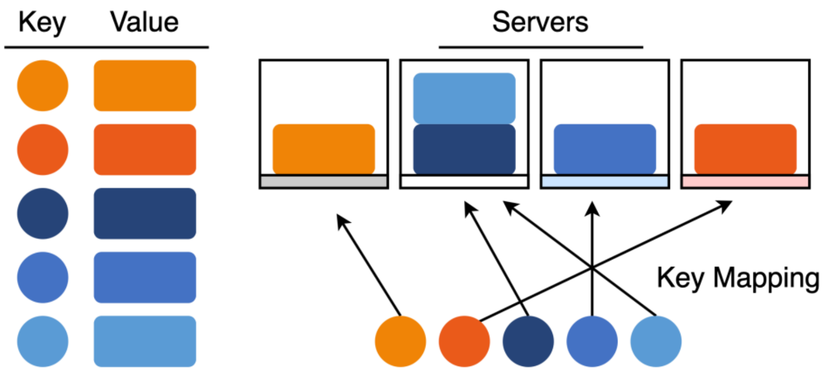
Cryptographic Hash Functions

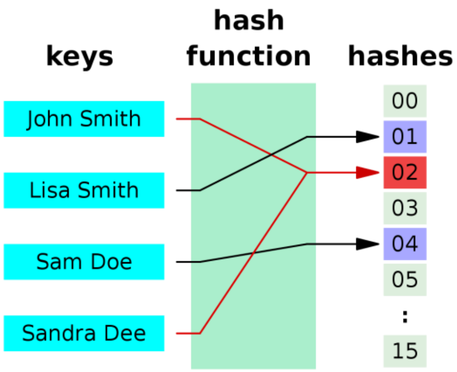
Dhiren Patel

(22 Feb 2023, 28/29 Feb 2023)

Hash function (HF)

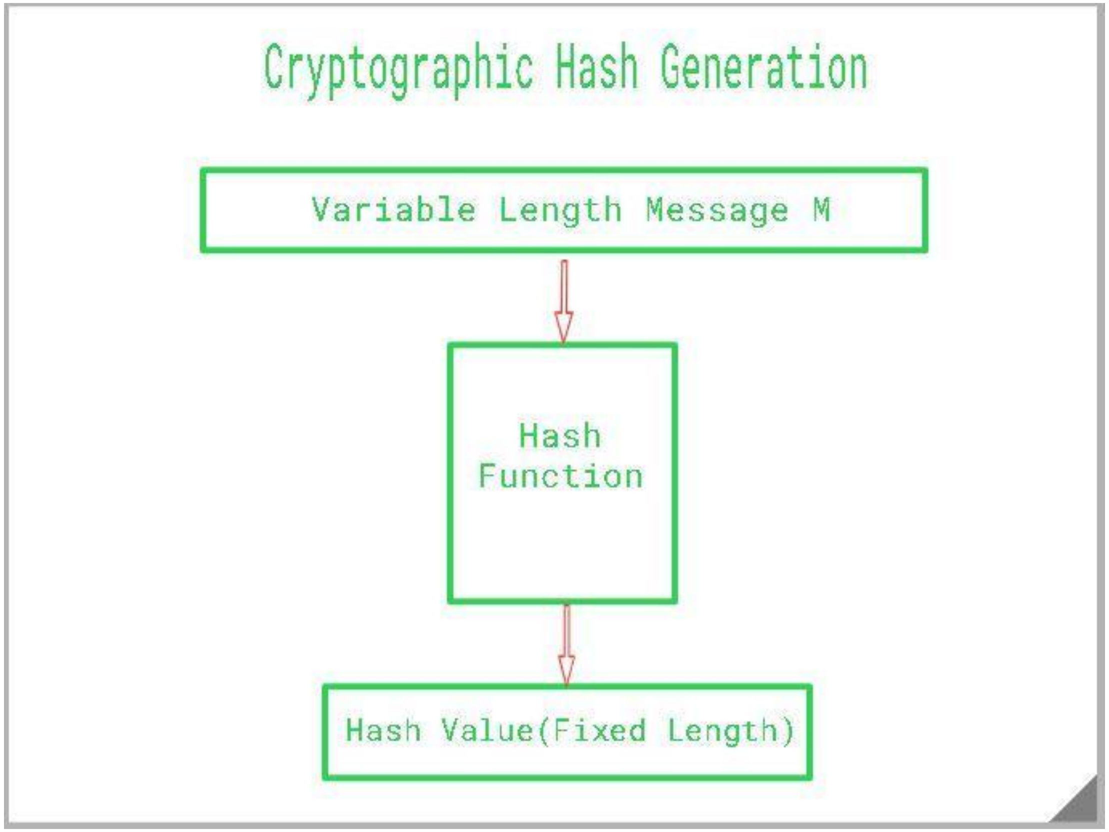
- function that can be used to map data of arbitrary size to fixed-size values





