Hash Algorithms
Hash Algorithms

Driven by the slowness of RSA in signing a message. The idea was to create (relatively fast) a digest of a message and sign that. This was the origin of MD and MD2 algorithms by Ron Rivest in 1989.

Merkle developed SNEFRU in 1990. It was many times faster than MD2.

This prompted Rivest in 1990 to create MD4 which exploited microprocessor operations on newer chips.

SNEFRU was cracked in 1992 by Shamir.

A variant of MD4 was cracked in 1991.


MD4 was severely cracked in 1995.

In the 2000s: can create two files with the same hash, can fake SSL certificate validity – MD5 no longer recommended.
Hash Algorithms

SHA1 (Secure Hash Algorithm) NSA (1995)
Successor to and replacement for MD5
Used in IPSec, SSL, TLS, PGP, SSH, and more (shows up in Java)
Was required by US government crypto applications

Also: SHA2-224, SHA2-256, SHA2-384, SHA2-512
SHA2-224 has digest to match 3DES keys
SHA3-224, SHA3-256, SHA3-384, SHA3-512
Now required by US government crypto applications

Takes digest of up to $2^{64}$ bit messages – digest size is 160 bits

August, 2005: attack announced – find collisions in fewer than $2^{63}$ operations (should have been $2^{80}$ – square root of $2^{160}$)

Operates in stages – each stage mangles the pre-stage message by a sequence of operations based on current message block
<table>
<thead>
<tr>
<th>Algorithm and variant</th>
<th>Output size (bits)</th>
<th>Internal state size (bits)</th>
<th>Block size (bits)</th>
<th>Rounds</th>
<th>Operations</th>
<th>Security (in bits) against collision attacks</th>
<th>Capacity against length extension attacks</th>
<th>Performance on Skylake (median cpb)</th>
<th>First published</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MD5</strong> (as reference)</td>
<td>128</td>
<td>128 (4 × 32)</td>
<td>512</td>
<td>64</td>
<td>And, XOR, ROT, ADD (mod 2(^{32})), OR</td>
<td>≤18 (collisions found[2])</td>
<td>0</td>
<td>4.99</td>
<td>55.00</td>
</tr>
<tr>
<td><strong>SHA-0</strong></td>
<td>160</td>
<td>160 (5 × 32)</td>
<td>512</td>
<td>80</td>
<td>And, XOR, ROT, ADD (mod 2(^{32})), OR</td>
<td>&lt;34 (collisions found)</td>
<td>0</td>
<td>≈ SHA-1</td>
<td>≈ SHA-1</td>
</tr>
<tr>
<td><strong>SHA-1</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;63 (collisions found[3])</td>
<td>0</td>
<td>3.47</td>
<td>52.00</td>
</tr>
<tr>
<td><strong>SHA-2</strong></td>
<td><strong>SHA-224</strong></td>
<td>224</td>
<td>256</td>
<td>512</td>
<td>64 And, XOR, ROT, ADD (mod 2(^{32})), OR, SHR</td>
<td>112 128</td>
<td>32 0</td>
<td>7.62</td>
<td>84.50</td>
</tr>
<tr>
<td></td>
<td><strong>SHA-256</strong></td>
<td>256</td>
<td>256</td>
<td>512</td>
<td>(8 × 32)</td>
<td>64 And, XOR, ROT, ADD (mod 2(^{32})), OR, SHR</td>
<td>128 192 256</td>
<td>384 0</td>
<td>85.25</td>
</tr>
<tr>
<td><strong>SHA-384</strong></td>
<td>384</td>
<td>512 (8 × 64)</td>
<td>1024</td>
<td>80</td>
<td>And, XOR, ROT, ADD (mod 2(^{64})), OR, SHR</td>
<td>112 128</td>
<td>288 256</td>
<td>5.12</td>
<td>5.06</td>
</tr>
<tr>
<td><strong>SHA-512</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>≈ SHA-384</td>
<td>≈ SHA-384</td>
<td>135.75</td>
<td>135.50</td>
</tr>
<tr>
<td><strong>SHA-512/224</strong></td>
<td>224</td>
<td>256</td>
<td>256</td>
<td>512</td>
<td>(8 × 64)</td>
<td>64 And, XOR, ROT, ADD (mod 2(^{64})), OR, SHR</td>
<td>112 128</td>
<td>288 256</td>
<td>≈ SHA-384</td>
</tr>
<tr>
<td><strong>SHA-3</strong></td>
<td><strong>SHA3-224</strong></td>
<td>224</td>
<td>1600</td>
<td>1152</td>
<td>24 And, XOR, ROT, NOT</td>
<td>112 128 192 256</td>
<td>128 152 768 1024</td>
<td>8.12</td>
<td>11.06</td>
</tr>
<tr>
<td></td>
<td><strong>SHA3-256</strong></td>
<td>256</td>
<td>1088</td>
<td>1152</td>
<td>(5 × 5 × 64)</td>
<td>24[4] And, XOR, ROT, NOT</td>
<td>112 128 192 256</td>
<td>128 152 768 1024</td>
<td>8.59</td>
</tr>
<tr>
<td></td>
<td><strong>SHA3-384</strong></td>
<td>384</td>
<td>832</td>
<td>1152</td>
<td>(5 × 5 × 64)</td>
<td>24[4] And, XOR, ROT, NOT</td>
<td>min(d/2, 128)</td>
<td>256 512</td>
<td>8.59</td>
</tr>
<tr>
<td></td>
<td><strong>SHA3-512</strong></td>
<td>512</td>
<td>576</td>
<td>1152</td>
<td>(5 × 5 × 64)</td>
<td>24[4] And, XOR, ROT, NOT</td>
<td>min(d/2, 128)</td>
<td>256 512</td>
<td>8.59</td>
</tr>
<tr>
<td><strong>SHAKE128</strong></td>
<td>d (arbitrary)</td>
<td></td>
<td>1344</td>
<td>1152</td>
<td>24[4] And, XOR, ROT, NOT</td>
<td>min(d/2, 128)</td>
<td>256 512</td>
<td>8.59</td>
<td>15.88</td>
</tr>
<tr>
<td><strong>SHAKE256</strong></td>
<td>d (arbitrary)</td>
<td></td>
<td>1088</td>
<td>1152</td>
<td>24[4] And, XOR, ROT, NOT</td>
<td>min(d/2, 128)</td>
<td>256 512</td>
<td>8.59</td>
<td>15.88</td>
</tr>
</tbody>
</table>
Merkle-Damgård construction for SHA-1 and SHA-2

