Cryptography 101: From Theory to Practice

Chapter 6 – Cryptographic Hash Functions

Rolf Oppliger

March 1, 2022

Terms of Use

- This work is published with a CC BY-ND 4.0 license (◎④)
 - CC = Creative Commons (ⓒ)
 - BY = Attribution ()
 - ND = No Derivatives (⑤)

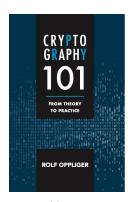
whoami



rolf-oppliger.ch
rolf-oppliger.com

- Swiss National Cyber Security Centre NCSC (scientific employee)
- eSECURITY Technologies Rolf Oppliger (founder and owner)
- University of Zurich (adjunct professor)
- Artech House (author and series editor for information security and privacy)

Reference Book



© Artech House, 2021 ISBN 978-1-63081-846-3

https://books.esecurity.ch/crypto101.html

Challenge Me



Outline

6. Cryptographic Hash Functions

- 1 Introduction
- 2 Cryptographic Systems
- 3 Random Generators
- 4 Random Functions
- 5 One-Way Functions
- Pseudorandom Generators
- 8 Pseudorandom Functions
- 9 Symmetric Encryption
- 10 Message Authentication
- 11 Authenticated Encryption
- 12 Key Establishment
- 13 Asymmetric Encryption
- 14 Digital Signatures
- 15 Zero-Knowledge Proofs of Knowledge
- 16 Key Management
- 17 Summary
- 18 Outlook



- 6.1 Introduction
- 6.2 Merkle-Damgård Construction
- 6.3 Historical Perspective
- 6.4 Exemplary Hash Functions
- 6.5 Final Remarks

- According to Definition 2.5, a **hash function** is an efficiently computable function $h: \Sigma_{in}^* \to \Sigma_{out}^n$ that maps $x \in \Sigma_{in}^*$ to $y \in \Sigma_{out}^n$ (for a fixed output length n)
- Additional properties
 - *H* is **one-way** (**preimage resistant**) if it is computationally infeasible to find $x \in \Sigma_{in}^*$ with h(x) = y for $y \in_R \Sigma_{out}^n$
 - H is second-preimage resistant (weak collision resistant) if it is computationally infeasible to find $x' \in \Sigma_{in}^*$ with $x' \neq x$ and h(x') = h(x) for $x \in_R \Sigma_{in}^*$
 - H is collision-resistant (strong collision resistant) if it is computationally infeasible to find $x, x' \in \Sigma_{in}^*$ with $x' \neq x$ and h(x') = h(x)



- According to Definition 2.6, a hash function h is cryptographic if it is one-way and either second-preimage resistant or collision-resistant
- Remarks (1)
 - Due to the pigeonhole principle, the term "collision free" is wrong and should not be used here
 - If one wants to use complexity-theoretic arguments, then one must consider families of (cryptographic) hash functions
 - Collision resistance implies second-preimage resistance, but not vice versa (this is why the terms "weak collision resistant" and "strong collision resistant" are used in the first place)



6.1 Introduction

Remarks (2)

- Preimage resistance (one-wayness) and collision resistance are inherently different properties
- On the one hand, a preimage resistant function need not be (strong or weak) collision-resistant
 - If g is an n-bit preimage resistant hash function, then the function $h(x) = g(x \mid_n)$ is still preimage resistant but not collision-resistant
 - All $x \parallel y$ (with $|x| \ge n$ and y arbitrary) hash to the same value and yield collisions

- Remarks (3)
 - On the other hand, a (strong or weak) collision-resistant hash function need not be preimage resistant (e.g., Maurer's counterexample)
 - If g is an n-bit collision-resistant hash function, then the (n+1)-bit hash function

$$h(x) = \begin{cases} 1 \mid \mid x & \text{if } \mid x \mid = n \\ 0 \mid \mid g(x) & \text{otherwise} \end{cases}$$

- is still collision-resistant but not preimage resistant
- For all h(x) that begin with 1, it is trivial to find a preimage (just drop the 1)

- The notion of a collision can be generalized to multicollisions
- More specifically, an *r*-collision is an *r*-tuple $(x_1, ..., x_r)$ with $h(x_1) = ... = h(x_r)$
- For r = 2, a 2-collision is a "normal" collision
- Finding multicollisions is not substantially more difficult than finding "normal" collisions

- In practice, Σ_{in} and Σ_{out} are often set to $\{0,1\}$
- A respective hash function is a mapping from $\{0,1\}^*$ to $\{0,1\}^n$
- A practically relevant question is how large n should be
- There is a trade-off here, i.e., n should be as short as possible, but as long as needed
- A lower bound for n is obtained by the birthday attack that exploits the birthday paradox (e.g., $n \ge 256$ to achieve a 128-bit security level)

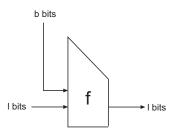
6. Cryptographic Hash Functions

- In general, there are many ways to construct a cryptographic hash function
- According to ISO/IEC 10118-1
 - Hash functions that employ block ciphers (ISO/IEC 10118-2)
 - Dedicated hash functions (ISO/IEC 10118-3)
 - Hash functions based on modular arithmetic (ISO/IEC 10118-4)
- Mainly due to their performance advantages, dedicated hash functions are usually the preferred choice
- Most functions employ the Merkle-Damgård construction and yield iterated hash functions



6.2 Merkle-Damgård Construction

In the late 1980s, Ralph C. Merkle and Ivan B. Damgård independently proposed a construction that can be used to turn a collision-resistant compression function $f: \Sigma^{b+l} \longrightarrow \Sigma^l$ (with $b, l \in \mathbb{N}$) into an iterated hash function h



6.2 Merkle-Damgård Construction

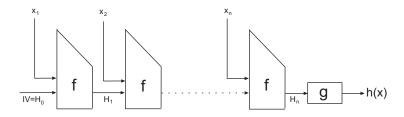
- There are many possibilities to design and come up with a compression function f
- A popular (i.e., widely used) possibility is a Davies-Meyer compression function
- It applies a block cipher E on a chaining value H_i , where the respective message block x_i serves as the key:

$$H_i = E_{x_i}(H_{i-1}) \oplus H_{i-1}$$
 for $i = 1, ..., n$



6.2 Merkle-Damgård Construction

- In a typical setting, I is 160 or 256 bits and b is 512 bits
- An iterated hash function looks as follows:



6.2 Merkle-Damgård Construction

Such a function h can be defined as follows:

$$H_0 = IV$$

 $H_i = f(H_{i-1}, x_i)$ for $i = 1, ..., n$
 $h(x) = g(H_n)$

- The message x must be padded to a multiple of b bits
- The padding method of choice is to append (at the end of the message) a 1, a variable number of 0s, and the binary encoding of the message length

6.2 Merkle-Damgård Construction

Merkle and Damgård showed that finding a collision for h is at least as hard as finding a collision for f

Theorem (Merkle-Damgård)

If the compression function f is collision-resistant, then the iterated hash function h that is built according to the Merkle-Damgård construction is also collision-resistant

■ There are only a few cryptographic hash functions that don't employ the Merkle-Damgård construction (e.g., Keccak)