Cryptographic Hash Function - CHF

- takes random (any, variable) size input and yields a fixed-size output
- Output is known as hash code, hash values, hash, message digest, message finger print/thumb print etc.
- a mathematical function or algorithm that takes a variable number of characters (called a ”message”) and converts it into a string with a fixed number of characters

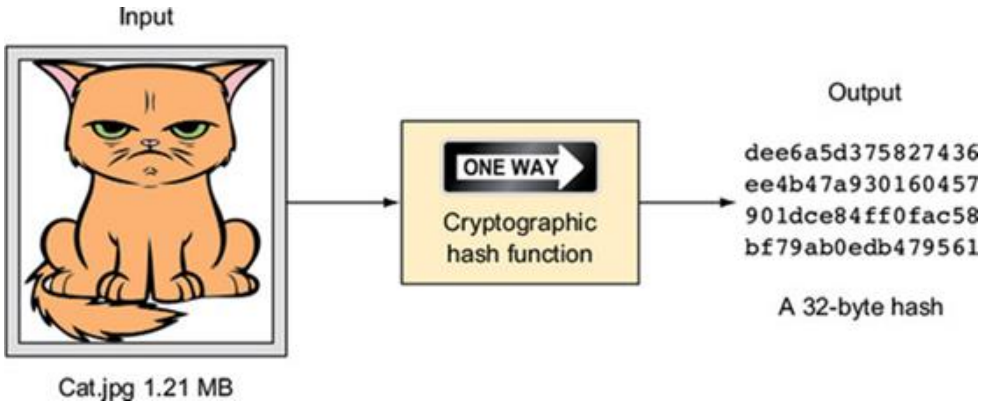


Cryptographic Hash Generation

Variable Length Message M

Hash Function

Hash Value(Fixed Length)



Properties

- **Deterministic:** the same message always results in the same hash.
- **Quick:** It is quick to compute the hash value for any given message.
- **Avalanche Effect:** every minor change in the message results in a major change in the hash value.
- **One-Way Function:** You cannot reverse the cryptographic hash function to get to the data.
- **Collision Resistance:** It is infeasible to find two different messages that produce the same hash value.
- **Non Predictable:** The hash value shouldn’t be predictable from the given string and vice versa.

Data integrity check

- Hashing is useful to ensure the authenticity of a piece of data and that it has not been tampered with since even a small change in the message will create an entirely different hash
- Rather than compare the data in its original (and larger) form, by comparing the two hashes of the data, computers can quickly confirm that the data has not been tampered with and changed.
- Hash functions are the basic tools of modern cryptography that are used in information security to authenticate transactions, messages, and digital signatures

One Way Hash Function - OWHF

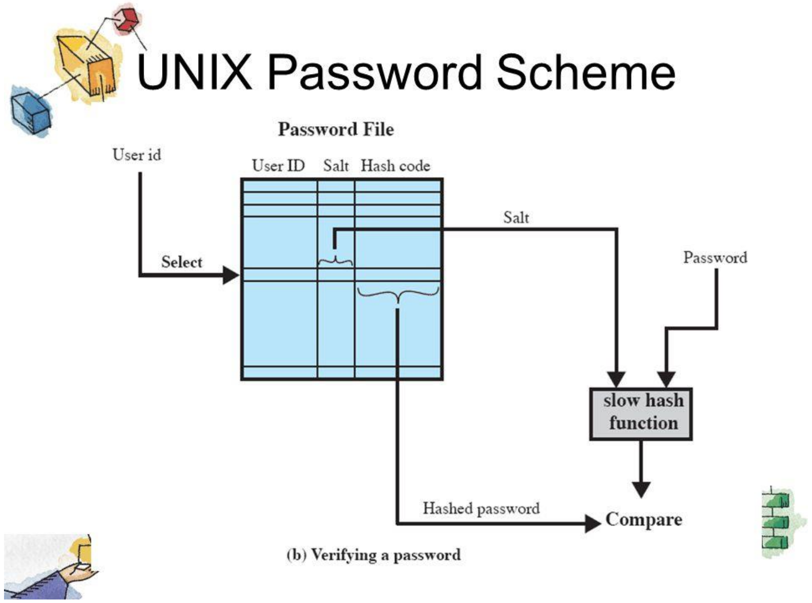
- a one-way function (irreversible), which means that it is easy to convert a message into a hash but very difficult to “reverse hash” a hash value back to its original message
- If a hash function produces the same output from two different pieces of data, it is known as a “hash collision,”
- Brute-force-attack - using trial and error to find a message that fits the hash value and see if it produces a match