$f$ is a one-way function that transforms two fixed length inputs to an output of the same size as one of the inputs

IV is an initialization vector
SHA-1 Algorithm

Pre-process the message – pad and end with size:

One bit of '1' following message

Pad with 0's to within 64 bits of end of block

Last 64 bits of last block get the size of the message in bits
SHA-1 Algorithm

Break block into 16 32 bit words:

512 bit block

$W_0$  $W_1$  $W_2$  ...  $W_{15}$

32 bit words
SHA-1 Algorithm

Break block into 16 32 bit words:

\[
W_0 \quad W_1 \quad W_2 \quad \ldots \quad W_{15}
\]

512 bit block

Extend to 80 32 bit words (2560 bits):

\[
\text{for } t \text{ from 16 to 79} \quad W_t = (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) \ll 1
\]
Define variables \( A = 0x67452301, B = 0xEFCDAB89, C = 0x98BADCFE, D = 0x10325476, E = 0xC3D2E1F0 \)

Define \( F \) and \( K_t \) according to round:

- **Rounds 1 to 20:** \( F = (B \land C) \lor (\neg B \land D) \)
  \[ K_t = 0x5A827999 \quad 2^{30} \cdot \text{sqrt}(2) \]

- **Rounds 21 to 40:** \( F = (B \oplus C \oplus D) \)
  \[ K_t = 0x6ED9EBA1 \quad 2^{30} \cdot \text{sqrt}(3) \]

- **Rounds 41 to 60:** \( F = (B \land C) \lor (B \land D) \lor (C \land D) \)
  \[ K_t = 0x8F1BBCDC \quad 2^{30} \cdot \text{sqrt}(5) \]

- **Rounds 61 to 80:** \( F = (B \oplus C \oplus D) \)
  \[ K_t = 0xCA62C1D6 \quad 2^{30} \cdot \text{sqrt}(10) \]
SHA-1 Algorithm

