$  or  $a \geq \text{len}(X)$  then
2:   return the empty string.
3: else if  $b \leq \text{len}(X)$  then
4:   return the bits of  $X$  from position  $a$  to position  $b - 1$ , inclusive.
5: else
6:   return the bits of  $X$  from position  $a$  to position  $\text{len}(X) - 1$ , inclusive.
```

3. cSHAKE

3.1. Overview

The two variants, cSHAKE128 and cSHAKE256, support claimed security strengths of 128 and 256 bits, respectively. They are defined using the SHAKE and KECCAK[*c*] functions specified in FIPS 202 [1].

3.2. Parameters

Both cSHAKE functions take four parameters:

- X is the main input bitstring. It may be of any length¹, including zero.
- L is an integer representing the requested output length² in bits.
- N is a function-name bitstring 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 bitstring. The user selects this string to define a variant of the function. When no customization is desired, S is set to the empty string.³

An implementation of cSHAKE may 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 **shall** 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,

$$\begin{aligned}\text{cSHAKE128}(X, L, "", "") &= \text{SHAKE128}(X, L), \quad \text{and} \\ \text{cSHAKE256}(X, L, "", "") &= \text{SHAKE256}(X, L).\end{aligned}$$

cSHAKE is designed to provide domain separation across different choices of the function name string N and customization string S . Thus, for two calls

$$\text{cSHAKE}(X, L, N_1, S_1) \quad \text{and} \quad \text{cSHAKE}(X, L, N_2, S_2),$$

¹When 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.

²When the requested output length is zero (i.e., $L = 0$), cSHAKE, KMAC, TUPLEHASH, and PARALLELHASH return the empty string as the output.

³In 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.

if $N_1 \neq N_2$ or $S_1 \neq S_2$, the calls are domain-separated, and their outputs are expected to behave as outputs of distinct functions. This includes the case in which 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 [1].

3.3. cSHAKE 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 .

Algorithm 6 cSHAKE(s)(X, L, N, S)

Validity Conditions: $s \in \{128, 256\}$, $L \geq 0$, $\text{len}(N) < 2^{2040}$, and $\text{len}(S) < 2^{2040}$

```

1: if  $N = ""$  and  $S = ""$  then
2:   return SHAKE( $s$ )( $X, L$ )
3: else
4:    $r \leftarrow 1600 - 2s$ 
5:   return KECCAK[ $2s$ ](bytepad( $\text{encode\_string}(N) \parallel \text{encode\_string}(S), r/8$ )  $\parallel X \parallel 00, L$ )
```

3.4. Using the Function Name

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.⁴ This parameter provides a level of domain separation by function name. Users of cSHAKE should not assign their own values to N ; application-specific customization should instead be provided through the customization string S . Nonstandard values of N could cause interoperability problems with future NIST-defined functions.

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 .

Different customization strings can be used to domain-separate different applications of cSHAKE. For example, an application could use one customization string when computing a key fingerprint

⁴NIST will always make the function name N a byte-oriented value.

and a different customization string when computing a file identifier. This makes two uses behave as distinct functions, even when the same underlying cSHAKE variant is used.

The customization string could in principle be of any length less than 2^{2040} bits. However, implementations may restrict the length of S that they will accept.

3.6. SHAKE and cSHAKE as XOFs

cSHAKE is shown as a single function call with the full input and output known at the time the call is made. In addition to the single-call interface, cSHAKE (and SHAKE) can be accessed incrementally. cSHAKE can begin processing inputs before all input bits (or even the number of input bits) are known. Similarly, it can begin producing output bits before the total number of output bits is known, though all input bits must be fed into cSHAKE before any output bits are produced. This requires a somewhat more complicated “streaming” interface. Section 7 provides the specification for how the cSHAKE algorithm supports this kind of access. As with the single function call above, calling cSHAKE in streaming mode with $S = ""$, $N = ""$ gives SHAKE in streaming mode.

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. For fixed-output invocations of KMAC, the requested output length is encoded as part of the input to the underlying cSHAKE call. Therefore, invocations that differ only in the requested output length are domain-separated and generally produce different outputs. KMAC has two variants, KMAC128 and KMAC256, built from cSHAKE128 and cSHAKE256, respectively. These variants provide claimed security strengths of 128 bits and 256 bits, respectively, as discussed in Section 9.

4.2. Parameters

Both KMAC functions take the following parameters:

- K is a bitstring of any length, including zero.
- X is the main input bitstring and may be of any length, including zero.
- L is an integer representing the requested output length⁵ in bits.
- S is an optional customization bitstring of any length, including zero. If no customization is desired, S is set to the empty string.

Although the KMAC function is mathematically defined for keys of any length, approved uses require the length of K to be at least the required security strength (see Section 9.4.1).

4.3. KMAC Specification

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 $N = \text{"KMAC"} = 11010010\ 10110010\ 10000010\ 11000010$,⁶ and the optional customization string S .

Algorithm 7 $\text{KMAC}_{\langle s \rangle}(K, X, L, S)$

Validity Conditions: $s \in \{128, 256\}$, $\text{len}(K) < 2^{2040}$, $0 \leq L < 2^{2040}$, and $\text{len}(S) < 2^{2040}$

- 1: $r \leftarrow 1600 - 2s$
 - 2: $\text{newX} \leftarrow \text{bytepad}(\text{encode_string}(K), r/8) \parallel X \parallel \text{right_encode}(L)$
 - 3: **return** $\text{cSHAKE}_{\langle s \rangle}(\text{newX}, L, \text{"KMAC"}, S)$
-

⁵There 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. 4.4.

⁶In binary representation, bytes are written with the low-order bit first, as specified in Sec. 2.

4.4. KMACXOF Specification

Some applications of KMAC may not know the number of output bits that they will need until after the outputs begin to be produced. For these applications, the XOF variant of KMAC, called KMACXOF, provides cSHAKE-like behavior.

Conceptually, KMACXOF produces an infinite-length output string, and the caller simply uses as many bits of the output string as needed. Truncated outputs of KMACXOF can be computed by the function $\text{KMACXOF}_{\langle s \rangle}(K, X, L, S)$ given by the pseudocode below. KMACXOF is identical to the ordinary version of KMAC except that the encoded output length is set to 0, regardless of the requested output length L , as shown in Step 2 of the pseudocode below.

Algorithm 8 $\text{KMACXOF}_{\langle s \rangle}(K, X, L, S)$

Validity Conditions: $s \in \{128, 256\}$, $\text{len}(K) < 2^{2040}$, $L \geq 0$, and $\text{len}(S) < 2^{2040}$

- 1: $r \leftarrow 1600 - 2s$
 - 2: $\text{newX} \leftarrow \text{bytepad}(\text{encode_string}(K), r/8) \parallel X \parallel \text{right_encode}(0)$
 - 3: **return** $\text{cSHAKE}_{\langle s \rangle}(\text{newX}, L, \text{"KMAC"}, S)$
-

As with cSHAKE, KMACXOF is shown above as a single function call with the full input and output known at the time the call is made. KMACXOF can also be accessed incrementally, as specified in Section 7.

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 potentially empty input strings in an unambiguous way. Such a tuple may consist of any number of strings, including zero, and is represented in this document as a sequence of strings or variables in parentheses, like ("a", "b", "c", ..., "z").

TUPLEHASH is designed to hash a sequence of strings unambiguously, so 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 of the remaining input parameters are kept the same, and the two resulting concatenated strings are identical without string encoding.

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 bitstrings, any or all of which may be an empty string.
- L is an integer that represents the requested output length in bits.
- S is an optional customization bitstring of any length, including zero. If no customization is desired, S is set to the empty string.

5.3. TUPLEHASH Specification

TUPLEHASH encodes the sequence of input strings in an unambiguous way, 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" and the optional customization string S .

If X is a tuple of n bitstrings, let $X[i]$ be the i th bitstring, numbering from 0. The TUPLEHASH functions are defined in pseudocode as follows:

Algorithm 9 $\text{TUPLEHASH}_{\langle s \rangle}(X, L, S)$

Validity Conditions: $s \in \{128, 256\}$, $0 \leq L < 2^{2040}$, $\text{len}(X[i]) < 2^{2040}$ for every i such that $0 \leq i < n$, and $\text{len}(S) < 2^{2040}$