HF Applications

- very popular tools for cryptographic applications such as
- Data Integrity Verification or message integrity check,
- message authentication,
- digital signature,
- **software distribution, - example - license keys**
- protection scheme for pass-phrases or passwords
- electronic funds transfer, data storage, and other applications where data integrity is very important

Password Storing – Challenge Response

- Instead of keeping passwords in plain text, it is possible to store only a hash of the passwords in databases.
- So, if attackers get access to the database, they will need to process the hashes (typically, using brute force or **rainbow tables** strategies) to recover the passwords



(b) Verifying a password

Commitment Protocol

- Goal: A and B wish to play “odd or even” over the network
- Naive Commitment Protocol
- A picks a number X and sends it to B
- B picks a number Y and sends it to A
- A wins if X+Y is odd
- B wins if X+Y is even
- Problem: How can we guarantee that B doesn’t cheat?

Commitment Protocols with Hash function

- A picks a number X and sends value of Z = H(X) to B
- B picks a number Y and sends value of Y to A
- A now sends value of X to B
- B checks if X complies with Z that was sent before
- A wins if X+Y is odd
- B wins if X+Y is even
- Solution: In this protocol B cannot cheat

Commitment Protocols with Hash

- Hash function does two things in the protocol:
- Hides the number X from B at the beginning of the game
- Makes A commit to the number X until the end of the game
- Question: What if A always picks small numbers so that B can make a list of all the hash values?
- Answer: A should select random values for the protocol:
- Select the number X from a very large space of numbers
- Mask the number X with a random noise from a very large space

Commitment schemes

- a kind of digital envelope
- Allows one party to “commit” to a message m by sending a commitment c to the counterparty
- Set $c = H(m \,||\, r)$ where r is a random n-bit string
- Hiding: c reveals nothing about m
- Binding: Infeasible for c to be opened to a different message m'

Digital Signature

- As an electronic analogue of a written signature, a digital signature provides assurance that:
- *the claimed signatory signed the information, and*
- *the information was not modified after signature generation.*
- you can sign the hash of a document by encrypting it with your private key, producing a digital signature for the document.
- Anyone else can then check that you authenticated the text by decrypting the signature with your public key to obtain the original hash again, and comparing it with their hash of the text.
- FIPS 186-4 DSS specifies three NIST-approved digital signature algorithms: **DSA**, **RSA**, and **ECDSA**

Block Ciphers v/s Hash Functions

- ❑ A block cipher (is a function which maps) n-bit plaintext blocks to n-bit ciphertext blocks; n is called the *block length*.
  - – E: $\{0,1\}^n \times \{0,1\}^k \rightarrow \{0,1\}^n$
- ❑ To allow unique decryption, the encryption function must be one-to-one (i.e., invertible)

- ❑ Hash Function - takes input (of variable-length) and returns a fixed size output string h (usually much smaller than input)

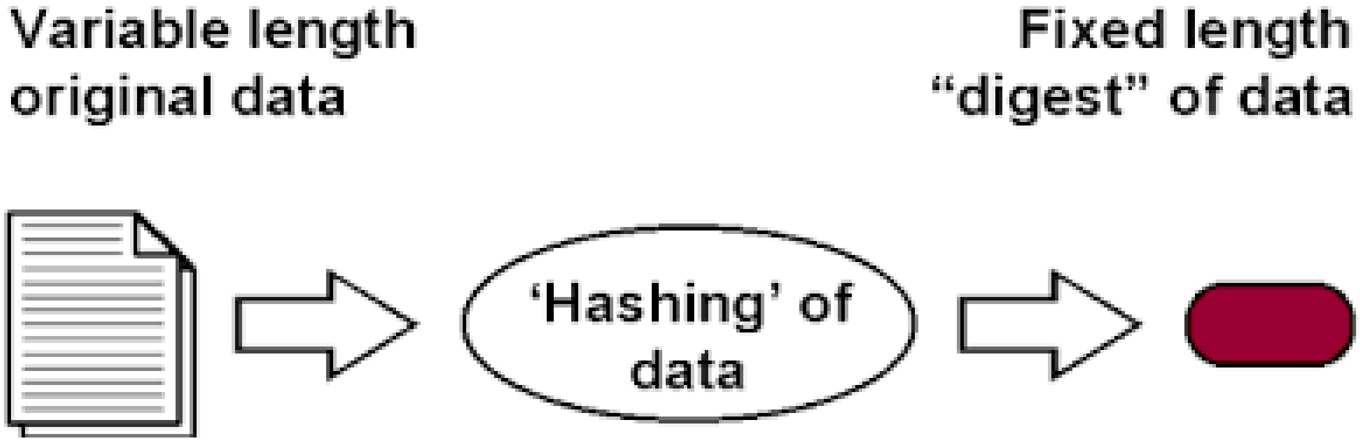
H: $\{0,1\}^* \rightarrow \{0,1\}^n$, $h = \mathrm{H}(M)$