- The first cryptographic hash function was developed in the 1980s by RSA Security (acronymed MD for "message digest")
- It was proprietary and never published
- MD2 (RFC 1319) was the first cryptographic hash function that was published and used in the field
- After the announcemnet of SNEFRU by Ralph C. Merkle, RSA Security released MD4 (RFC 1320)
- In 1991, SNEFRU and some other hash functions were broken and weaknesses were found in MD4
- As a result, RSA Security came up with MD5 (RFC 1321)
 and released RFCs 1319 1321 in April 1992

- In 1993, the U.S. NIST proposed the Secure Hash Algorithm (SHA), which is similar to MD5, but more strengthened and a little bit slower
- Probably after discovering a never-published weakness in the original SHA proposal, the U.S. NIST released SHA-1
- In 1995, SHA-1 was specified in FIPS PUB 180 (later in RFC 4634) and has been revised multiple times since then
- The latest revision is FIPS PUB 180-4 (August 2015)
- It also introduces the SHA-2 family

Message Size	Block Size	Word Size	Hash Value Size
$< 2^{64}$ bits	512 bits	32 bits	160 bits
$< 2^{64}$ bits	512 bits	32 bits	224 bits
$< 2^{64}$ bits	512 bits	32 bits	256 bits
$< 2^{128} \text{ bits}$	1,024 bits	64 bits	384 bits
$< 2^{128}$ bits	1,024 bits	64 bits	512 bits
$< 2^{128} \text{ bits}$	1,024 bits	64 bits	224 bits
$< 2^{128}$ bits	1,024 bits	64 bits	256 bits
	$< 2^{64}$ bits $< 2^{64}$ bits $< 2^{64}$ bits $< 2^{128}$ bits $< 2^{128}$ bits $< 2^{128}$ bits	$<2^{64}$ bits 512 bits $<2^{64}$ bits 512 bits $<2^{64}$ bits 512 bits $<2^{128}$ bits 512 bits $<2^{128}$ bits 1,024 bits $<2^{128}$ bits 1,024 bits $<2^{128}$ bits 1,024 bits $<2^{128}$ bits 1,024 bits	$<2^{64}$ bits 512 bits 32 bits $<2^{128}$ bits 1,024 bits 64 bits

- During the 1990s, a series of results showed that MD4 was insecure and MD5 was partially broken
- In the early 2000s, a group of Chinese researchers (headed by Xiaoyun Wang) published collisions for MD4, MD5, and a few other cryptographic hash functions
- In 2008, a Dutch research group (headed by Arjen Lenstra) found a way to exploit an MD5 collision to create a rogue CA certificate
- Consequently, MD4, MD5, and a few other cryptographic hash functions should no longer be used (they may still serve as study objects)

- In 2005, Wang et al. also presented collisions for SHA-1
- The original attack required 2⁶⁹ (instead of 2⁸⁰) hash operations to find a collision, but it can be improved to 2⁶³
- The attack was widely discussed in the media and led to a better adoption of SHA-2
- Also, a NIST competition for SHA-3 was initiated
- In 2012, Keccak was announced as the winner of the competition
- RIPEMD and RIPEMD-160 are European versions of MD5 and SHA-1 (not further addressed)

6.4 Exemplary Hash Functions — MD4

- MD4 follows the Merkle-Damgård construction and uses a Davies-Meyer compression function with b = 512 and l = 128
- The output length is 128 bits
- The function was designed to be efficiently executed on 32-bit processors with a little-endian architecture
- This means that a 4-byte word $a_1a_2a_3a_4$ is stored as $a_4a_3a_2a_1$, representing the integer $a_42^{24} + a_32^{16} + a_22^8 + a_1$

6.4 Exemplary Hash Functions — MD4

- Let $m = m_0 m_1 \dots m_{s-1}$ be an s-bit message
- MD4 first generates an array w of n 32-bit words

$$w = w[0] || [1] || ... || w[n-1]$$

where *n* is a multiple of 16, i.e., $n \equiv 0 \pmod{16}$

■ Hence, the bitlength of w is a multiple of $32 \cdot 16 = 512$ bits

$$w[0] = m_0 m_1 \dots m_{31}$$

 $w[1] = m_{32} m_{33} \dots m_{63}$
 \dots
 $w[n-1] = m_{s-32} m_{s-31} \dots m_{s-1}$

6.4 Exemplary Hash Functions — MD4

- More specifically, w is constructed in two steps:
 - First, *m* is padded (with a 1 and variable number of 0s) so that the bitlength is congruent to 448 modulo 512 (i.e., 64 bits short of being a multiple of 512 bits)
 - Second, a 64-bit binary representation of s is appended (to form the last two words of w)



6.4 Exemplary Hash Functions — MD4

 \blacksquare F, g, and h are logical functions

$$f(X, Y, Z) = (X \land Y) \lor ((\neg X) \land Z)$$

$$g(X, Y, Z) = (X \land Y) \lor (X \land Z) \lor (Y \land Z)$$

$$h(X, Y, Z) = X \oplus Y \oplus Z$$

- lacksquare C_1 and C_2 are constants
- $W \stackrel{\frown}{\hookleftarrow} c$ refers to the c-bit left rotation (circular left shift) of word w (0 < c < 31)

Х	Υ	Z	f	g	h
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	0	1	0
1	1	0	1	1	0
1	1	1	1	1	1

6.4 Exemplary Hash Functions — MD4

Overview:

```
(m)
Construct w = w[0] \| w[1] \| \dots \| w[n-1]
A = 0x67452301
B = 0xEFCDAB89
C = 0x98BADCFE
D = 0 \times 10325476
for i = 0 to n/16 - 1 do
     for i = 0 to 15 do X[i] = w[i \cdot 16 + i]
     A' = A
     B' = B
     C' = C
     D' = D
     Round 1
     Round 2
     Round 3
     A = A + A'
     B = B + B'
     C = C + C'
     D = D + D'
```

Round 1:

```
1. A = (A + f(B, C, D) + X[0]) \stackrel{\frown}{\leftarrow} 3
2. D = (D + f(A, B, C) + X[1]) \stackrel{\frown}{\leftarrow} 7
3. C = (C + f(D, A, B) + X[2]) \stackrel{\frown}{\leftarrow} 11
4. B = (B + f(C, D, A) + X[3]) \stackrel{\frown}{\longleftrightarrow} 19
5. A = (A + f(B, C, D) + X[4]) \stackrel{\sim}{\leftarrow} 3
6. D = (D + f(A, B, C) + X[5]) \stackrel{\frown}{\leftarrow} 7
7. C = (C + f(D, A, B) + X[6]) \stackrel{\frown}{\leftarrow} 11
8. B = (B + f(C, D, A) + X[7]) \stackrel{\frown}{\leftarrow} 19
9. A = (A + f(B, C, D) + X[8]) \stackrel{\frown}{\leftarrow} 3
10. D = (D + f(A, B, C) + X[9]) \stackrel{\frown}{\leftarrow} 7
11. C = (C + f(D, A, B) + X[10]) \stackrel{\frown}{\leftarrow} 11
12. B = (B + f(C, D, A) + X[11]) \stackrel{\frown}{\leftarrow} 19
13. A = (A + f(B, C, D) + X[12]) \stackrel{\frown}{\leftarrow} 3
14. D = (D + f(A, B, C) + X[13]) \stackrel{\frown}{\leftarrow} 7
15. C = (C + f(D, A, B) + X[14]) \stackrel{\frown}{\leftarrow} 11
16. B = (B + f(C, D, A) + X[15]) \stackrel{\frown}{\leftarrow} 19
```