```

1:  $z \leftarrow ""$ 
2:  $n \leftarrow$  the number of input strings in the tuple  $X$ 
3: for  $i = 0$  to  $n - 1$  do
4:    $z \leftarrow z \parallel \text{encode\_string}(X[i])$ 
5:  $\text{newX} \leftarrow z \parallel \text{right\_encode}(L)$ 
6: return  $\text{cSHAKE}_{\langle s \rangle}(\text{newX}, L, \text{"TupleHash"}, S)$ 

```

5.4. TUPLEHASHXOF Specification

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, the XOF variant of TUPLEHASH, called TUPLEHASHXOF, provides XOF behavior analogous to cSHAKE.

Conceptually, TUPLEHASHXOF produces an infinite-length output string, and the caller simply uses as many bits of the output string as are needed. Truncated outputs of TUPLEHASHXOF can be computed by the function $\text{TUPLEHASHXOF}_{\langle s \rangle}(X, L, S)$ given by the pseudocode below. TUPLEHASHXOF is identical to the ordinary version of TUPLEHASH except that the encoded output length is set to 0, regardless of the requested output length L , as shown in Step 5 of the pseudocode below.

Algorithm 10 $\text{TUPLEHASHXOF}_{\langle s \rangle}(X, L, S)$

Validity Conditions: $s \in \{128, 256\}$, $L \geq 0$, $\text{len}(X[i]) < 2^{2040}$ for every i such that $0 \leq i < n$, and $\text{len}(S) < 2^{2040}$

```

1:  $z \leftarrow ""$ 
2:  $n \leftarrow$  the number of input strings in the tuple  $X$ 
3: for  $i = 0$  to  $n - 1$  do
4:    $z \leftarrow z \parallel \text{encode\_string}(X[i])$ 
5:  $\text{newX} \leftarrow z \parallel \text{right\_encode}(0)$ 
6: return  $\text{cSHAKE}_{\langle s \rangle}(\text{newX}, L, \text{"TupleHash"}, S)$ 

```

As with cSHAKE, TUPLEHASHXOF is shown above as a single function call with the full input and output known at the time the call is made. TUPLEHASHXOF can also be accessed incrementally, as specified in Section 7. The streaming interface for TUPLEHASHXOF is slightly different than those of cSHAKE and KMACXOF in that each ABSORB call processes one complete tuple element.

6. PARALLELHASH

6.1. Overview

The purpose of PARALLELHASH⁷ 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 bitstring. It may be of any length, including zero.
- B is the block size in bytes that is used to partition X for parallel hashing. It may be any integer such that $0 < B < 2^{2040}$.
- L is an integer that represents the requested output length in bits.
- S is an optional customization bitstring of any length, including zero. If no customization is desired, S is set to the empty string.

6.3. PARALLELHASH Specification

PARALLELHASH divides the input bitstring X into a sequence of contiguous, non-overlapping blocks of at most B bytes, where every block except possibly the final block has a length of exactly B bytes. It 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", the optional customization string S , and some encoded integer values to generate the final hash value of the function.

The PARALLELHASH functions are defined in pseudocode as follows:

⁷A 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.

Algorithm 11 $\text{PARALLELHASH}_{\langle s \rangle}(X, B, L, S)$

Validity Conditions: $s \in \{128, 256\}$, $0 < B < 2^{2040}$, $\lceil \text{len}(X)/(8 \cdot B) \rceil < 2^{2040}$, $0 \leq L < 2^{2040}$, and $\text{len}(S) < 2^{2040}$

```

1:  $n \leftarrow \lceil \text{len}(X)/(8B) \rceil$ 
2:  $z \leftarrow \text{left\_encode}(B)$ 
3: for  $i = 0$  to  $n - 1$  do
4:    $P \leftarrow \text{substring}(X, 8iB, 8(i + 1)B)$ 
5:    $z \leftarrow z \parallel \text{cSHAKE}_{\langle s \rangle}(P, 2s, "", "")$ 
6:  $z \leftarrow z \parallel \text{right\_encode}(n) \parallel \text{right\_encode}(L)$ 
7: return  $\text{cSHAKE}_{\langle s \rangle}(z, L, \text{"ParallelHash"}, S)$ 
```

6.4. PARALLELHASHXOF Specification

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, the XOF variant of PARALLELHASH, called PARALLELHASHXOF, provides XOF behavior analogous to cSHAKE.

Conceptually, PARALLELHASHXOF produces an infinite-length output string, and the caller simply uses as many bits of the output string as needed. Truncated outputs of PARALLELHASHXOF can be computed by the function $\text{PARALLELHASHXOF}_{\langle s \rangle}(X, B, L, S)$ given by the pseudocode below. PARALLELHASHXOF is identical to the ordinary version of PARALLELHASH except that the encoded output length is set to 0, regardless of the requested output length L , as shown in Step 6 of the pseudocode below.

Algorithm 12 $\text{PARALLELHASHXOF}_{\langle s \rangle}(X, B, L, S)$

Validity Conditions: $s \in \{128, 256\}$, $0 < B < 2^{2040}$, $\lceil \text{len}(X)/(8B) \rceil < 2^{2040}$, $L \geq 0$, and $\text{len}(S) < 2^{2040}$