- ❑ One way

Demo – HF (MD5, SHA-1, SHA-256)

- https://www.tools4noobs.com/online_tools/hash/
- (https://emn178.github.io/online-tools/SHA256)

How Hash functions are useful in Data integrity applications?



- Plain text, Hash (MITM - Attack)
- Plain text, Shared Secret, Hash (MAC)
- HF and Keyed HF?
- Digital Signature (later)

Building Cryptographic Hash Function

- *Operability:*
- H() should work on any input length
- H() should produce output of fixed size
- H() should be easy to compute

Additional Properties

- Compression → collisions
- a collision happens when different data inputs result in the same hash after being processed by a hashing mechanism
- Sparse over large input space
- More bits – output lookup table too large
- Weak collision resistance
- Strong collision resistance
- Weak collision resistance is bound to a particular input, whereas strong collision resistance applies to any two arbitrary inputs.

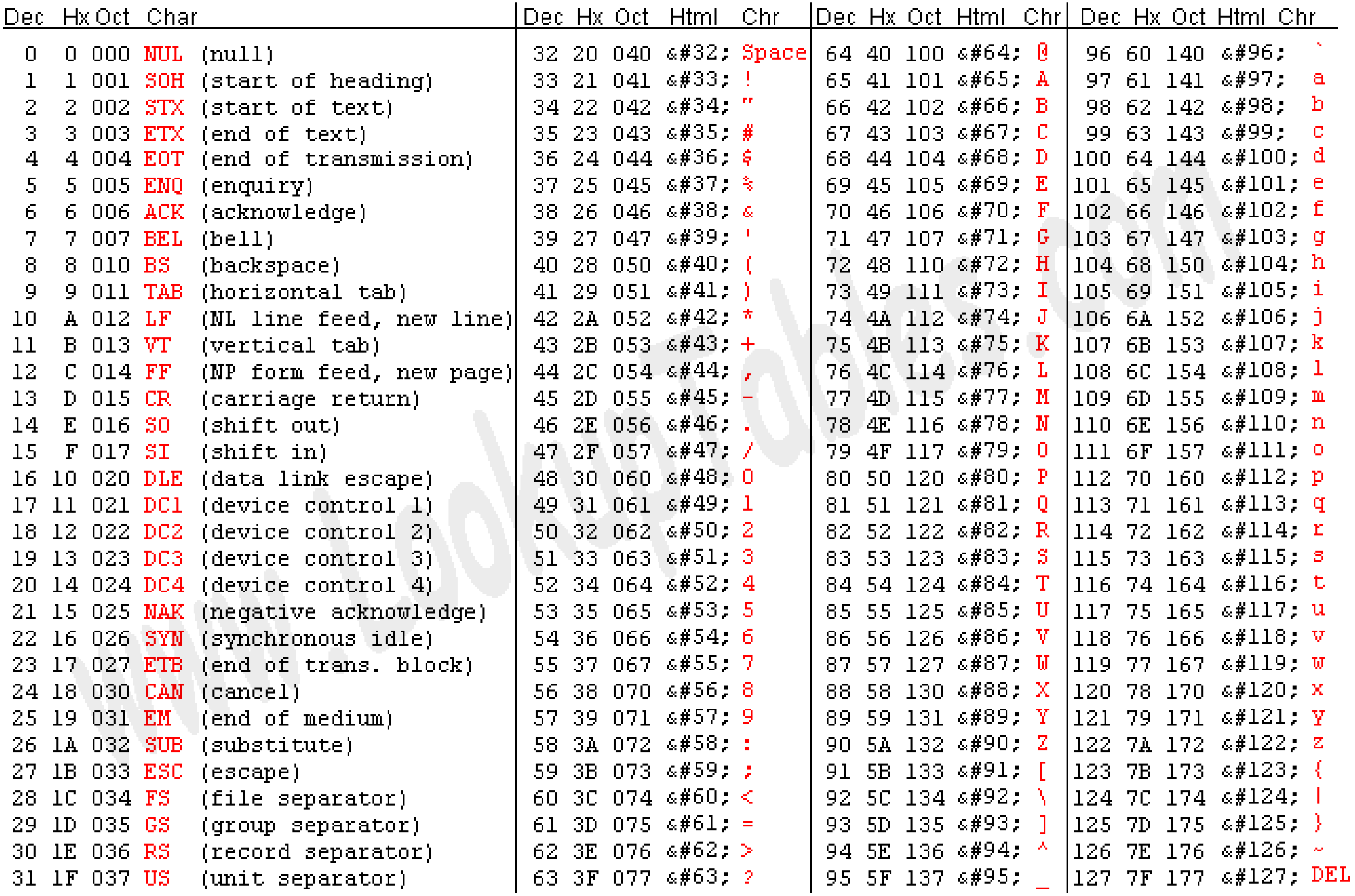
Definition

- Collision resistance is the property of a hash function that it is computationally infeasible to find two colliding inputs.
- **Weak collision resistance**: given an input X and a hashing function H(), it is very difficult to find another input X’ on which H(X) = H(X’).
- **Strong collision resistance** - given a hashing function H() and two arbitrary inputs X and Y, there exists an absolute minimum chance of H(X) being equal to H(Y).

2 bit HF v/s 32 bit HF (input size?)

- 00
- 11
- 10
- 01

CheckSum (CRC) as a HF --- too weak?



Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH	(start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX	(start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX	(end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT	(end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ	(enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK	(acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL	(bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS	(backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB	(horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT	(vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR	(carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO	(shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI	(shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE	(data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1	(device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2	(device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3	(device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4	(device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK	(negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN	(synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB	(end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN	(cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM	(end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB	(substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC	(escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS	(file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS	(group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS	(record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US	(unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

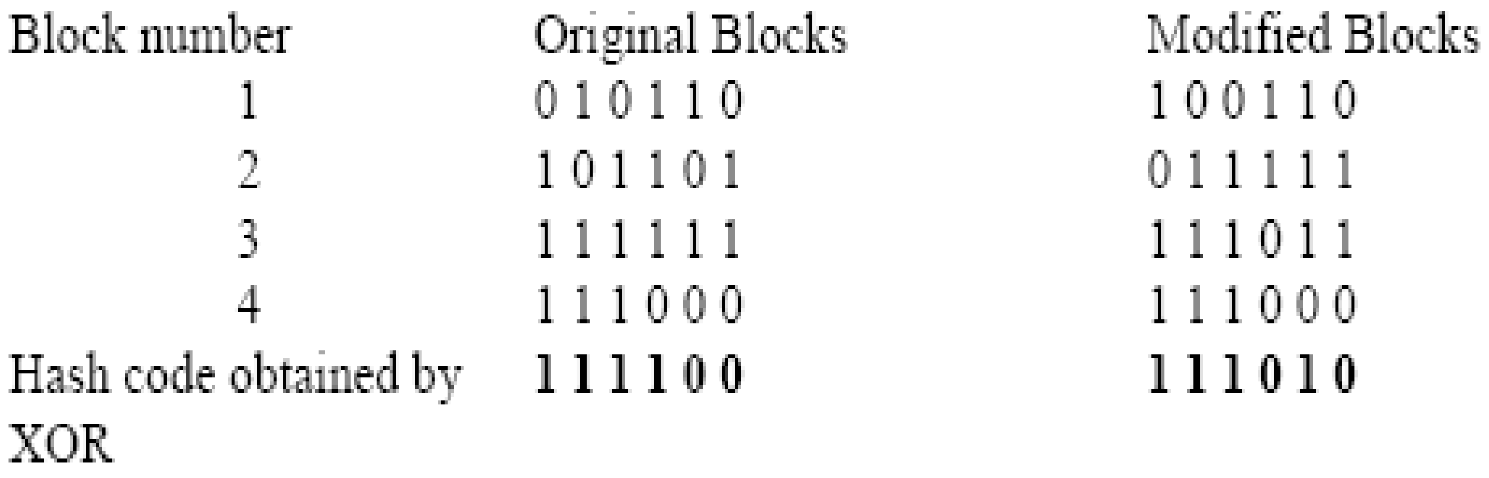
Source: www.LookupTables.com

Example: CRC manipulation

- Anil has mo g/p e/a t/y.