 $(h(m) = A \parallel B \parallel C \parallel D)$

6.4 Exemplary Hash Functions — MD4

Round 2:

1.
$$A = (A + g(B, C, D) + X[0] + c_1) \stackrel{\sim}{\sim} 3$$

2. $D = (D + g(A, B, C) + X[4] + c_1) \stackrel{\sim}{\sim} 5$
3. $C = (C + g(D, A, B) + X[8] + c_1) \stackrel{\sim}{\sim} 9$
4. $B = (B + g(C, D, A) + X[12] + c_1) \stackrel{\sim}{\sim} 13$
5. $A = (A + g(B, C, D) + X[1] + c_1) \stackrel{\sim}{\sim} 3$
6. $D = (D + g(A, B, C) + X[5] + c_1) \stackrel{\sim}{\sim} 5$
7. $C = (C + g(D, A, B) + X[9] + c_1) \stackrel{\sim}{\sim} 9$
8. $B = (B + g(C, D, A) + X[13] + c_1) \stackrel{\sim}{\sim} 13$
9. $A = (A + g(B, C, D) + X[2] + c_1) \stackrel{\sim}{\sim} 3$
10. $D = (D + g(A, B, C) + X[6] + c_1) \stackrel{\sim}{\sim} 5$
11. $C = (C + g(D, A, B) + X[10] + c_1) \stackrel{\sim}{\sim} 9$
12. $B = (B + g(C, D, A) + X[14] + c_1) \stackrel{\sim}{\sim} 13$
13. $A = (A + g(B, C, D) + X[3] + c_1) \stackrel{\sim}{\sim} 5$
15. $C = (C + g(D, A, B) + X[11] + c_1) \stackrel{\sim}{\sim} 5$
16. $C = (C + g(D, A, B) + X[11] + c_1) \stackrel{\sim}{\sim} 5$

Round 3:

1.
$$A = (A + h(B, C, D) + X[0] + c_2) \stackrel{\checkmark}{\longleftrightarrow} 3$$

2. $D = (D + h(A, B, C) + X[8] + c_2) \stackrel{\checkmark}{\longleftrightarrow} 9$
3. $C = (C + h(D, A, B) + X[4] + c_2) \stackrel{\checkmark}{\longleftrightarrow} 11$
4. $B = (B + h(C, D, A) + X[12] + c_2) \stackrel{\checkmark}{\longleftrightarrow} 15$
5. $A = (A + h(B, C, D) + X[2] + c_2) \stackrel{\checkmark}{\longleftrightarrow} 3$
6. $D = (D + h(A, B, C) + X[10] + c_2) \stackrel{\checkmark}{\longleftrightarrow} 9$
7. $C = (C + h(D, A, B) + X[6] + c_2) \stackrel{\checkmark}{\longleftrightarrow} 11$
8. $B = (B + h(C, D, A) + X[14] + c_2) \stackrel{\checkmark}{\longleftrightarrow} 15$
9. $A = (A + h(B, C, D) + X[1] + c_2) \stackrel{\checkmark}{\longleftrightarrow} 3$
10. $D = (D + h(A, B, C) + X[9] + c_2) \stackrel{\checkmark}{\longleftrightarrow} 9$
11. $C = (C + h(D, A, B) + X[5] + c_2) \stackrel{\checkmark}{\longleftrightarrow} 11$
12. $B = (B + h(C, D, A) + X[13] + c_2) \stackrel{\checkmark}{\longleftrightarrow} 1$

5.
$$A = (A + h(B, C, D) + X[2] + c_2) \stackrel{\checkmark}{\longleftrightarrow} 3$$

6. $D = (D + h(A, B, C) + X[10] + c_2) \stackrel{\checkmark}{\longleftrightarrow} 9$
7. $C = (C + h(D, A, B) + X[6] + c_2) \stackrel{\checkmark}{\longleftrightarrow} 11$
8. $B = (B + h(C, D, A) + X[14] + c_2) \stackrel{\checkmark}{\longleftrightarrow} 15$

9.
$$A = (A + h(B, C, D) + X[1] + c_2) \stackrel{\checkmark}{\longleftrightarrow} 3$$

10. $D = (D + h(A, B, C) + X[9] + c_2) \stackrel{\checkmark}{\longleftrightarrow} 9$
11. $C = (C + h(D, A, B) + X[5] + c_2) \stackrel{\checkmark}{\longleftrightarrow} 11$

12.
$$B = (B + h(C, D, A) + X[13] + c_2) \stackrel{\frown}{\leftarrow} 15$$

13. $A = (A + h(B, C, D) + X[3] + c_2) \stackrel{\frown}{\leftarrow} 3$

14.
$$D = (D + h(A, B, C) + X[11] + c_2) \stackrel{\frown}{\longleftrightarrow} 9$$

15. $C = (C + h(D, A, B) + X[7] + c_2) \stackrel{\frown}{\longleftrightarrow} 11$

16.
$$B = (B + h(C, D, A) + X[15] + c_2) \longleftrightarrow 15$$

16.
$$B = (B + h(C, D, A) + X[15] + c_2) \stackrel{\leftarrow}{\longleftrightarrow} 15$$

6.4 Exemplary Hash Functions — MD5

- MD5 is a strengthened version of MD4
- It is conceptually and structurally similar to MD4
- The main difference is that MD5 invokes 4 rounds (instead of only 3)
- This is advantageous from a security viewpoint, but it is disadvantageous from a performance viewpoint (i.e., performance decreases one third)
- MD5 uses a slightly modified function f, an additional function i, and a word table T with 64 entries that is constructed from the sine function (instead of c_1 and c_2)

6.4 Exemplary Hash Functions — MD5

Overview:

```
(m)
```

```
Construct w = w[0] \| w[1] \| \dots \| w[n-1]
A = 0x67452301
B = 0xEFCDAB89
C = 0x98BADCFE
D = 0 \times 10325476
for i = 0 to n/16 - 1 do
     for i = 0 to 15 do X[i] = w[i \cdot 16 + i]
     A' = A
     R' = R
     C' = C
     D' = D
     Round 1
     Round 2
     Round 3
     Round 4
     A = A + A'
     B = B + B'
     C = C + C'
     D = D + D'
```

Round 1:

```
1. A = (A + f(B, C, D) + X[0] + T[1]) \stackrel{\frown}{\leftarrow} 7
2. D = (D + f(A, B, C) + X[1] + T[2]) \stackrel{\frown}{\leftarrow} 12
3. C = (C + f(D, A, B) + X[2] + T[3]) \stackrel{\frown}{\leftarrow} 17
4. B = (B + f(C, D, A) + X[3] + T[4]) \stackrel{\frown}{\leftarrow} 22
5. A = (A + f(B, C, D) + X[4] + T[5]) \stackrel{\frown}{\leftarrow} 7
6. D = (D + f(A, B, C) + X[5] + T[6]) \stackrel{\frown}{\leftarrow} 12
7. C = (C + f(D, A, B) + X[6] + T[7]) \stackrel{\triangle}{\leftarrow} 17
8. B = (B + f(C, D, A) + X[7] + T[8]) \stackrel{\frown}{\leftarrow} 22
9. A = (A + f(B, C, D) + X[8] + T[9]) \stackrel{\sim}{\leftarrow} 7
10. D = (D + f(A, B, C) + X[9] + T[10]) \stackrel{\frown}{\leftarrow} 12
11. C = (C + f(D, A, B) + X[10] + T[11]) \stackrel{\frown}{\leftarrow} 17
12. B = (B + f(C, D, A) + X[11] + T[12]) \stackrel{\frown}{\leftarrow} 22
13. A = (A + f(B, C, D) + X[12] + T[13]) \stackrel{\sim}{\leftarrow} 7
14. D = (D + f(A, B, C) + X[13] + T[14]) \stackrel{\frown}{\leftarrow} 12
15. C = (C + f(D, A, B) + X[14] + T[15]) \stackrel{\frown}{\leftarrow} 17
16. B = (B + f(C, D, A) + X[15] + T[16]) \stackrel{\frown}{\leftarrow} 22
```