```

1:  $n \leftarrow \left\lceil \frac{\text{len}(X)}{8B} \right\rceil$ 
2:  $z \leftarrow \text{left\_encode}(B)$ 
3: for  $i = 0$  to  $n - 1$  do
4:    $P \leftarrow \text{substring}(X, 8iB, 8(i + 1)B)$ 
5:    $z \leftarrow z \parallel \text{cSHAKE}_{\langle s \rangle}(P, 2s, "", "")$ 
6:  $z \leftarrow z \parallel \text{right\_encode}(n) \parallel \text{right\_encode}(0)$ 
7: return  $\text{cSHAKE}_{\langle s \rangle}(z, L, \text{"ParallelHash"}, S)$ 
```

As with cSHAKE, PARALLELHASHXOF is specified above as a single function call. Section 7 specifies streaming access to PARALLELHASHXOF in which portions of the input may be provided

535 incrementally. The boundaries between successive input portions need not coincide with the
536 B -byte PARALLELHASHXOF block boundaries.

7. XOF Access and Streaming Interfaces

This section specifies XOF access for cSHAKE, KMACXOF, TUPLEHASHXOF, and PARALLELHASHXOF, including a streaming interface for absorbing input and squeezing output incrementally. The interfaces specify the required input-output behavior but do not require any particular internal representation.

For cSHAKE, the output can be viewed as a single stream of bits. For fixed values of the function-name string N , the customization string S , and the input, requesting additional output extends the same stream. Consequently, multiple squeeze calls produce the same result as a single squeeze call that requests the same total number of output bits.

7.1. Interface Model

The streaming interface is defined in terms of a context object denoted by ctx that maintains the state of an ongoing computation across calls. The internal representation of ctx is not specified by this recommendation. A conforming implementation may use any representation of ctx , provided that the same sequence of non-context inputs will result in the same sequence of non-context outputs when the implementation performs a series of calls to the interface using its own representation of ctx .

For a function F , the streaming interface consists of three operations:

- **INIT**, which creates a new context for a particular function and its parameters
- **ABSORB**, which provides input to the context
- **SQUEEZE**, which returns output from the context

These operations can be written as:

$$\begin{aligned}\text{ctx} &\leftarrow F.\text{INIT}(\dots) \\ \text{ctx} &\leftarrow F.\text{ABSORB}(\text{ctx}, X) \\ \text{ctx}, Z &\leftarrow F.\text{SQUEEZE}(\text{ctx}, \ell)\end{aligned}$$

where X is an input string, ℓ is the requested number of output bits, and Z is the returned output string of length ℓ .

The interface has two phases: an absorbing phase followed by a squeezing phase. The absorbing operation may be called zero or more times before the first call to the squeezing operation. The squeezing operation may then be called one or more times. Once squeezing has begun, no additional input may be absorbed into the same context. An implementation that receives an absorb request after squeezing has begun **shall** signal an error and **shall not** modify the context as a result of that request.

The interpretation of an absorb call depends on the function. For cSHAKE and KMACXOF, successive absorb calls provide consecutive portions of a single input string. For TUPLEHASHXOF, the interface distinguishes individual tuple elements. For PARALLELHASHXOF, successive absorb calls provide consecutive portions of the input string, and the implementation partitions their concatenation into B -byte blocks. The function-specific behavior is specified in the following subsections.

7.2. Streaming Access to cSHAKE

Streaming access to cSHAKE is required to produce the same output stream as a single invocation of cSHAKE on the concatenation of all absorbed input strings. Specifically, if

$$X = X_1 \parallel X_2 \parallel \cdots \parallel X_n,$$

then a sequence of calls

```

ctx ← cSHAKE⟨s⟩.INIT( $N, S$ ),
ctx ← cSHAKE⟨s⟩.ABSORB(ctx,  $X_1$ ),
⋮
ctx ← cSHAKE⟨s⟩.ABSORB(ctx,  $X_n$ ),
( $ctx, Z_1$ ) ← cSHAKE⟨s⟩.SQUEEZE(ctx,  $\ell_1$ ),
⋮
( $ctx, Z_t$ ) ← cSHAKE⟨s⟩.SQUEEZE(ctx,  $\ell_t$ )

```

shall satisfy

$$Z_1 \parallel \cdots \parallel Z_t = \text{cSHAKE}\langle s \rangle(X, \sum_{i=1}^t \ell_i, N, S).$$

Algorithm 13 gives a reference procedure that satisfies this requirement. It is not intended to prescribe the internal representation of ctx. An implementation may maintain the underlying sponge state or use any other representation, provided that the requirements above are satisfied.

Algorithm 13 Reference procedure for streaming XOF access to cSHAKE $\langle s \rangle$

Validity conditions: $s \in \{128, 256\}$, $\text{len}(N) < 2^{2040}$, $\text{len}(S) < 2^{2040}$, and $\ell \geq 0$ for each squeeze call.

Structure ctx:

ctx.state: the 1600-bit Keccak state;
ctx.index: the current position within the rate portion of ctx.state; during absorption, it is the number of rate bits already filled, and during squeezing, it is the number of rate bits already returned;
ctx.suffix: the domain-separation suffix;
ctx.squeezing: a Boolean indicating whether the context is in the squeezing phase.

Initialization

```

1: function cSHAKE $\langle s \rangle$ .INIT( $N, S$ )
2:    $r \leftarrow 1600 - 2s$ 
3:   Create a new context ctx
4:   ctx.state  $\leftarrow 0^{1600}$ ; ctx.index  $\leftarrow 0$ ; ctx.squeezing  $\leftarrow$  false
5:   if  $N = ""$  and  $S = ""$  then
6:     ctx.suffix  $\leftarrow$  1111
7:   else
8:     ctx.suffix  $\leftarrow$  00
9:      $P \leftarrow \text{bytepad}(\text{encode\_string}(N) \parallel \text{encode\_string}(S), r/8)$ 
10:    ctx  $\leftarrow$  cSHAKE $\langle s \rangle$ .ABSORB(ctx,  $P$ )
11:  return ctx

```

Absorbing

```

1: function cSHAKE $\langle s \rangle$ .ABSORB(ctx,  $X$ )
2:    $r \leftarrow 1600 - 2s$ 
3:   if ctx.squeezing or ctx.index  $\notin \{0, \dots, r - 1\}$  then
4:     Signal an error and terminate
5:   position  $\leftarrow 0$ 
6:   while position  $< \text{len}(X)$  do
7:      $m \leftarrow \min(\text{len}(X) - \text{position}, r - \text{ctx.index})$ 
8:     str  $\leftarrow \text{substring}(X, \text{position}, \text{position} + m)$ 
9:      $B \leftarrow 0^{\text{ctx.index}} \parallel \text{str} \parallel 0^{1600 - \text{ctx.index} - m}$ 
10:    ctx.state  $\leftarrow \text{ctx.state} \oplus B$ 
11:    ctx.index  $\leftarrow \text{ctx.index} + m$ 
12:    position  $\leftarrow \text{position} + m$ 
13:    if ctx.index =  $r$  then
14:      ctx.state  $\leftarrow \text{KECCAK-p}[1600, 24](\text{ctx.state})$ 
15:      ctx.index  $\leftarrow 0$ 
16:  return ctx

```

Squeezing