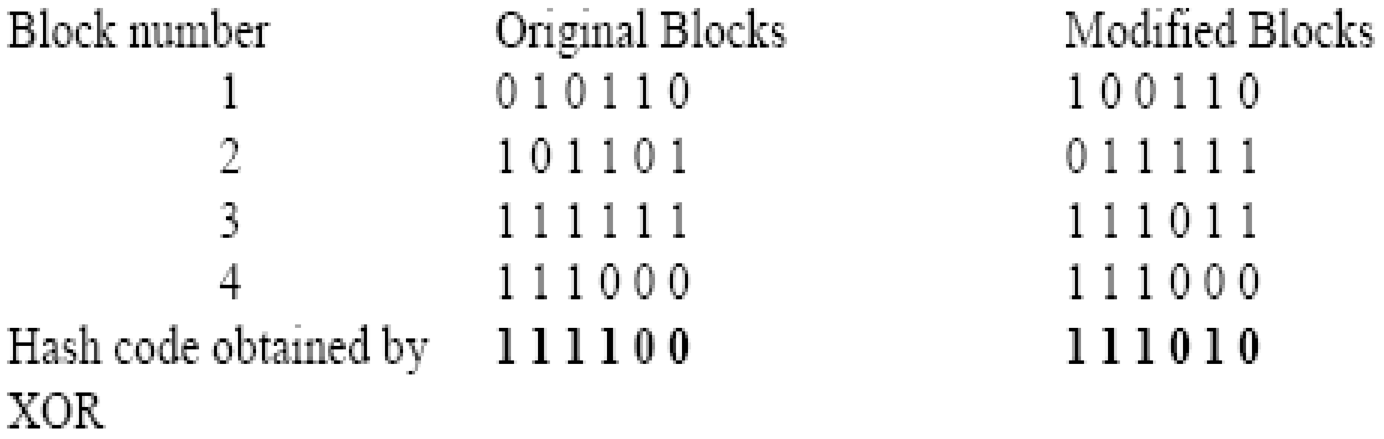
Building a hash function - example

- Simple Hash Function - based on XOR of message blocks

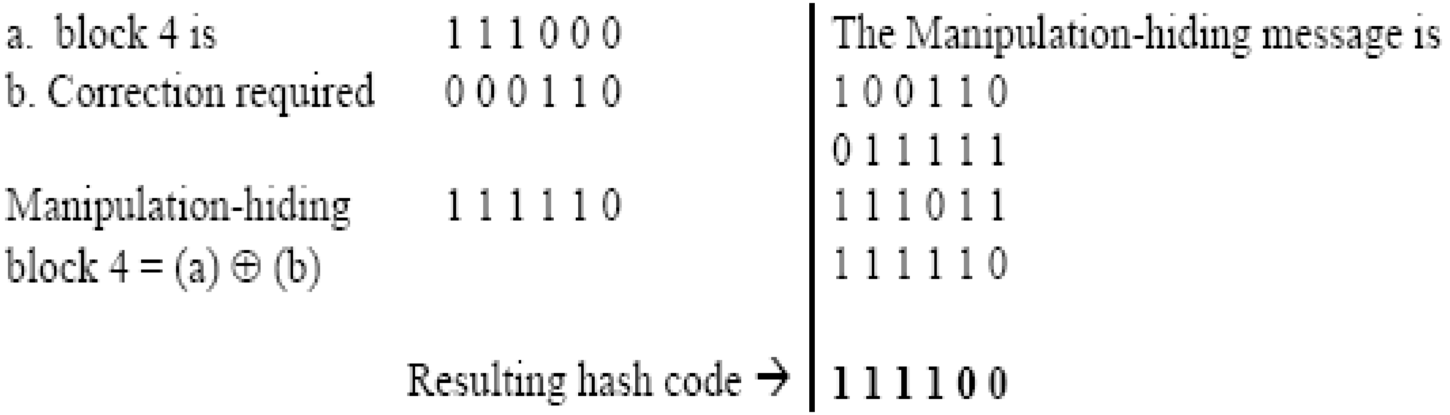


Block number	Original Blocks	Modified Blocks
1	0 1 0 1 1 0	1 0 0 1 1 0
2	1 0 1 1 0 1	0 1 1 1 1 1
3	1 1 1 1 1 1	1 1 1 0 1 1
4	1 1 1 0 0 0	1 1 1 0 0 0
Hash code obtained by XOR	**1 1 1 1 0 0**	**1 1 1 0 1 0**

not secure as manipulation is easy for any message



Block number	Original Blocks	Modified Blocks
1	0 1 0 1 1 0	1 0 0 1 1 0
2	1 0 1 1 0 1	0 1 1 1 1 1
3	1 1 1 1 1 1	1 1 1 0 1 1
4	1 1 1 0 0 0	1 1 1 0 0 0
Hash code obtained by XOR	**1 1 1 1 0 0**	**1 1 1 0 1 0**