 $(h(m) = A \parallel B \parallel C \parallel D)$

6.4 Exemplary Hash Functions — MD5

Round 2:

```
1. A = (A + g(B, C, D) + X[1] + T[17]) \stackrel{\sim}{\leftarrow} 5
2. D = (D + g(A, B, C) + X[6] + T[18]) \stackrel{\sim}{\leftarrow} 9
3. C = (C + g(D, A, B) + X[11] + T[19]) \stackrel{\frown}{\leftarrow} 14
4. B = (B + g(C, D, A) + X[0] + T[20]) \stackrel{\frown}{\leftarrow} 20
5. A = (A + g(B, C, D) + X[5] + T[21]) \stackrel{\frown}{\leftarrow} 5
6. D = (D + g(A, B, C) + X[10] + T[22]) \stackrel{\frown}{\leftarrow} 9
7. C = (C + g(D, A, B) + X[15] + T[23]) \stackrel{\frown}{\longleftrightarrow} 14
8. B = (B + g(C, D, A) + X[4] + T[24]) \stackrel{\frown}{\leftarrow} 20
9. A = (A + g(B, C, D) + X[9] + T[25]) \stackrel{\sim}{\leftarrow} 5
10. D = (D + g(A, B, C) + X[14] + T[26]) \stackrel{\sim}{\leftarrow} 9
11. C = (C + g(D, A, B) + X[3] + T[27]) \stackrel{\frown}{\leftrightarrow} 14
12. B = (B + g(C, D, A) + X[8] + T[28]) \stackrel{\frown}{\leftarrow} 20
13. A = (A + g(B, C, D) + X[13] + T[29]) \stackrel{\frown}{\longleftrightarrow} 5
14. D = (D + g(A, B, C) + X[2] + T[30]) \stackrel{\frown}{\leftarrow} 9
15. C = (C + g(D, A, B) + X[7] + T[31]) \stackrel{\frown}{\leftarrow} 14
16. B = (B + g(C, D, A) + X[12] + T[32]) \stackrel{\frown}{\leftarrow} 20
```

Round 3:

```
1. A = (A + h(B, C, D) + X[5] + T[33]) \stackrel{\frown}{\leftarrow} 4
2. D = (D + h(A, B, C) + X[8] + T[34]) \stackrel{\frown}{\longleftrightarrow} 11
3. C = (C + h(D, A, B) + X[11] + T[35]) \stackrel{\frown}{\leftarrow} 16
4. B = (B + h(C, D, A) + X[14] + T[36]) \stackrel{\frown}{\leftarrow} 23
5. A = (A + h(B, C, D) + X[1] + T[37]) \stackrel{\sim}{\leftarrow} 4
6. D = (D + h(A, B, C) + X[4] + T[38]) \stackrel{\frown}{\leftarrow} 11
7. C = (C + h(D, A, B) + X[7] + T[39]) \stackrel{\frown}{\leftarrow} 16
8. B = (B + h(C, D, A) + X[10] + T[40]) \stackrel{\frown}{\leftarrow} 23
9. A = (A + h(B, C, D) + X[13] + T[41]) \stackrel{\sim}{\leftarrow} 4
10. D = (D + h(A, B, C) + X[0] + T[42]) \stackrel{\frown}{\leftarrow} 11
11. C = (C + h(D, A, B) + X[3] + T[43]) \stackrel{\frown}{\leftarrow} 16
12. B = (B + h(C, D, A) + X[6] + T[44]) \stackrel{\frown}{\leftarrow} 23
13. A = (A + h(B, C, D) + X[9] + T[45]) \stackrel{\frown}{\leftarrow} 4
14. D = (D + h(A, B, C) + X[12] + T[46]) \stackrel{\frown}{\leftarrow} 11
15. C = (C + h(D, A, B) + X[15] + T[47]) \stackrel{\frown}{\leftarrow} 16
16. B = (B + h(C, D, A) + X[2] + T[48]) \stackrel{\frown}{\leftarrow} 23
```

6.4 Exemplary Hash Functions — MD5

Round 4:

```
1. A = (A + i(B, C, D) + X[0] + T[49]) \stackrel{\frown}{\leftarrow} 6
2. D = (D + i(A, B, C) + X[7] + T[50]) \stackrel{\frown}{\leftarrow} 10
3. C = (C + i(D, A, B) + X[14] + T[51]) \stackrel{\frown}{\leftarrow} 15
4. B = (B + i(C, D, A) + X[5] + T[52]) \stackrel{\frown}{\leftarrow} 21
5. A = (A + i(B, C, D) + X[12] + T[53]) \stackrel{\sim}{\leftarrow} 6
6. D = (D + i(A, B, C) + X[3] + T[54]) \stackrel{\frown}{\leftarrow} 10
7. C = (C + i(D, A, B) + X[10] + T[55]) \stackrel{\frown}{\leftarrow} 15
8. B = (B + i(C, D, A) + X[1] + T[56]) \stackrel{\frown}{\longleftrightarrow} 21
9. A = (A + i(B, C, D) + X[8] + T[57]) \stackrel{\sim}{\leftarrow} 6
10. D = (D + i(A, B, C) + X[15] + T[58]) \stackrel{\frown}{\leftarrow} 10
11. C = (C + i(D, A, B) + X[6] + T[59]) \stackrel{\frown}{\leftarrow} 15
12. B = (B + i(C, D, A) + X[13] + T[60]) \stackrel{\frown}{\leftarrow} 21
13. A = (A + i(B, C, D) + X[4] + T[61]) \stackrel{\frown}{\leftarrow} 6
14. D = (D + i(A, B, C) + X[11] + T[62]) \stackrel{\frown}{\leftarrow} 10
15. C = (C + i(D, A, B) + X[2] + T[63]) \stackrel{\frown}{\leftarrow} 15
16. B = (B + i(C, D, A) + X[9] + T[64]) \stackrel{\frown}{\leftarrow} 21
```

- MD5 is susceptible to collision attacks
- While a "normal" attack requires 2⁶⁴ hash computations, the collision attack of Wang et al. requires 2³⁹ and the best-known attack 2³²
- This value is so small that MD5 must not be used anymore

6.4 Exemplary Hash Functions — SHA-1

- **SHA-1** was specified by the U.S. NIST in FIPS PUB 180 (currently FIPS PUB 180-4)
- Again, it is conceptually and structurally similar to MD5
- Major differences
 - SHA-1 is optimized for computer systems with a big-endian architecture (instead of a little-endian architecture)
 - SHA-1 employs 5 registers A, B, C, D, and E (instead of 4)
 - SHA-1 yields 160-bit hash values (instead of 128-bit hash values)