```

1: function cSHAKE $\langle s \rangle$ .SQUEEZE(ctx,  $\ell$ )
2:    $r \leftarrow 1600 - 2s$ 
3:   if  $\ell < 0$  or ctx.index  $\notin \{0, \dots, r - 1\}$  then
4:     Signal an error and terminate
5:   if ctx.squeezing = false then
6:     ctx  $\leftarrow$  cSHAKE $\langle s \rangle$ .ABSORB(ctx, ctx.suffix)
7:      $P \leftarrow 1 \parallel 0^{(r - \text{ctx.index} - 2) \bmod r} \parallel 1$ 
8:     ctx  $\leftarrow$  cSHAKE $\langle s \rangle$ .ABSORB(ctx,  $P$ )
9:     ctx.squeezing  $\leftarrow$  true
10:  ToGo  $\leftarrow \ell$ 
11:   $Z \leftarrow ""$ 
12:  while ToGo  $> 0$  do
13:     $m \leftarrow \min(\text{ToGo}, r - \text{ctx.index})$ 
14:     $Z \leftarrow Z \parallel \text{substring}(\text{ctx.state}, \text{ctx.index}, \text{ctx.index} + m)$ 
15:    ctx.index  $\leftarrow \text{ctx.index} + m$ 
16:    ToGo  $\leftarrow \text{ToGo} - m$ 
17:    if ctx.index =  $r$  then
18:      ctx.state  $\leftarrow \text{KECCAK-p}[1600, 24](\text{ctx.state})$ 
19:      ctx.index  $\leftarrow 0$ 
20:  return (ctx,  $Z$ )

```

7.3. Streaming Access to SHAKE

Streaming access to SHAKE is required to produce the same output stream as a single invocation of SHAKE on the concatenation of all absorbed input strings. Specifically, if

$$X = X_1 \parallel X_2 \parallel \cdots \parallel X_n,$$

then a sequence of calls

$$\begin{aligned} \text{ctx} &\leftarrow \text{SHAKE}\langle s \rangle.\text{INIT}(), \\ \text{ctx} &\leftarrow \text{SHAKE}\langle s \rangle.\text{ABSORB}(\text{ctx}, X_1), \\ &\vdots \\ \text{ctx} &\leftarrow \text{SHAKE}\langle s \rangle.\text{ABSORB}(\text{ctx}, X_n), \\ (\text{ctx}, Z_1) &\leftarrow \text{SHAKE}\langle s \rangle.\text{SQUEEZE}(\text{ctx}, \ell_1), \\ &\vdots \\ (\text{ctx}, Z_t) &\leftarrow \text{SHAKE}\langle s \rangle.\text{SQUEEZE}(\text{ctx}, \ell_t) \end{aligned}$$

shall satisfy

$$Z_1 \parallel \cdots \parallel Z_t = \text{SHAKE}\langle s \rangle(X, \sum_{i=1}^t \ell_i).$$

Given that cSHAKE is defined to be equivalent to SHAKE when the strings N and S are empty, a streaming implementation for SHAKE can be derived from a streaming implementation for cSHAKE as follows:

1. $\text{SHAKE}\langle s \rangle.\text{INIT}() = \text{cSHAKE}\langle s \rangle.\text{INIT}("", "")$
2. $\text{SHAKE}\langle s \rangle.\text{ABSORB}(\text{ctx}, X) = \text{cSHAKE}\langle s \rangle.\text{ABSORB}(\text{ctx}, X)$
3. $\text{SHAKE}\langle s \rangle.\text{SQUEEZE}(\text{ctx}, \ell) = \text{cSHAKE}\langle s \rangle.\text{SQUEEZE}(\text{ctx}, \ell)$

Other approaches are also permissible as long as the requirements above are met.

7.4. Streaming Access to KMACXOF

In the fixed-output definition of KMAC, the requested output length L is encoded as part of the input to the underlying cSHAKE call. For XOF access, the encoded output length is set to zero by appending `right_encode(0)` to the input. Consequently, KMACXOF is domain-separated from the fixed-output KMAC and is not expected to produce the same output, even when the same key, input string, customization string, and total output length are used.

Successive calls to $\text{KMACXOF}\langle s \rangle.\text{ABSORB}$ provide consecutive portions of a single input string. Specifically, it is required that if

$$X = X_1 \parallel X_2 \parallel \cdots \parallel X_n,$$

615 then a sequence of calls

```

616         ctx ← KMACXOF⟨s⟩.INIT(K, S),
617         ctx ← KMACXOF⟨s⟩.ABSORB(ctx, X1),
618         ⋮
619         ctx ← KMACXOF⟨s⟩.ABSORB(ctx, Xn),
620         (ctx, Z1) ← KMACXOF⟨s⟩.SQUEEZE(ctx, ℓ1),
621         ⋮
622         (ctx, Zt) ← KMACXOF⟨s⟩.SQUEEZE(ctx, ℓt)

```

shall satisfy

$$Z_1 \parallel Z_2 \parallel \cdots \parallel Z_t = \text{KMACXOF}\langle s \rangle \left(K, X, \sum_{i=1}^t \ell_i, S \right).$$

623 Algorithm 14 gives a reference procedure that satisfies the requirements for streaming access to
624 KMACXOF. It is not intended to prescribe the internal representation of the context.

Algorithm 14 Reference procedure for streaming XOF access to KMACXOF⟨s⟩

Validity Conditions: $s \in \{128, 256\}$, $\text{len}(K) < 2^{2040}$, $\text{len}(S) < 2^{2040}$, and $\ell \geq 0$ for each squeeze call.

```

1: function KMACXOF⟨s⟩.INIT(K, S)
2:    $r \leftarrow 1600 - 2s$ 
3:    $X \leftarrow \text{bytepad}(\text{encode\_string}(K), r/8)$ 
4:    $\text{ctx} \leftarrow \text{cSHAKE}\langle s \rangle.\text{INIT}(\text{"KMAC"}, S)$ 
5:    $\text{ctx} \leftarrow \text{cSHAKE}\langle s \rangle.\text{ABSORB}(\text{ctx}, X)$ 
6:   return ctx

7: function KMACXOF⟨s⟩.ABSORB(ctx, X)
8:    $\text{ctx} \leftarrow \text{cSHAKE}\langle s \rangle.\text{ABSORB}(\text{ctx}, X)$ 
9:   return ctx

10: function KMACXOF⟨s⟩.SQUEEZE(ctx, ℓ)
11:   if ctx.squeezing = false then
12:      $\text{ctx} \leftarrow \text{cSHAKE}\langle s \rangle.\text{ABSORB}(\text{ctx}, \text{right\_encode}(0))$ 
13:    $(\text{ctx}, Z) \leftarrow \text{cSHAKE}\langle s \rangle.\text{SQUEEZE}(\text{ctx}, \ell)$ 
14:   return (ctx, Z)

```

The first call to $\text{KMACXOF}\langle s \rangle.\text{SQUEEZE}$ appends $\text{right_encode}(0)$ to the absorbed input and transitions the context to the squeezing phase. Once squeezing has begun, no additional input may be absorbed.

7.5. Streaming Access to TUPLEHASHXOF

In the fixed-output definition of TUPLEHASH, the requested output length L is encoded as part of the input to the underlying cSHAKE call. For XOF access, the encoded output length is set to zero by appending $\text{right_encode}(0)$ to the input. Consequently, TUPLEHASHXOF is domain-separated from the fixed-output TUPLEHASH and is not expected to produce the same output, even when the same input tuple, customization string, and total output length are used.

Successive calls to $\text{TUPLEHASHXOF}\langle s \rangle.\text{ABSORB}$ provide the individual elements of the input tuple. Each call supplies one complete tuple element, which is encoded using encode_string before it is absorbed into the underlying cSHAKE context.

Specifically, if

$$X = (X_0, X_1, \dots, X_{n-1}),$$

then a sequence of calls

$$\begin{aligned} \text{ctx} &\leftarrow \text{TUPLEHASHXOF}\langle s \rangle.\text{INIT}(S), \\ \text{ctx} &\leftarrow \text{TUPLEHASHXOF}\langle s \rangle.\text{ABSORB}(\text{ctx}, X_0), \\ &\vdots \\ \text{ctx} &\leftarrow \text{TUPLEHASHXOF}\langle s \rangle.\text{ABSORB}(\text{ctx}, X_{n-1}), \\ (\text{ctx}, Z_1) &\leftarrow \text{TUPLEHASHXOF}\langle s \rangle.\text{SQUEEZE}(\text{ctx}, \ell_1), \\ &\vdots \\ (\text{ctx}, Z_t) &\leftarrow \text{TUPLEHASHXOF}\langle s \rangle.\text{SQUEEZE}(\text{ctx}, \ell_t) \end{aligned}$$

shall satisfy

$$Z_1 \parallel Z_2 \parallel \dots \parallel Z_t = \text{TUPLEHASHXOF}\langle s \rangle \left(X, \sum_{i=1}^t \ell_i, S \right).$$

A call to $\text{TUPLEHASHXOF}\langle s \rangle.\text{ABSORB}(\text{ctx}, X)$ represents one complete tuple element. Thus, two absorb calls with inputs X_0 and X_1 represent the two-element tuple (X_0, X_1) rather than the one-element tuple $(X_0 \parallel X_1)$.

Algorithm 15 gives a reference procedure that satisfies the requirements for streaming access to TUPLEHASHXOF. It is not intended to prescribe the internal representation of the context.

Algorithm 15 Reference procedure for streaming XOF access to $\text{TUPLEHASHXOF}\langle s \rangle$

Validity Conditions: $s \in \{128, 256\}$, $\text{len}(S) < 2^{2040}$, and $\ell \geq 0$ for each squeeze call.