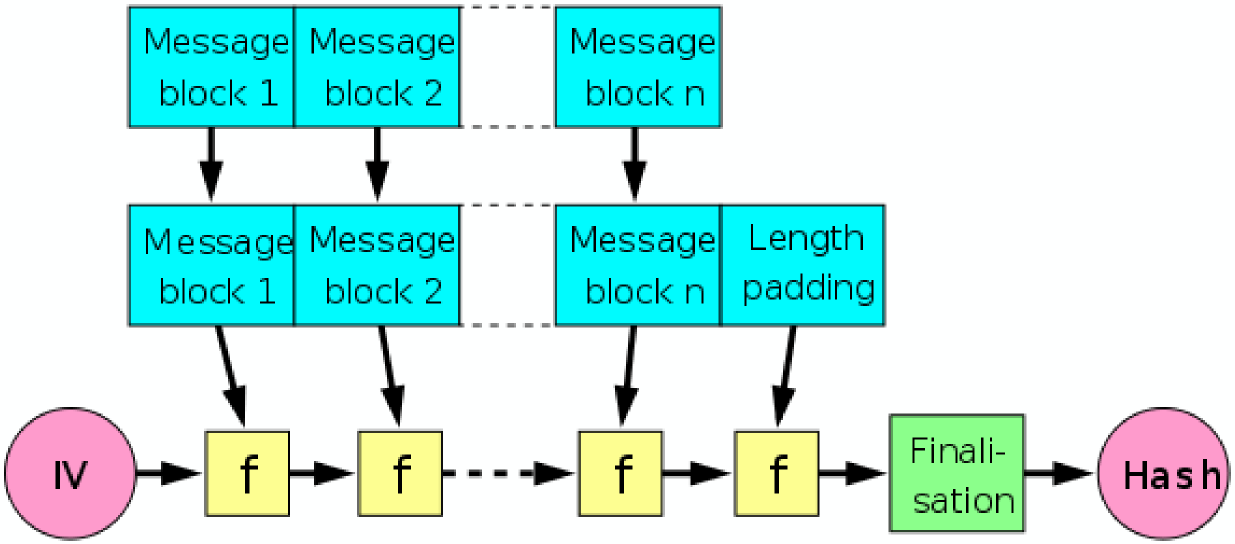
a. block 4 is	1 1 1 0 0 0	The Manipulation-hiding message is
b. Correction required	0 0 0 1 1 0	1 0 0 1 1 0
		0 1 1 1 1 1
Manipulation-hiding block 4 = (a) ⊕ (b)	1 1 1 1 1 0	1 1 1 0 1 1
		1 1 1 1 1 0
	Resulting hash code →	**1 1 1 1 0 0**

Hash Function construction

Merkle-Damgard:

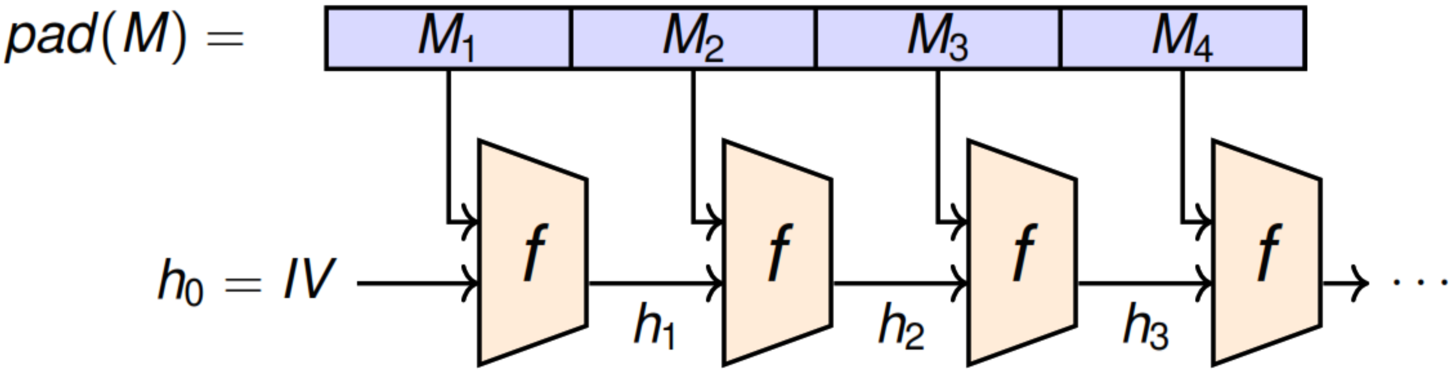
iterative application of compression function

- MD-strengthening → The procedure of fixing the *IV* and adding a representation of the length of input.