160 bit block (5 32 bit words)

Last round:
A-E is the digest

\[
\text{temp} = (A \lll 5) + F + E + K_t + w_t
\]

\[
E = D
\]

\[
D = C
\]

\[
C = B \lll 30
\]

\[
B = A
\]

\[
A = \text{temp}
\]

Addition mod \(2^{32}\)
SHA-1 Algorithm

Finally, update the 160 bit hash:

\[
H_1' \leftarrow A \\
H_2' \leftarrow B \\
H_3' \leftarrow C \\
H_4' \leftarrow D \\
H_5' \leftarrow E
\]
SHA-256 Algorithm

// Initialize variables
// First 32 bits of fractional part of the square roots of the first 8 primes
h0 = 0x6a09e667
h1 = 0xbb67ae85
h2 = 0x3c6ef372
h3 = 0xa54ff53a
h4 = 0x510e527f
h5 = 0x9b05688c
h6 = 0x1f83d9ab
h7 = 0x5be0cd19
SHA-256 Algorithm

// Initialize table of round constants:  
// First 32 bits of the fractional part of the cube roots of the first 64 primes  
// \( K_1 - K_{64} = \)

0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,
0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,
0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,
0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7,
0xc6e00bf3, 0xe4a00f47, 0xefbe4786, 0x0fc19dc6,
SHA-256 Algorithm

Same pre-processing, same initialization of $W_1$ - $W_{15}$

//Extend the 16 32-bit words into 64 32-bit words:
  for t from 16 to 63
    $s0 = (W_{i-15} \gg 7) \oplus (W_{i-15} \gg 18) \oplus (W_{i-15} \gg 3)$
    $s1 = (W_{i-2} \gg 17) \oplus (W_{i-2} \gg 19) \oplus (W_{i-2} \gg 10)$
    $W_i = W_{i-16} + s0 + W_{i-7} + s1$
SHA-256 Algorithm

Same pre-processing, same initialization of $W_1 - W_{15}$

//Extend the 16 32-bit words into 64 32-bit words:
for t from 16 to 63
  s0 = ($W_{i-15} >> 7$) ⊕ ($W_{i-15} >> 18$) ⊕ ($W_{i-15} >> 3$)
  s1 = ($W_{i-2} >> 17$) ⊕ ($W_{i-2} >> 19$) ⊕ ($W_{i-2} >> 10$)
  $W_t = W_{i-16} + s0 + W_{i-7} + s1$

//Initialize hash value for this chunk:
A = h0
B = h1
C = h2
D = h3
E = h4
F = h5
G = h6
H = h7
SHA-256 Algorithm

Same pre-processing, same initialization of $W_1$ - $W_{15}$

//Main loop:
for t from 0 to 63
  $s0 = (A \ggg_2) \oplus (A \ggg_{13}) \oplus (A \ggg_{22})$
  maj = $(A \land B) \lor (B \land C) \lor (C \land A)$
  $t0 = s0 + maj$
  $s1 = (E \ggg_6) \oplus (E \ggg_{11}) \oplus (E \ggg_{25})$
  ch = $(E \land F) \lor (\neg E \land G)$
  $t1 = H + s1 + ch + K_t + W_t$
  H = G
  G = F
  F = E
  E = D + t1
  D = C
  C = B
  B = A
  A = t0 + t1
SHA-256 Algorithm

//Add this chunk's hash to result so far:
  h0 := h0 + A
  h1 := h1 + B
  h2 := h2 + C
  h3 := h3 + D
  h4 := h4 + E
  h5 := h5 + F
  h6 := h6 + G
  h7 := h7 + H

//Output the final hash value (big-endian):
digest = hash =
  h0 || h1 || h2 || h3 || h4 || h5 || h6 || h7
Types of Attack on Hashes

1. **Preimage** – An attacker has an output and finds an input that hashes to that output

2. **2nd Preimage** – An attacker has an output and an input $x$ and finds a 2nd input that produces the same output as $x$

3. **Collision** – An attacker finds two inputs that hash to the same output

4. **Length Extension** – An attacker, knowing the length of message $M$ and a digest of $M$ signed by a sender can extend $M$ with an additional message $N$ and can compute the digest of $M || N$ even without the key used to sign the digest of $M$
Types of Attack on Hashes