```

1: function  $\text{TUPLEHASHXOF}\langle s \rangle.\text{INIT}(S)$ 
2:    $\text{ctx} \leftarrow \text{cSHAKE}\langle s \rangle.\text{INIT}(\text{"TupleHash"}, S)$ 
3:   return  $\text{ctx}$ 

4: function  $\text{TUPLEHASHXOF}\langle s \rangle.\text{ABSORB}(\text{ctx}, X)$ 
   Validity Conditions:  $\text{len}(X) < 2^{2040}$ .
5:    $\text{ctx} \leftarrow \text{cSHAKE}\langle s \rangle.\text{ABSORB}(\text{ctx}, \text{encode\_string}(X))$ 
6:   return  $\text{ctx}$ 

7: function  $\text{TUPLEHASHXOF}\langle s \rangle.\text{SQUEEZE}(\text{ctx}, \ell)$ 
8:   if  $\text{ctx.squeezing} = \text{false}$  then
9:      $\text{ctx} \leftarrow \text{cSHAKE}\langle s \rangle.\text{ABSORB}(\text{ctx}, \text{right\_encode}(0))$ 
10:   $(\text{ctx}, Z) \leftarrow \text{cSHAKE}\langle s \rangle.\text{SQUEEZE}(\text{ctx}, \ell)$ 
11:  return  $(\text{ctx}, Z)$ 

```

650 The first call to $\text{TUPLEHASHXOF}\langle s \rangle.\text{SQUEEZE}$ appends $\text{right_encode}(0)$ to the encoded tuple
651 and transitions the context to the squeezing phase. Once squeezing has begun, no additional
652 tuple elements may be absorbed.

653 7.6. Streaming Access to PARALLELHASHXOF

654 In the fixed-output definition of PARALLELHASH , the requested output length L is encoded as
655 part of the input to the underlying cSHAKE call. For XOF access, the encoded output length
656 is set to zero by appending $\text{right_encode}(0)$ to the input. Consequently, PARALLELHASHXOF
657 is domain-separated from the fixed-output PARALLELHASH and is not expected to produce the
658 same output, even when the same input string, block size, customization string, and total output
659 length are used.

660 Successive calls to $\text{PARALLELHASHXOF}\langle s \rangle.\text{ABSORB}$ provide consecutive portions of a single
661 input string. The boundaries between absorb calls do not determine the B -byte blocks used by
662 PARALLELHASH . Instead, the concatenation of all absorbed strings is divided into consecutive
663 blocks of at most B bytes, as specified in Section 6.4.

Specifically, if

$$X = X_1 \parallel X_2 \parallel \cdots \parallel X_n,$$

664 then a sequence of calls

```
665         ctx ← PARALLELHASHXOF⟨s⟩.INIT(B, S),
666         ctx ← PARALLELHASHXOF⟨s⟩.ABSORB(ctx, X1),
667         ⋮
668         ctx ← PARALLELHASHXOF⟨s⟩.ABSORB(ctx, Xn),
669         (ctx, Z1) ← PARALLELHASHXOF⟨s⟩.SQUEEZE(ctx, ℓ1),
670         ⋮
671         (ctx, Zt) ← PARALLELHASHXOF⟨s⟩.SQUEEZE(ctx, ℓt)
```

672 **shall** satisfy

$$Z_1 \parallel \cdots \parallel Z_t = \text{PARALLELHASHXOF}\langle s \rangle \left(X, B, \sum_{i=1}^t \ell_i, S \right).$$

674 Algorithm 16 gives a reference procedure that satisfies the requirements for streaming access to
675 PARALLELHASHXOF. It is not intended to prescribe the internal representation of the context.

Algorithm 16 Reference procedure for streaming XOF access to $\text{PARALLELHASHXOF}\langle s \rangle$

Validity Conditions: $s \in \{128, 256\}$, $0 < B < 2^{2040}$, $\text{len}(S) < 2^{2040}$, and $\ell \geq 0$ for each squeeze call.

Structure ctx

outer: a context for $\text{cSHAKE}\langle s \rangle$
buffer: bitstring
blockSize: positive integer
blockCount: non-negative integer
squeezing: Boolean

```

1: function  $\text{PARALLELHASHXOF}\langle s \rangle.\text{INIT}(B, S)$ 
2:   Create a new context  $\text{ctx}$ 
3:    $\text{ctx.out} \leftarrow \text{cSHAKE}\langle s \rangle.\text{INIT}(\text{"ParallelHash"}, S)$ 
4:    $\text{ctx.out} \leftarrow \text{cSHAKE}\langle s \rangle.\text{ABSORB}(\text{ctx.out}, \text{left\_encode}(B))$ 
5:    $\text{ctx.buffer} \leftarrow \text{" "}$ 
6:    $\text{ctx.blockSize} \leftarrow B$ 
7:    $\text{ctx.blockCount} \leftarrow 0$ 
8:    $\text{ctx.squeezing} \leftarrow \text{false}$ 
9:   return  $\text{ctx}$ 

10: function  $\text{PARALLELHASHXOF}\langle s \rangle.\text{PROCESS\_BLOCK}(\text{ctx}, P)$ 
    Validity Condition:  $0 < \text{len}(P) \leq 8\text{ctx.blockSize}$ 
11:   if  $\text{ctx.blockCount} = 2^{2040} - 1$  then
12:     Signal an error and terminate
13:    $Y \leftarrow \text{cSHAKE}\langle s \rangle(P, 2s, \text{" "}, \text{" "})$ 
14:    $\text{ctx.out} \leftarrow \text{cSHAKE}\langle s \rangle.\text{ABSORB}(\text{ctx.out}, Y)$ 
15:    $\text{ctx.blockCount} \leftarrow \text{ctx.blockCount} + 1$ 
16:   return  $\text{ctx}$ 

17: function  $\text{PARALLELHASHXOF}\langle s \rangle.\text{ABSORB}(\text{ctx}, X)$ 
18:   if  $\text{ctx.squeezing} = \text{true}$  then
19:     Signal an error and terminate
20:    $\text{ctx.buffer} \leftarrow \text{ctx.buffer} \parallel X$ 
21:   while  $\text{len}(\text{ctx.buffer}) \geq 8 \cdot \text{ctx.blockSize}$  do
22:      $P \leftarrow \text{substring}(\text{ctx.buffer}, 0, 8\text{ctx.blockSize})$ 
23:      $\text{ctx.buffer} \leftarrow \text{substring}(\text{ctx.buffer}, 8\text{ctx.blockSize}, \text{len}(\text{ctx.buffer}))$ 
24:      $\text{ctx} \leftarrow \text{PARALLELHASHXOF}\langle s \rangle.\text{process\_block}(\text{ctx}, P)$ 
25:   return  $\text{ctx}$ 