6.4 Exemplary Hash Functions — SHA-1

■ Instead of f, g, h, and i, SHA-1 uses a sequence of 80 logical functions f₀, f₁, . . . , f₇₉:

$$f_t(X,Y,Z) = \left\{ \begin{array}{ll} \textit{Ch}(X,Y,Z) = (X \wedge Y) \oplus ((\neg X) \wedge Z) & 0 \leq t \leq 19 \\ \textit{Parity}(X,Y,Z) = X \oplus Y \oplus Z & 20 \leq t \leq 39 \\ \textit{Maj}(X,Y,Z) = (X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z) & 40 \leq t \leq 59 \\ \textit{Parity}(X,Y,Z) = X \oplus Y \oplus Z & 60 \leq t \leq 79 \end{array} \right.$$

Note that the *Parity* function occurs twice $(20 \le t \le 39)$ and $60 \le t \le 79$, and that *Ch* and *Maj* are similar to *f* and *g* (\lor is replaced with \oplus , but this doesn't change the result)

6.4 Exemplary Hash Functions — SHA-1

Instead of c_1 and c_2 (MD4) or the 64 words of table T (MD5), SHA-1 uses 4 constant 32-bit words that are used to build a sequence of 80 words K_0 , K_1 , ..., K_{79} :

$$K_t = \left\{ \begin{array}{ll} \lfloor 2^{30}\sqrt{2} \rfloor = 0 \text{x5A827999} & 0 \leq t \leq 19 \\ \lfloor 2^{30}\sqrt{3} \rfloor = 0 \text{x6ED9EBA1} & 20 \leq t \leq 39 \\ \lfloor 2^{30}\sqrt{5} \rfloor = 0 \text{x8F1BBCDC} & 40 \leq t \leq 59 \\ \lfloor 2^{30}\sqrt{10} \rfloor = 0 \text{xCA62C1D6} & 60 \leq t \leq 79 \end{array} \right.$$

6.4 Exemplary Hash Functions — SHA-1

- While w is an array of 32-bit words in MD4 and MD5, SHA-1 uses an array b of 16-word blocks instead
- Hence, b[i] (i = 0, 1, ..., n-1) refers to a 16-word block that is $16 \cdot 32 = 512$ bits long
- SHA-1 uses each 16-word block *b* to recursively derive an 80-word message schedule *W*:

$$W_{t} = \begin{cases} b_{t} & 0 \leq t \leq 15 \\ (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) & 1 & 16 \leq t \leq 79 \end{cases}$$

■ The 16 words of b become the first 16 words of W, and the remaining 80-16=64 words of W are generated according to the formula

6.4 Exemplary Hash Functions — SHA-1

Overview:

```
(m)  \begin{aligned} &\text{Construct } b = b[0] \parallel b[1] \parallel \ldots \parallel b[n-1] \\ &A = 0x67452301 \\ &B = 0xEFCDAB89 \\ &C = 0x98BADCFE \\ &D = 0x10325476 \\ &E = 0xC3D2E1F0 \\ &for \ i = 0 \ to \ n-1 \ do \\ &\text{Derive message schedule } W \ \text{from } b[i] \\ &A' = A \\ &B' = B \\ &C' = C \\ &D' = D \\ &E' = E \\ &| \end{aligned}
```

```
for t = 0 to 79 do

T = (A \stackrel{\longleftarrow}{\leftarrow} 5) + f_t(B, C, D) + E + K_t + W_t

E = D

D = C

C = B \stackrel{\longleftarrow}{\leftarrow} 30

B = A

A = T

A = A + A'

B = B + B'

C = C + C'

D = D + D'

E = E + E'
```

(h(m) = A || B || C || D || E)

- SHA-1 was first broken in 2005 (2⁶⁹ instead of 2⁸⁰ hash computations)
- The attack was later improved (2⁶³ hash computations)
- In 2011, the U.S. NIST deprecated SHA-1, and disallowed its use for digital signatures by the end of 2013
- Two recent attacks have brought SHA-1 to the end of its life cycle
 - SHAttered (2017)
 - SHA-1 is a Shambles (2019)

6.4 Exemplary Hash Functions — SHA-2 family

- The functions of the **SHA-2 family** are listed in Table 6.1
- The functions employ the *Ch* and *Maj* functions from SHA-1 (applied to 32-bit or 64-bit words)
- In the case of SHA-224 and SHA-256, these functions are complemented by 4 32-bit functions:

$$\Sigma_0^{\{256\}}(X) = (X \stackrel{\curvearrowleft}{\hookrightarrow} 2) \oplus (X \stackrel{\backsim}{\hookrightarrow} 13) \oplus (X \stackrel{\backsim}{\hookrightarrow} 22)
\Sigma_1^{\{256\}}(X) = (X \stackrel{\backsim}{\hookrightarrow} 6) \oplus (X \stackrel{\backsim}{\hookrightarrow} 11) \oplus (X \stackrel{\backsim}{\hookrightarrow} 25)
\sigma_0^{\{256\}}(X) = (X \stackrel{\backsim}{\hookrightarrow} 7) \oplus (X \stackrel{\backsim}{\hookrightarrow} 18) \oplus (X \hookrightarrow 3)
\sigma_1^{\{256\}}(X) = (X \stackrel{\backsim}{\hookrightarrow} 17) \oplus (X \stackrel{\backsim}{\hookrightarrow} 19) \oplus (X \hookrightarrow 10)$$

■ Note that \hookrightarrow refers to the *c*-bit right shift operator

6.4 Exemplary Hash Functions — SHA-2 family

All other hash functions from the SHA-2 family use similar 64-bit functions:

$$\Sigma_0^{\{512\}}(X) = (X \stackrel{\curvearrowleft}{\hookrightarrow} 28) \oplus (X \stackrel{\backsim}{\hookrightarrow} 34) \oplus (X \stackrel{\backsim}{\hookrightarrow} 39)
\Sigma_1^{\{512\}}(X) = (X \stackrel{\backsim}{\hookrightarrow} 14) \oplus (X \stackrel{\backsim}{\hookrightarrow} 18) \oplus (X \stackrel{\backsim}{\hookrightarrow} 41)
\sigma_0^{\{512\}}(X) = (X \stackrel{\backsim}{\hookrightarrow} 1) \oplus (X \stackrel{\backsim}{\hookrightarrow} 8) \oplus (X \hookrightarrow 7)
\sigma_1^{\{512\}}(X) = (X \stackrel{\backsim}{\hookrightarrow} 19) \oplus (X \stackrel{\backsim}{\hookrightarrow} 61) \oplus (X \hookrightarrow 6)$$

■ All Σ -functions are used in the round functions, whereas all σ -functions are used to derive the message schedule W

6.4 Exemplary Hash Functions — SHA-2 family

■ While SHA-1 uses four 32-bit words to represent the constants K_0, K_1, \ldots, K_{79} , SHA-224 and SHA-256 use a sequence of 64 distinct 32-bit words that serve as constants

$$\textit{K}_{0}^{\{256\}},\textit{K}_{1}^{\{256\}},\ldots,\textit{K}_{63}^{\{256\}}$$

■ The 64 words are generated by taking the first 32 bits of the fractional parts of the cube roots of the first 64 prime numbers (not addressed here)