SHA-1 and SHA-2 are susceptible to length extension attacks if the algorithm is misused as follows:

Let S be the secret that is shared by sender and receiver
The sender sends message M to the receiver with
digest Hash(S || M) = D
The receiver should compute Hash(S || M) and check for match with D; M is authentic in case of a match, not otherwise

Attacker appends N to M and is able to continue the digest algorithm to complete the Hash(S || M || N) – attacker replaces M with M || N and Hash(S || N) with Hash(S || M || N)

Original Data: count=10&lat=37.351&user_id=1&long=-119.827&waffle=eggo
New Data: count=10&lat=37.351&user_id=1&long=-119.827&waffle=eggo\x80\x00\x00 ... \x00\x00\x02\x28&waffle=liege

Where 0x228 is length of key plus original message; 0x80 is the ‘1’ at start of pad
Types of Attack on Hashes

SHA1, SHA2:
2\textsuperscript{nd} preimage attacks against long messages are always much more efficient than brute force

Multicollisions (many messages with the same hash) can be found with only a little more work than collisions
Hash Algorithms

SHA-3 Competition (2007-2013)

NIST sponsored competition to replace SHA-2 due to fears that the 2005 success against SHA-1 could be duplicated for SHA-2, especially in private (Merkle-Damgård construction)

Required:
- function had to perform well regardless of implementation
- should withstand known attacks while maintaining a large safety factor. It should emit the same four hash sizes as SHA-2 (224-, 256-, 384-, or 512-bits wide), but be able to supply longer hash sizes if need be
- function had to be subjected to cryptanalysis
- could not use the Merkle-Damgård construction to produce the message hash
Hash Algorithms

SHA-3 Competition (2007-2013)

**Semi-finalists:** Blake (Israel), Grøstl (Austria), Luffa (Hitachi), Blue midnight wish (Norway), Hamsi (Belgium), Shabal (Europe), CubeHash (USA), JH (China?), SHA
dite-3 (Israel, France), ECHO (France), Keccak (Dutch), SIMD (Germany?), Fugue (IBM), Skein (USA – Schneier and others)

**Finalists:** Blake, Grøstl, JH, Keccak, Skein

**Winner:** Keccak (final NIST specification August 2015)

**Did not make the first cut:**
MD6 (MIT) + 10 others

**Broken Entries:**
31
Hash Algorithms

SHA-3

sponge construction:
Data is ‘absorbed’ into the ‘sponge’ then ‘squeezed’ out.

absorbing: message blocks are xored with a subset of state then transformed using a permutation function \( f \)
squeezing: output blocks are read from the same subset of state alternated with an application of \( f \)

security-level: level \( n \) means an attacker will have to perform \( 2^n \) operations to break a system
rate: the size of the part of the state that is written and read
capacity: the size of the part of the state untouched by I/O
capacity determined security: max security level = capacity/2
Hash Algorithms

SHA-3

sponge construction:
  hash function: input is a bit string called message, output called a digest
  extendable output function: function on bit strings where output can be extended to any length.
  state: an array of $b$ bits represented as a 3 dimensional array of size $5 \times 5 \times w$ where $w = b/25$. A bit is accessed by $A[x,y,z]$ sub-arrays are planes and columns
  Keccak-$p$ permutation: comprised of $b$ bits – $n_r$ iterations or rounds are performed before and output is generated denoted Keccak-$p[b, n_r]$
Associated parameters $w = b/25$, $l = \log_2(b/25)$
Hash Algorithms

SHA-3

sponge construction:
conversion from bits $S$ to Array: $A[x,y,z] = S[w(5y+x)+z]$

other conversions:
lan$e(i,j) = \text{column of bits in 2-d position } i,j \text{ (both < 5)}$
plane$(j) = \text{lane}(0,j) \| \text{lane}(1,j) \| \text{lane}(2,j) \| \text{lane}(3,j) \| \text{lane}(4,j)$
$S = \text{plane}(0) \| \text{plane}(1) \| \text{plane}(2) \| \text{plane}(3) \| \text{plane}(4)$
Hash Algorithms