26: function  $\text{PARALLELHASHXOF}\langle s \rangle.\text{SQUEEZE}(\text{ctx}, \ell)$ 
27:   if  $\text{ctx.squeezing} = \text{false}$  then
28:     if  $\text{len}(\text{ctx.buffer}) > 0$  then
29:        $\text{ctx} \leftarrow \text{PARALLELHASHXOF}\langle s \rangle.\text{process\_block}(\text{ctx}, \text{ctx.buffer})$ 
30:        $\text{ctx.buffer} \leftarrow \text{" "}$ 
31:        $T \leftarrow \text{right\_encode}(\text{ctx.blockCount}) \parallel \text{right\_encode}(0)$ 
32:        $\text{ctx.out} \leftarrow \text{cSHAKE}\langle s \rangle.\text{ABSORB}(\text{ctx.out}, T)$ 
33:        $\text{ctx.squeezing} \leftarrow \text{true}$ 
34:        $(\text{ctx.out}, Z) \leftarrow \text{cSHAKE}\langle s \rangle.\text{squeeze}(\text{ctx.out}, \ell)$ 
35:       return  $(\text{ctx}, Z)$ 

```

676 During the absorbing phase, each complete B -byte block is processed separately by using cSHAKE.
677 The resulting $2s$ -bit string is absorbed into an outer cSHAKE context. Any input that remains
678 after the last complete block is retained in the context until more input is provided or the first
679 squeeze call is made.

680 The first call to `PARALLELHASHXOF $\langle s \rangle$ .SQUEEZE` processes any remaining nonempty partial
681 block, appends the number of processed blocks using `right_encode`, and appends `right_encode(0)`
682 to select XOF access. The context then transitions to the squeezing phase. Once squeezing has
683 begun, no additional input may be absorbed.

8. Implementation Considerations

8.1. Precomputation

cSHAKE is defined so that all calls to the underlying KECCAK- f function [1] for accommodating 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.

8.2. Limited Implementations

The cSHAKE, KMAC, TUPLEHASH, and PARALLELHASH functions are defined to accept a wide range of possible inputs (e.g., unreasonably long inputs, inputs that involve 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, 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 that 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 **shall not** produce an output.

8.3. 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.

716 PARALLELHASH can be implemented in a straightforward and reasonably efficient way even when
717 only sequential processing is available. However, a much faster implementation is possible when
718 each of the individual blocks of the message can be handled in parallel. The choice of block size
719 B can significantly affect this performance. PARALLELHASH is designed so that any machine that
720 can apply parallel processing can, in principle, benefit from that parallel processing. For example,
721 a machine that can hash four blocks in parallel and a machine that can hash 32 blocks in parallel
722 can each benefit from all of the parallel processing ability that is available.

9. Security Considerations

9.1. 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 up to 128 bits, subject to the output-length considerations in Section 9.3. 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.

9.2. Security Properties for the Function Name and Customization Strings

9.2.1. Equivalent Security to SHAKE for Any Legal N and S

For any valid function name string N and the customization string S , cSHAKE128(X, L, N, S) and cSHAKE256(X, L, N, S) are intended to provide the same claimed security strengths as SHAKE128(X, L) and SHAKE256(X, L), respectively. There are no known weak values for N or S .

9.2.2. Domain Separation Using Function Name or Customization Strings

Suppose that (n_1, s_1) and (n_2, s_2) are two function-name and customization-string pairs, and either $n_1 \neq n_2$ or $s_1 \neq s_2$. The corresponding cSHAKE invocations are domain-separated. In particular, for requested output lengths q_1 and q_2 , the functions cSHAKE(X, q_1, n_1, s_1) and cSHAKE(X, q_2, n_2, s_2) may be treated as distinct functions of X with output lengths q_1 and q_2 , respectively. This means, for example, that:

- Keys k_1 and k_2 , where $k_1 = \text{cSHAKE}(x_1, q_1, n_1, s_1)$, and $k_2 = \text{cSHAKE}(x_2, q_2, n_2, s_2)$, and both keys are derived from secret but related quantities x_1 and x_2 , 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 $\text{cSHAKE128}(x_1, L, n_1, s_1) = \text{cSHAKE128}(x_2, L, n_2, s_2)$ is expected to require a computational complexity on the order of $\min(2^{L/2}, 2^{128})$. Similarly, finding a collision such that $\text{cSHAKE256}(x_1, L, n_1, s_1) = \text{cSHAKE256}(x_2, L, n_2, s_2)$ is expected to 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 that are 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 $s_1 \neq s_2$, $\text{PARALLELHASH}(X, B, L, s_1)$ and $\text{PARALLELHASH}(X, B, L, s_2)$ are expected to have no particular relationship.

Except when used in XOF mode, KMAC, TUPLEHASH, and PARALLELHASH have the additional property that instances of the functions with different output lengths can be treated as unrelated functions of their remaining inputs. For example, $\text{PARALLELHASH}(X, B, q_1, S)$ and $\text{PARALLELHASH}(X, B, q_2, S)$ can be treated as independent hash functions with input X and output lengths q_1 and q_2 , respectively.

cSHAKE does not share this property. For $q_1 < q_2$, $\text{cSHAKE}(X, q_1, N, S)$ is a prefix of $\text{cSHAKE}(X, q_2, N, S)$. 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.

9.3. Collision and Preimage Resistance

All of 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 generally 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.

9.4. Guidelines 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.

9.4.1. 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 message-tag 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. Guidelines for cryptographic algorithm and key-size selection are available in SP 800-57 [8].

9.4.2. 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 make approximately 2^L forgery attempts 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 that is less than 64 bits after a careful risk analysis is performed.

To illustrate the security properties of KMAC for given parameter settings, Table 2 lists a few other approved MAC algorithms along with equivalent settings for KMAC as an example. 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 equivalent settings of different MAC algorithms do not result in the same output.

Table 2: KMAC parameter choices corresponding to selected MAC algorithms. In each call, K denotes the secret key, and S denotes the KMAC customization string, which may be set to the empty string when no customization is desired.

Selected 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)

9.5. Security of Sponge Constructions

The sponge construction underlies KECCAK and, thus, all KECCAK-derived functions. Generic security of the sponge construction has been studied extensively. While a full review of the literature on this topic is beyond the scope of this document, useful starting points include [5, 9, 10].

As cryptographic standards move toward quantum-secure algorithms, the security of these constructions must be considered in the context of quantum computers. In this setting, the underlying permutation is public, and superposition queries can be made on it without knowledge of any secret information, which contrasts with most symmetric cryptographic primitives. The quantum security of the sponge construction is studied in [11–13].

813 A number of works have also analyzed specific security properties of generic sponge constructions.
814 The security of keyed sponges (e.g., KMAC) is discussed in [14]. A more detailed analysis of
815 sponge constructions' security against collisions is discussed in [15, 16]. Preimage resistance is
816 discussed in [17, 18].

References

- [1] National Institute of Standards and Technology (2015) SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions (National Institute of Standards and Technology, Gaithersburg, MD), Federal Information Processing Standards Publication (FIPS PUB) 202. DOI:[10.6028/NIST.FIPS.202](https://doi.org/10.6028/NIST.FIPS.202). Available at <http://dx.doi.org/10.6028/NIST.FIPS.202>
- [2] Bertoni G, Daemen J, Peeters M, Van Assche G (2013) Keccak. *Advances in Cryptology – EUROCRYPT 2013*, eds Johansson T, Nguyen PQ (Springer, Berlin, Heidelberg), *Lecture Notes in Computer Science*, Vol. 7881, pp 313–314. DOI:[10.1007/978-3-642-38348-9_19](https://doi.org/10.1007/978-3-642-38348-9_19)
- [3] Bertoni G, Daemen J, Peeters M, Van Assche G (2011) The Keccak SHA-3 submission, version 3, <https://keccak.team/files/Keccak-submission-3.pdf>.