6.4 Exemplary Hash Functions — SHA-2 family

 Similarly, SHA-384, SHA-512, SHA-512/224, and SHA-512/256 use a sequence of 80 distinct 64-bit words that serve as constants

$$K_0^{\{512\}}, K_1^{\{512\}}, \dots, K_{79}^{\{512\}}$$

The 80 words represent the first 64 bits of the fractional parts of the cube roots of the first 80 prime numbers (so the first 32 bits of the first 64 values are the same as with SHA-224 and SHA-256)

6.4 Exemplary Hash Functions — SHA-2 family

- While SHA-224 and SHA-256 require messages to be padded to a multiple of 512 bits, all other SHA-2 hash functions require messages to be padded to a multiple of 1024 bits
- In this case, the length of the original message is encoded in the final two 64-bit words (instead of two 32-bit words)
- All functions from the SHA-2 family operate on 8 32- or 64-bit registers A, B, C, D, E, F, G, and H
- The registers are initialized in a particular way (not addressed here)

6.4 Exemplary Hash Functions — SHA-2 family

- SHA-256 uses message schedule W (with 64 words) and two temporary variables T_1 and T_2 (\rightarrow animation)
- SHA-224 uses different initialization values and truncates the output to 224 bits

```
(m)
Construct b = b[0] \| b[1] \| \dots \| b[n-1]
 A = 0x6A09E667 B = 0xBB67AE85
 C = 0x3C6EF372 D = 0xA54FF53A
   F = 0x510E527F F = 0x9B05688C
 G = 0 \times 1 = 
for i = 0 to n - 1 do
                                  Derive message schedule W from b[i]
                                  A' = A B' = B C' = C D' = D
                                  E' = E F' = F G' = G H' = H
                                  for t = 0 to 63 do
                                                                     T_1 = H + \Sigma_1^{\{256\}}(E) + Ch(E, F, G) + K_t^{\{256\}} + W_t
                                                                     T_2 = \Sigma_0^{\{256\}}(A) + Maj(A, B, C)
                                                                    E = D + T_1 D = C
                                                                    C = B B = A
                                                                    A = T_1 + T_2
                                  A = A + A' B = B + B' C = C + C'
                                  D = D + D' F = F + F' F = F + F'
                                   G = G + G' H = H + H'
```

 $(h(m) = A \parallel B \parallel C \parallel D \parallel E \parallel F \parallel G \parallel H)$

6.4 Exemplary Hash Functions — SHA-2 family

- SHA-512 uses 64-bit words, the same message schedule W (with 80 words), and T₁ and T₂
- SHA-384, SHA-512/ 224, and SHA-512/ 256 use different initialization values and truncate the output

```
(m)
Construct b = b[0] \| b[1] \| \dots \| b[n-1]
C = 0x3C6EF372FE94F82B D = 0xA54FF53A5F1D36F1
F = 0 \times 510E527FADE682D1 F = 0 \times 9B05688C2B3E6C1F
G = 0x1F83D9ABFB41BD6B
                       H = 0 \times 5 RE C C D 1 9 1 3 7 E 2 1 7 9
for i = 0 to n - 1 do
    Derive message schedule W from b[i]
    A' = A B' = B C' = C D' = D
    E' = E F' = F G' = G H' = H
    for t = 0 to 79 do
         T_1 = H + \Sigma_1^{\{512\}}(E) + Ch(E, F, G) + K_t^{\{512\}} + W_t
         T_2 = \Sigma_0^{\{512\}}(A) + Maj(A, B, C)
         E = D + T_1 D = C
         C = B B = A
         A = T_1 + T_2
    A = A + A' B = B + B' C = C + C' D = D + D'
    F = F + F' F = F + F' G = G + G' H = H + H'
```

 $(h(m) = A \parallel B \parallel C \parallel D \parallel E \parallel F \parallel G \parallel H)$

6.4 Exemplary Hash Functions — SHA-2 family

- As of this writing, the cryptographic hash functions from the SHA-2 family are considered to be secure
- They are used in many applications, such as Bitcoin (double SHA-2) and many other cryptocurrencies
- There is no need to replace them in the short term
- If one is worried about quantum computers, then SHA-384 and SHA-512 can be used

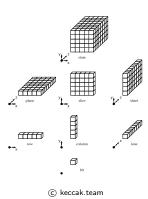
- Keccak is the algorithm selected by the U.S. NIST as the winner of the public SHA-3 competition in 2012
- FIPS PUB 202 complements FIPS PUB 180-4
- It specifies 4 cryptographic hash functions and 2 extendable-output functions (XOFs)
 - SHA3-224, SHA3-256, SHA3-384, and SHA3-512
 - SHAKE128 and SHAKE256 (where SHAKE stands for "Secure Hash Algorithm with Keccak")
- Keccak/SHA-3 relies on the **sponge construction** (instead of the Merkle-Damgård construction)



- In December 2016, NIST released SP 800-18540 that specifies complementary functions derived from Keccak/SHA-3
 - Customizable SHAKE (cSHAKE) is a SHAKE XOF that can be customized with a particular bit string to provide domain separation (conceptually similar to a "salt")
 - KMAC is a keyed MAC construction that is based on Keccak
 - **TupleHash** is a SHA-3-derived function that can be used to hash a tuple of input strings (that are uniquely serialized)
 - ParallelHash takes advantage of the parallelism available in some modern processors



- The sponge construction is based on a permutation operating on a data structure known as the state
- The state can either be seen as a (one-dimensional) b-bit string S or a three-dimensional array $\mathbf{A}[x, y, z]$ of bits with appropriate values for x, y, and z (i.e., $xyz \le b$)



6.4 Exemplary Hash Functions — Keccak/SHA-3

- In the case of SHA-3, b = 1600, $0 \le x, y < 5$, and $0 \le z < w$ (where $w = 2^l = 64$ for l = 6)
- Consequently, the state is either a 1600-bit string S or a $(5 \times 5 \times 64)$ -array **A** of 1600 bits
- For all $0 \le x, y < 5$ and $0 \le z < w$, the relationship between S and A is as follows:

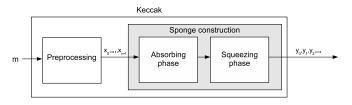
$$\mathbf{A}[x,y,z] = S[w(5y+x)+z]$$

■ **A**[0,0,0] translates to S[0], whereas **A**[4,4,63] translates to S[64((5 · 4) + 4) + 63] = S[64 · 24 + 63] = S[1599]

```
= A = plane[0] \parallel plane[1] \parallel \dots \parallel plane[4]
   = lane[0,0] || lane[1,0] || ... || lane[4,0] ||
           lane[0,1] \parallel lane[1,1] \parallel \dots \parallel lane[4,1] \parallel
           lane[0,2] \parallel lane[1,2] \parallel \dots \parallel lane[4,2] \parallel
           lane[0,3] \parallel lane[1,3] \parallel \dots \parallel lane[4,3] \parallel
           lane[0,4] \parallel lane[1,4] \parallel ... \parallel lane[4,4]
          bit[0,0,0] \parallel bit[0,0,1] \parallel bit[0,0,2] \parallel \ldots \parallel bit[0,0,63] \parallel
           bit[1,0,0] \parallel bit[1,0,1] \parallel bit[1,0,2] \parallel \dots \parallel bit[1,0,63] \parallel
           bit[2,0,0] \parallel bit[2,0,1] \parallel bit[2,0,2] \parallel \dots \parallel bit[2,0,63] \parallel
           bit[3, 4, 0] \parallel bit[3, 4, 1] \parallel bit[3, 4, 2] \parallel \dots \parallel bit[3, 4, 63] \parallel
           bit[4, 4, 0] \parallel bit[4, 4, 1] \parallel bit[4, 4, 2] \parallel \dots \parallel bit[4, 4, 63]
```