SHA-3

sponge construction:

step mappings:

There are 5 of these - each takes b bits input and produces b bits output
Hash Algorithms

SHA-3

sponge construction - step mappings:

**First mapping:** input is original array \( \mathbf{A} \)

for all pairs \((x, z)\) such that \(0 \leq x < 5\) and \(0 \leq z < w\)

let \( C[x, z] = \mathbf{A}[x, 0, z] \oplus \mathbf{A}[x, 1, z] \oplus \mathbf{A}[x, 2, z] \oplus \mathbf{A}[x, 3, z] \oplus \mathbf{A}[x, 4, z] \)

for all pairs \((x, z)\) such that \(0 \leq x < 5\) and \(0 \leq z < w\)

let \( D[x, z] = C[(x-1) \mod 5, z] \oplus C[(x+1) \mod 5, (z-1) \mod w] \)

for all triples \((x, y, z)\) such that \(0 \leq x, y < 5\), and \(0 \leq z < w\)

let \( \mathbf{A}'[x, y, z] = \mathbf{A}[x, y, z] \oplus D[x, z] \)

XOR each bit in the state with

the parities of two columns in the array
Hash Algorithms

SHA-3

sponge construction - step mappings:

Second mapping: input is array $A$ from first mapping

for all $z$ such that $0 \leq z < w$, let $A'[0, 0, z] = A[0, 0, z]

let $(x, y) = (1, 0)$

for $t$ from 0 to 23:

a. for all $z$ such that $0 \leq z < w$,

let $A'[x, y, z] = A[x, y, (z - (t + 1)(t + 2)/2) \mod w];$

b. let $(x, y) = (y, (2x + 3y) \mod 5)$

Black dots are bits whose coordinates are 0. The shaded cube is the location of the bit after 2nd mapping.
Hash Algorithms

SHA-3

sponge construction - step mappings:

Third mapping: input is array $A$ from second mapping for all triples $(x, y, z)$ such that $0 \leq x, y < 5$, and $0 \leq z < w$

let $A'[x, y, z] = A[(x + 3y) \mod 5, x, z]$

rearrange the positions of the lanes, as shown for slices in the figure
Hash Algorithms

SHA-3

sponge construction - step mappings:

Fourth mapping: input is array $A$ from third mapping

for all triples $(x, y, z)$ such that $0 \leq x, y < 5$, and $0 \leq z < w$,

let $A'[x, y, z] = A[x, y, z] \oplus ((A[(x+1) \mod 5, y, z] \oplus 1) \cdot A[(x+2) \mod 5, y, z])$

(dot means integer multiplication which translates to logic and)

XOR each bit with a non-linear function of two other bits in its row, as illustrated in the figure
Hash Algorithms

SHA-3

sponge construction - step mappings:

Algorithm rc(t):
    if t mod 255 = 0, return 1.
    let R = 10000000.
    for i from 1 to t mod 255, let:
        R = 0 || R;
        R[0] = R[0] ⊕ R[8];
        R = Trunc 8 [R].
    Return R[0]
SHA-3

sponge construction - step mappings:

**Fifth mapping:** input is array $A$ and round index $i_r$

for all triples $(x, y, z)$ such that $0 \leq x,y < 5$, and $0 \leq z < w$,
let $A'[x, y, z] = A[x, y, z]$
let $RC = 0^w$
for $j$ from 0 to 1, let $RC[2^j − 1] = rc(j + 7i_r)$.
for all $z$ such that $0 \leq z < w$
let $A' [0, 0, z] = A' [0, 0, z] \oplus RC[z]$

The effect is to modify some of the bits of lane (0,0) in a manner that depends on the round index $i_r$. The other 24 lanes are not affected by this operation.
Hash Algorithms

SHA-3

sponge construction – perform the permutation:

KECCAK-$p[b,n_r](S)$: $S$ is string of length $b$, $n_r$ is # rounds

convert $S$ into a state array, $A$

for $i_r$ from $12 + 2l - n_r$ to $12 + 2l - 1$

let $A = \text{ResultOfFiveMappingSteps}(A, i_r)$.

convert $A$ into a string $S'$ of length $b$

return $S'$
Hash Algorithms

SHA-3

sponge construction:
The sponge construction is a framework for specifying functions on binary data with arbitrary output length. The construction employs the following three components:

1. An underlying function $f$ on fixed-length strings
2. A parameter $r$ called the rate
3. A padding rule, denoted by $pad$

The function that the construction produces from these components is called a sponge function, denoted by $\text{SPONGE} \ [f, \ pad, \ r]$. A sponge function takes two inputs: a bit string $N$ and the bit length $d$ of the output string, $\text{SPONGE} \ [f, \ pad, \ r](N, \ d)$
SHA-3

sponge construction:

The rate \( r \) is a positive integer that is strictly less than the width \( b \). The capacity \( c \) is such that \( r + c = b \).
Hash Algorithms

SHA-3

sponge construction:

SPONGE[f,pad,r](N,d): output is string Z s.t. len(Z) = d

let $P = N \| pad(r, \text{len}(N))$

let $n = \text{len}(P)/r$

let $c = b-r$

let $P_0, \ldots, P_{n-1}$ be the unique sequence of strings of length $r$

such that $P = P_0 \| \ldots \| P_{n-1}$

let $S = 0^b$

for $i$ from 0 to $n-1$, let $S = f(S \oplus (P_i \| 0^c))$

let $Z$ be the empty string

q: let $Z = Z \| \text{Trunc}_r(S)$.

if $d \leq |Z|$, then return $\text{Trunc}_d(Z)$; else continue.

let $S = f(S)$, and goto q
Hash Algorithms

SHA-3

sponge construction:

\textit{pad}(x,m): \ x \text{ is positive integer, } m \text{ is non-negative integer}

output is string \( P \) s.t. \( m + \text{len}(P) \) is multiple of \( x \)

let \( j = (–m - 2) \mod x \)

return \( P = 1 \parallel 0^j \parallel 1 \)
Hash Algorithms

SHA-3

SHA-3 functions:
- SHA3-224(M) = Keccak[448](M || 01, 224)
- SHA3-256(M) = Keccak[512](M || 01, 256)
- SHA3-384(M) = Keccak[768](M || 01, 384)
- SHA3-512(M) = Keccak[1024](M || 01, 512)

The 01 appended to the message is a code that distinguishes this family of SHA-3 functions from others that may be defined later.
Hash Algorithms

SHA-3: Security
Not susceptible to length extension attack

SHA-2 splits data into elementary blocks and produces, for each block, an output which has exactly the same size as the function output. Moreover, the output for a complete message is the current output after processing all blocks of the message.

The sponge construction has an internal state that is much larger than the hash function output. So hashing an output block – with most of the state missing – is not going to work as the entire state is needed to continue the digest.
## Hash Algorithms

### SHA-3: Security

<table>
<thead>
<tr>
<th>Function</th>
<th>Output Size</th>
<th>Security Strengths in Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Collision</td>
</tr>
<tr>
<td>SHA-1</td>
<td>160</td>
<td>&lt; 80</td>
</tr>
<tr>
<td>SHA-224</td>
<td>224</td>
<td>112</td>
</tr>
<tr>
<td>SHA-512/224</td>
<td>224</td>
<td>112</td>
</tr>
<tr>
<td>SHA-256</td>
<td>256</td>
<td>128</td>
</tr>
<tr>
<td>SHA-512/256</td>
<td>256</td>
<td>128</td>
</tr>
<tr>
<td>SHA-384</td>
<td>384</td>
<td>192</td>
</tr>
<tr>
<td>SHA-512</td>
<td>512</td>
<td>256</td>
</tr>
<tr>
<td>SHA3-224</td>
<td>224</td>
<td>112</td>
</tr>
<tr>
<td>SHA3-256</td>
<td>256</td>
<td>128</td>
</tr>
<tr>
<td>SHA3-384</td>
<td>384</td>
<td>192</td>
</tr>
<tr>
<td>SHA3-512</td>
<td>512</td>
<td>256</td>
</tr>
<tr>
<td>SHAKE128</td>
<td>$d$</td>
<td>$\min(d/2, 128)$</td>
</tr>
<tr>
<td>SHAKE256</td>
<td>$d$</td>
<td>$\min(d/2, 256)$</td>
</tr>
</tbody>
</table>