- [4] Bertoni G, Daemen J, Peeters M, Van Assche G (2011) The KECCAK reference, Version 3.0. Accessed: 2016-12-21 Available at <http://keccak.noekeon.org/Keccak-reference-3.0.pdf>.
- [5] Bertoni G, Daemen J, Peeters M, Van Assche G (2008) On the indistinguishability of the sponge construction. *Advances in Cryptology – EUROCRYPT 2008*, ed Smart NP (Springer, Berlin, Heidelberg), *Lecture Notes in Computer Science*, Vol. 4965, pp 181–197. DOI:[10.1007/978-3-540-78967-3_11](https://doi.org/10.1007/978-3-540-78967-3_11)
- [6] Kelsey J, Chang Sj, Perlner R (2016) SHA-3 derived functions: cSHAKE, KMAC, TupleHash, and ParallelHash (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication 800-185. DOI:[10.6028/NIST.SP.800-185](https://doi.org/10.6028/NIST.SP.800-185)
- [7] Bertoni G, Daemen J, Peeters M, Van Assche G (2011) Cryptographic sponge functions, Version 0.1. Accessed: 2016-12-21 Available at <http://sponge.noekeon.org/CSF-0.1.pdf>.
- [8] Barker E (2020) Recommendation for Key Management, Part 1: General (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-57 Part 1 Revision 5. DOI:[10.6028/NIST.SP.800-57pt1r5](https://doi.org/10.6028/NIST.SP.800-57pt1r5). Available at <http://dx.doi.org/10.6028/NIST.SP.800-57pt1r5>
- [9] Lefevre C (2023) Indistinguishability of the sponge construction with a restricted number of message blocks. *IACR Transactions on Symmetric Cryptology* 2023(1):224–243. DOI:[10.46586/tosc.v2023.i1.224-243](https://doi.org/10.46586/tosc.v2023.i1.224-243). Available at <https://doi.org/10.46586/tosc.v2023.i1.224-243>
- [10] Li H, Li C, Hu H, Liu F (2026) Indistinguishability of the sponge hash family: New attacks and the optimal construction. *IACR Transactions on Symmetric Cryptology* 2026(1):31–75. DOI:[10.46586/tosc.v2026.i1.31-75](https://doi.org/10.46586/tosc.v2026.i1.31-75). Available at <https://doi.org/10.46586/tosc.v2026.i1.31-75>
- [11] Czajkowski J, Bruinderink LG, Hülsing A, Schaffner C, Unruh D (2018) Post-quantum security of the sponge construction. *Post-Quantum Cryptography – PQCrypto 2018*, eds Lange T, Steinwandt R (Springer, Cham), *Lecture Notes in Computer Science*, Vol. 10786, pp 185–204. DOI:[10.1007/978-3-319-79063-3_9](https://doi.org/10.1007/978-3-319-79063-3_9). Available at https://doi.org/10.1007/978-3-319-79063-3_9

- [12] Czajkowski J, Hülsing A, Schaffner C (2019) Quantum indistinguishability of random sponges. *Advances in Cryptology – CRYPTO 2019, Part II*, eds Boldyreva A, Micciancio D (Springer, Cham), *Lecture Notes in Computer Science*, Vol. 11693, pp 296–325. DOI:[10.1007/978-3-030-26951-7_11](https://doi.org/10.1007/978-3-030-26951-7_11). Available at https://doi.org/10.1007/978-3-030-26951-7_11
- [13] Alagic G, Carolan J, Majenz C, Tokat S (2025) The sponge is quantum indifferentiable. *2025 IEEE 66th Annual Symposium on Foundations of Computer Science (FOCS)* (IEEE), pp 2591–2630. DOI:[10.1109/FOCS63196.2025.00135](https://doi.org/10.1109/FOCS63196.2025.00135). Available at <https://doi.org/10.1109/FOCS63196.2025.00135>
- [14] Andreeva E, Daemen J, Mennink B, Van Assche G (2015) Security of keyed sponge constructions using a modular proof approach. *Fast Software Encryption – FSE 2015*, ed Leander G (Springer, Berlin, Heidelberg), *Lecture Notes in Computer Science*, Vol. 9054, pp 364–384. DOI:[10.1007/978-3-662-48116-5_18](https://doi.org/10.1007/978-3-662-48116-5_18)
- [15] Lefevre C, Mennink B (2024) Permutation-based hashing beyond the birthday bound. *IACR Transactions on Symmetric Cryptology* 2024(1):71–113. DOI:[10.46586/tosc.v2024.i1.71-113](https://doi.org/10.46586/tosc.v2024.i1.71-113). Available at <https://doi.org/10.46586/tosc.v2024.i1.71-113>
- [16] Freitag C, Ghoshal A, Komargodski I (2026) Time-space tradeoffs for sponge hashing: Attacks and limitations for short collisions. *Journal of Cryptology* 39:9. DOI:[10.1007/s00145-025-09558-1](https://doi.org/10.1007/s00145-025-09558-1). Article 9 Available at <https://doi.org/10.1007/s00145-025-09558-1>
- [17] Lefevre C, Mennink B (2022) Tight preimage resistance of the sponge construction. *Advances in Cryptology – CRYPTO 2022, Part IV*, eds Dodis Y, Shrimpton T (Springer, Cham), *Lecture Notes in Computer Science*, Vol. 13510, pp 185–204. DOI:[10.1007/978-3-031-15985-5_7](https://doi.org/10.1007/978-3-031-15985-5_7). Available at https://doi.org/10.1007/978-3-031-15985-5_7
- [18] Carolan J, Poremba A (2024) Quantum one-wayness of the single-round sponge with invertible permutations. *Advances in Cryptology – CRYPTO 2024, Part VI*, eds Reyzin L, Stebila D (Springer, Cham), *Lecture Notes in Computer Science*, Vol. 14925, pp 218–252. DOI:[10.1007/978-3-031-68391-6_7](https://doi.org/10.1007/978-3-031-68391-6_7). Available at https://doi.org/10.1007/978-3-031-68391-6_7

Appendix A KMAC, TUPLEHASH, and PARALLELHASH in Terms of KECCAK_[c]

FIPS 202 [1] 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, KMACXOF, TUPLEHASH, TUPLEHASHXOF, PARALLELHASH, and PARALLELHASHXOF are defined directly in terms of KECCAK_[c]. These definitions are exactly equivalent to the definitions made in terms of cSHAKE in Sec. 4, 5, and 6.

Algorithm 17 KMAC_[s](K, X, L, S) in terms of KECCAK_[c]

Validity Conditions: $s \in \{128, 256\}$, $\text{len}(K) < 2^{2040}$, $0 \leq L < 2^{2040}$, and $\text{len}(S) < 2^{2040}$

- 1: $r \leftarrow 1600 - 2s$
 - 2: $\text{newX} \leftarrow \text{bytepad}(\text{encode_string}(K), r/8) \parallel X \parallel \text{right_encode}(L)$
 - 3: $T \leftarrow \text{bytepad}(\text{encode_string}(\text{"KMAC"}), r/8) \parallel \text{encode_string}(S), r/8)$
 - 4: **return** KECCAK_[2s]($T \parallel \text{newX} \parallel 00, L$)
-

Algorithm 18 KMACXOF_[s](K, X, L, S) in terms of KECCAK_[c]

Validity Conditions: $s \in \{128, 256\}$, $\text{len}(K) < 2^{2040}$, $L \geq 0$, and $\text{len}(S) < 2^{2040}$

- 1: $r \leftarrow 1600 - 2s$
 - 2: $\text{newX} \leftarrow \text{bytepad}(\text{encode_string}(K), r/8) \parallel X \parallel \text{right_encode}(0)$
 - 3: $T \leftarrow \text{bytepad}(\text{encode_string}(\text{"KMAC"}), r/8) \parallel \text{encode_string}(S), r/8)$
 - 4: **return** KECCAK_[2s]($T \parallel \text{newX} \parallel 00, L$)
-

Algorithm 19 TUPLEHASH_[s](X, L, S) in terms of KECCAK_[c]

Validity Conditions: $s \in \{128, 256\}$, $0 \leq L < 2^{2040}$, $\text{len}(X[i]) < 2^{2040}$ for every i such that $0 \leq i < n$, and $\text{len}(S) < 2^{2040}$

- 1: $r \leftarrow 1600 - 2s$
 - 2: $z \leftarrow \text{" "}$
 - 3: $n \leftarrow$ the number of input strings in the tuple X
 - 4: **for** $i = 0$ **to** $n - 1$ **do**
 - 5: $z \leftarrow z \parallel \text{encode_string}(X[i])$
 - 6: $\text{newX} \leftarrow z \parallel \text{right_encode}(L)$
 - 7: $T \leftarrow \text{bytepad}(\text{encode_string}(\text{"TupleHash"}), r/8) \parallel \text{encode_string}(S), r/8)$
 - 8: **return** KECCAK_[2s]($T \parallel \text{newX} \parallel 00, L$)
-

Algorithm 20 TUPLEHASHOF $\langle s \rangle(X, L, S)$ in terms of KECCAK $[c]$

Validity Conditions: $s \in \{128, 256\}$, $L \geq 0$, $\text{len}(X[i]) < 2^{2040}$ for every i such that $0 \leq i < n$,
and $\text{len}(S) < 2^{2040}$

- 1: $r \leftarrow 1600 - 2s$
 - 2: $z \leftarrow ""$
 - 3: $n \leftarrow$ the number of input strings in the tuple X
 - 4: **for** $i = 0$ **to** $n - 1$ **do**
 - 5: $z \leftarrow z \parallel \text{encode_string}(X[i])$
 - 6: $\text{newX} \leftarrow z \parallel \text{right_encode}(0)$
 - 7: $T \leftarrow \text{bytepad}(\text{encode_string}(\text{"TupleHash"}) \parallel \text{encode_string}(S), r/8)$
 - 8: **return** KECCAK $[2s](T \parallel \text{newX} \parallel 00, L)$
-

Algorithm 21 PARALLELHASH $\langle s \rangle(X, B, L, S)$ in terms of KECCAK $[c]$

Validity Conditions: $s \in \{128, 256\}$, $0 < B < 2^{2040}$, $\lceil \text{len}(X)/(8B) \rceil < 2^{2040}$, $0 \leq L < 2^{2040}$,
and $\text{len}(S) < 2^{2040}$

- 1: $r \leftarrow 1600 - 2s$
 - 2: $n \leftarrow \left\lceil \frac{\text{len}(X)}{8B} \right\rceil$
 - 3: $z \leftarrow \text{left_encode}(B)$
 - 4: **for** $i = 0$ **to** $n - 1$ **do**
 - 5: $P \leftarrow \text{substring}(X, 8iB, 8(i + 1)B)$
 - 6: $z \leftarrow z \parallel \text{KECCAK}[2s](P \parallel 1111, 2s)$
 - 7: $z \leftarrow z \parallel \text{right_encode}(n) \parallel \text{right_encode}(L)$
 - 8: $T \leftarrow \text{bytepad}(\text{encode_string}(\text{"ParallelHash"}) \parallel \text{encode_string}(S), r/8)$
 - 9: **return** KECCAK $[2s](T \parallel z \parallel 00, L)$
-

Algorithm 22 $\text{PARALLELHASHXOF}_{\langle s \rangle}(X, B, L, S)$ in terms of $\text{KECCAK}[c]$

Validity Conditions: $s \in \{128, 256\}$, $0 < B < 2^{2040}$, $\lceil \text{len}(X)/(8B) \rceil < 2^{2040}$, $L \geq 0$, and $\text{len}(S) < 2^{2040}$