6.4 Exemplary Hash Functions — Keccak/SHA-3

- The sponge construction operates in 2 phases:
 - In the **absorbing** or **input phase**, the *n* message blocks $x_0, x_1, \ldots, x_{n-1}$ are consumed and read into the state
 - In the **squeezing** or **output phase**, an output $y_0, y_1, y_2, ...$ of configurable length is generated from the state



© keccak.team

- There are a few parameters to configure the input and output sizes as well as the security of Keccak
 - The **state width** *b* can take any value $b = 5 \cdot 5 \cdot 2^l = 25 \cdot 2^l$ for l = 0, 1, ..., 6 (i.e., 25, 50, 100, 200, 400, 800, or 1600 bits)
 - The **bit rate** *r* determines the number of input bits that are processed simultaneously
 - The capacity c refers to the double security level of the construction
- In either case, b = r + c



6.4 Exemplary Hash Functions — Keccak/SHA-3

Hash Function	n	Ь	r	с	W
SHA3-224	224	1600	1152	448	64
SHA3-256	256	1600	1088	512	64
SHA3-384	384	1600	832	768	64
SHA3-512	512	1600	576	1024	64

Note that b = 1600 and w = 64 in all versions of SHA-3

- Before a message m can be processed, it must be padded properly (to make sure that the input is a multiple of r bits long)
- It uses a padding scheme known as multirate padding

$$\mathsf{Padding}(m) = \underbrace{m \parallel p \parallel 10^*1}_{\mathsf{multiple} \ \mathsf{of} \ r}$$

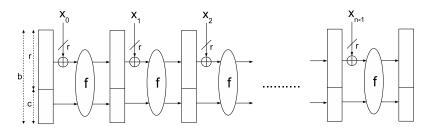
- The value of bit string p depends on the mode
 - 2-bit string 01 for hashing
 - 4-bit string 1111 for generating a variable-length output

- The sponge construction is based on a permutation of the state (called *f*-function or *f*-permutation)
- The same *f*-function is used in the absorbing and squeezing phases
- It takes b = r + c bits as input and generates an output of the same length
- Internally, the f-function consists of n_r round functions with the same input and output behavior

- Remember that I determines the state width according to $b = 25 \cdot 2^{I}$ (SHA-3 uses the fixed values I = 6 and hence b = 1600)
- The value I also determines n_r , i.e., the number of rounds, according to $n_r = 12 + 2I$
- So the possible state widths 25, 50, 100, 200, 400, 800, and 1600 come along with respective numbers of rounds, i.e., 12, 14, 16, 18, 20, 22, and 24
- As SHA-3 fixes the state width to 1600 bits, the number of rounds is also fixed to 24, i.e., $n_r = 24$

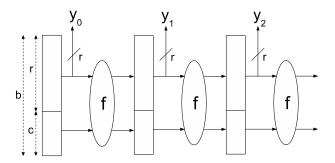
6.4 Exemplary Hash Functions — Keccak/SHA-3

■ Keccak absorbing phase



6.4 Exemplary Hash Functions — Keccak/SHA-3

■ Keccak squeezing phase



- In each round, a sequence of five step mappings is executed, where each mapping operates on the *b* bits of the state
- Each step mapping takes a state array $\bf A$ as input and returns an updated state array $\bf A'$ as output
- The five step mappings are denoted by Greek letters, i.e., theta (θ) , rho (ρ) , pi (π) , chi (χ) , and iota (ι)
- While θ must be applied first, the order of the other mappings is arbitrary and does not matter (and ρ and π are often applied simultaneously)

- The step mappings are relatively simple to capture visually, but more difficult to capture mathematically
- The x- and y-axes are labeled in an unusual manner

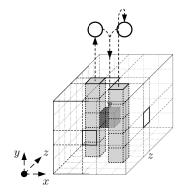
Table 6.7 The (x, y)-Coordinates of the Bits in a Slice

(3,2)	(4,2)	(0,2)	(1,2)	(2,2)
(3,1)	(4,1)	(0,1)	(1,1)	(2,1)
(3,0)	(4,0)	(0,0)	(1,0)	(2,0)
(3,4)	(4,4)	(0,4)	(1,4)	(2,4)
(3,3)	(4,3)	(0,3)	(1,3)	(2,3)

- The step mappings mainly operate on lanes, i.e., w-bit words that can be processed in a register on a modern processor $(lane[x, y] \text{ refers to } \mathbf{A}[x, y, \cdot])$
- The (mathematical) operations include the addition and multiplication modulo 2, i.e., the bitwise addition and multiplication in GF(2)
- This suggests that the addition is equal to the Boolean XOR operation (\oplus) and the multiplication is equal to the Boolean AND operation (\land)
- With the exception of the round constants $RC[i_r]$ used in ι (iota), the step mappings are the same in all rounds

6.4 Exemplary Hash Functions — Keccak/SHA-3

■ Step mapping θ (theta)



© keccak.team

$$\mathbf{A}'[x_0,y_0,z_0] = \mathbf{A}[x_0,y_0,z_0]$$

$$\mathbf{A}'[x_0, y_0, z_0] = \mathbf{A}[x_0, y_0, z_0] \quad \oplus \quad \bigoplus_{y=0}^4 \mathbf{A}[(x_0 - 1) \bmod 5, y, z_0]$$

$$\bigoplus_{y=0}^{4} \mathbf{A}[(x_0+1) \bmod 5, y, (z_0-1) \bmod w]$$

6.4 Exemplary Hash Functions — Keccak/SHA-3

■ Algorithm to compute θ (theta)

```
(A)
```

```
\begin{array}{l} \text{for } \ x=0 \ \text{to 4 do} \\ \text{for } \ z=0 \ \text{to } w-1 \ \text{do} \\ \text{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]} \\ \text{for } \ x=0 \ \text{to 4 do} \\ \text{for } \ z=0 \ \text{to } w-1 \ \text{do} \\ \text{D[x,z]} = \ \text{C[(x-1) mod 5,z]} \oplus \text{C[(x+1) mod 5,(z-1) mod } w] \\ \text{for } \ x=0 \ \text{to 4 do} \\ \text{for } \ y=0 \ \text{to 4 do} \\ \text{for } \ z=0 \ \text{to } w-1 \ \text{do} \\ \text{A'[x,y,z]} = \ \mathbf{A[x,y,z]} \oplus \text{D[x,z]} \end{array}
```

(A')

6.4 Exemplary Hash Functions — Keccak/SHA-3

- Step mapping ρ (rho) rotates the bits in each lane for a certain amount of bits (offset), while step mapping π (pi) permutes the position of the lanes
- Both mappings can be combined and expressed as

$$lane[y, 2x + 3y] = lane[x, y] \stackrel{\curvearrowleft}{\hookrightarrow} r[x, y]$$

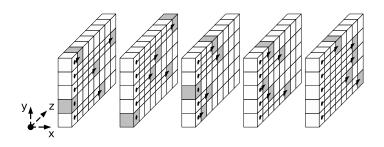
or

$$\mathbf{A}'[y,2x+3y,\cdot] = \mathbf{A}[x,y,\cdot] \stackrel{\curvearrowleft}{\hookrightarrow} r[x,y]$$



6.4 Exemplary Hash Functions — Keccak/SHA-3

■ Step mapping ρ (rho)



© keccak.team

	x = 3	x = 4	x = 0	x = 1	x = 2
y=2	25	39	3	10	43
y=1	55	20	36	44	6
y=0	28	27	0	1	62
y=4	56	14	18	2	61
y=3	21	8	41	45	15

6.4 Exemplary Hash Functions — Keccak/SHA-3

■ Algorithm to compute ρ (rho)

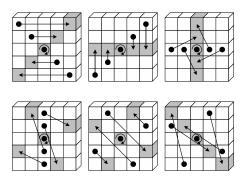
(A)

for
$$z = 0$$
 to $w - 1$ do $\mathbf{A}'[0, 0, z] = \mathbf{A}[0, 0, z]$
 $(x, y) = (1, 0)$
for $t = 0$ to 23 do
for $z = 0$ to $w - 1$ do $\mathbf{A}'[x, y, z] = \mathbf{A}[x, y, (z - (t+1)(t+2)/2) \mod w]$
 $(x, y) = (y, (2x + 3y) \mod 5)$

(A')

6.4 Exemplary Hash Functions — Keccak/SHA-3

• Step mapping π (pi)



© keccak.team

6.4 Exemplary Hash Functions — Keccak/SHA-3

■ Algorithm to compute π (pi)

```
(A)

for x = 0 to 4 do

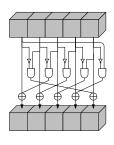
for y = 1 to 4 do

for z = 0 to w - 1 do \mathbf{A}'[x, y, z] = \mathbf{A}[(x + 3y) \mod 5, x, z]

(A')
```

6.4 Exemplary Hash Functions — Keccak/SHA-3

- Step mapping χ (chi) operates on lanes
- It combines lane[x, y] with lane[x + 1, y] and lane[x + 2, y] with the Boolean NOT (\neg) XOR (\oplus) , and AND (\land) operators
- It is the only nonlinear step mapping in the round function of KECCAK



© keccak.team

6.4 Exemplary Hash Functions — Keccak/SHA-3

$$\mathbf{A}'[x,y,\cdot] = \mathbf{A}[x,y,\cdot] \oplus ((\neg \mathbf{A}[x+1,y,\cdot]) \wedge \mathbf{A}[x+2,y,\cdot])$$

■ Algorithm to compute χ (chi)

```
for x = 0 to 4 do for y = 1 to 4 do for z = 0 to w - 1 do for z = 0 to w - 1 do \mathbf{A}'[x, y, z] = \mathbf{A}[x, y, z] \oplus ((\mathbf{A}[(x+1) \bmod 5, y, z] \oplus 1) \cdot \mathbf{A}[(x+2) \bmod 5, y, z]) (A')
```

◆ロ > ◆回 > ◆ き > ◆き > ・ き ・ り Q (

6.4 Exemplary Hash Functions — Keccak/SHA-3

- Step mapping ι (iota) adds modulo 2 a round-dependent constant $RC[i_r]$ to lane[0,0] and leaves all other 24 lanes unchanged
- The round constants $RC[i_r]$ (for $i_r = 0, ..., 23$) are constructed in a particular way (not addressed here)

$$\mathbf{A}'[0,0,\cdot] = \mathbf{A}[0,0,\cdot] \oplus \mathsf{RC}[i_r]$$



6.4 Exemplary Hash Functions — Keccak/SHA-3

Table 6.9 (new)

The 24 Round Constants $RC[i_r]$ Employed by SHA-3

RC[0]	0×0000000000000001	RC[12]	0×000000008000808B
RC[1]	0×0000000000008082	RC[13]	0×800000000000008B
RC[2]	0×800000000000808A	RC[14]	0×8000000000008089
RC[3]	0x8000000080008000	RC[15]	0×8000000000008003
RC[4]	0x000000000000808B	RC[16]	0×8000000000008002
RC[5]	0×0000000080000001	RC[17]	0×800000000000000000
RC[6]	0x8000000080008081	RC[18]	0×000000000000800A
RC[7]	0×8000000000008009	RC[19]	A0000008000000A
RC[8]	0×000000000000008A	RC[20]	0×8000000080008081
RC[9]	0×0000000000000088	RC[21]	0×8000000000008080
RC[10]	0×0000000080008009	RC[22]	0×0000000080000001
RC[11]	0×0000000008000000A	RC[23]	0×80000000080008

6.4 Exemplary Hash Functions — Keccak/SHA-3

• Given a state **A** and round index i_r , the round function Rnd is defined as

$$\operatorname{Rnd}(\mathbf{A}, i_r) = \iota(\chi(\pi(\rho(\theta(\mathbf{A})))), i_r)$$

■ The Keccak- $p[b, n_r]$ permutation consists of n_r iterations of Rnd:

convert
$$S$$
 into state \mathbf{A} for $i_r = 2l + 12 - n_r, \dots, 2l + 12 - 1$ do $\mathbf{A} = \operatorname{Rnd}(\mathbf{A}, i_r)$ convert \mathbf{A} into b -bit string S'

6.4 Exemplary Hash Functions — Keccak/SHA-3

- The Keccak-f family of permutations refers to the specialization of the Keccak-p family with $n_r = 12 + 12l$: Keccak-f[b] = Keccak-p[b, 12 + 2l]
- The Keccak-p[1600, 24] permutation that underlies the six SHA-3 functions is equivalent to Keccak-f[1600]
- There is no known attack against Keccak/SHA-3
- But Keccak/SHA-3 is still not widely deployed in the field

- Most cryptographic hash functions in use today follow the Merkle-Damgård construction and are iterated
- Consequences
 - Since each iteration can only start if the preceding iteration has finished, the hash function may become a performance bottleneck
 - The design of compression functions that are collision-resistant is still more of an art than a science (i.e., it lacks theoretical foundations)
- Against this background, people come up with ad hoc designs

- Sometimes, people try to improve collision resistance by concatenating two (or more) hash functions
- For example, instead of using MD5 or SHA-1 alone, they may apply one function after the other (e.g., SSL 3.0)
- Intuition suggests that the resulting (concatenated) hash function is more collision-resistant than each function applied individually
- In 2004, it was shown that intuition is illusive and wrong
- Since then, concatenating different hash functions is no longer used in the field

- An alternative design for cryptographic hash functions was proposed by Larry Carter and Mark Wegman in late 1970s
- Instead of using a single hash function, it uses families of such functions from which a specific function is randomly selected
- Such a family H consists of all hash functions $h: X \to Y$ that map values from X to values from Y
- *H* is called **two-universal**, if for every $x, y \in X$ with $x \neq y$

$$\Pr[h(x) = h(y)] \le \frac{1}{|Y|}$$





- This suggests that the images are uniformly distributed in Y, and that the probability of having two images collide is as small as possible (given the size of Y)
- This notion of universality can be generalized
- Using (two-)universal families of hash functions is referred to as universal hashing
- Universal hashing is the basic ingredient for Carter-Wegman MACs (as further addressed in Chapter 10)



Questions and Answers



Thank you for your attention