```
1:  $r \leftarrow 1600 - 2s$ 
2:  $n \leftarrow \left\lceil \frac{\text{len}(X)}{8B} \right\rceil$ 
3:  $z \leftarrow \text{left\_encode}(B)$ 
4: for  $i = 0$  to  $n - 1$  do
5:    $P \leftarrow \text{substring}(X, 8iB, 8(i + 1)B)$ 
6:    $z \leftarrow z \parallel \text{KECCAK}[2s](P \parallel 1111, 2s)$ 
7:  $z \leftarrow z \parallel \text{right\_encode}(n) \parallel \text{right\_encode}(0)$ 
8:  $T \leftarrow \text{bytepad}(\text{encode\_string}(\text{"ParallelHash"}) \parallel \text{encode\_string}(S), r/8)$ 
9: return  $\text{KECCAK}[2s](T \parallel z \parallel 00, L)$ 
```

Appendix B Hashing Into a Range

XOFs, PRFs, and hash functions with variable-length output (e.g., cSHAKE, KMAC, TUPLEHASH, PARALLELHASH) can easily be used to generate an integer N within the range $0 \leq N < R$ 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 of $\{0, 1, 2, \dots, R - 1\}$, do the following:

1. Let $k = \lceil \log_2(R) \rceil + 128$.
2. Call the hash function with a requested length of at least k bits. Let the resulting bitstring be Z .
3. Let $N = \text{bits_to_integer}(Z) \bmod R$, 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 $\{0, 1, 2, \dots, R - 1\}$. For any possible output value t such that $0 \leq t < R$, the following statement is true⁸:

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

In other words, the output of this process will have a very small bias. No value will be 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 $\{0, 1, \dots, R - 1\}$.

The `bits_to_integer` function converts a bitstring to an integer as follows:

Algorithm 23 `bits_to_integer( $b_0, b_1, \dots, b_{n-1}$ )`

- 1: Interpret $b_0, b_1, \dots, b_{n-1}$ as the bits of a bitstring, ordered from lowest-order bit to highest-order bit.
 - 2: $x = \sum_{i=0}^{n-1} 2^i b_i$
 - 3: **return** x .
-

⁸In fact, the bound is slightly tighter than this. If w = the length of the bitstring Z in bits ($w \geq \lceil \log_2(R) \rceil + 128$), then $|\text{Prob}(N = t) - 1/R| \leq 2^{-w}$.

Appendix C Summary of Changes

Revision 1 makes the following changes to SP 800-185:

- **Adds streaming interfaces for XOF access.** A new section specifies init, absorb, and squeeze interfaces for incremental input processing and output generation. Streaming access is specified for cSHAKE and SHAKE and for the XOF forms of KMAC, TUPLEHASH, and PARALLELHASH.
- **Clarifies the distinction between variable-length output and XOF access.** The revision explains that fixed-output KMAC, TUPLEHASH, and PARALLELHASH encode the requested output length as part of the input, whereas the encoded output length is set to zero in XOF mode.
- **Corrects the TUPLEHASH specification.** The indexing of tuple elements is updated to begin at zero, consistent with the definition of $X[i]$. The original publication defined $X[i]$ as numbered from zero but iterated from $i = 1, \dots, n$, resulting in an indexing inconsistency. Additionally, validity conditions are added for the lengths of individual tuple elements.
- **Corrects the PARALLELHASH input length validity condition.** The limit on the number of PARALLELHASH blocks is expressed in terms of the number of B-byte blocks, $\lceil \text{len}(X)/(8B) \rceil < 2^{2040}$. The original publication instead used $\lceil \text{len}(X)/(B) \rceil < 2^{2040}$, even though B is measured in bytes and len is measured in bits.
- **Merges algorithm specifications.** The original publication separately specified 128-bit and 256-bit variants. Revision 1 uses parameterized pseudocode based on the security-strength parameter s . The notation $F\langle s \rangle$, where $s \in \{128, 256\}$, is used.
- **Updates the bit-ordering conversion in Appendix B.** Algorithm 23 is revised so that the ordering of the input bits is consistent with the bit-ordering conventions used throughout this recommendation.
- **Makes editorial and organizational revisions throughout the document,** including updated terminology, notation, pseudocode formatting, and explanatory text.