Merkel Damgard transformation

- a method of building collision resistant cryptographic hash functions from collision-resistant one-way compression functions



Padding

- Pad 1111111111
- Or Pad 1000000000
- Or something else

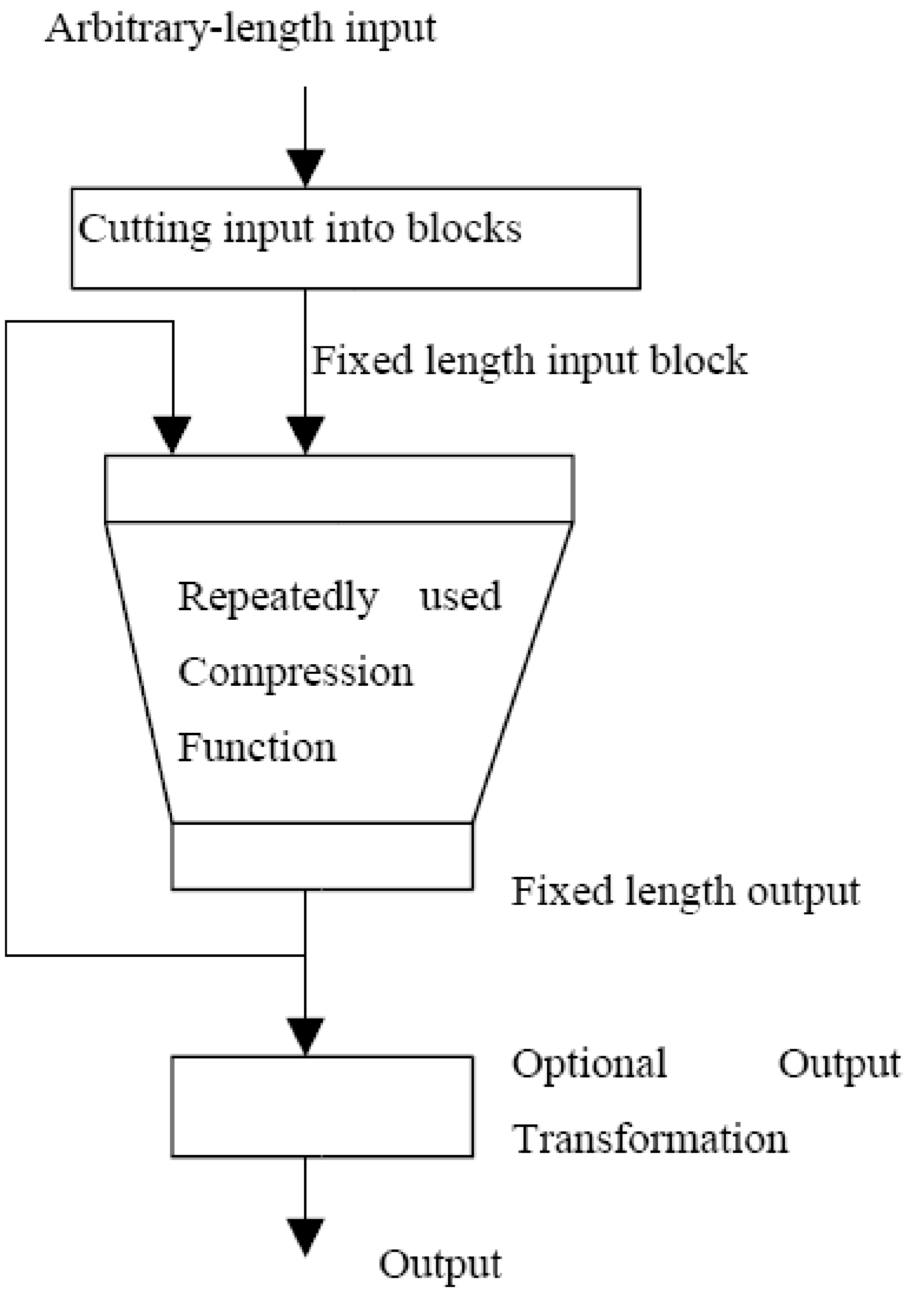
Padding in SHA256 (includes length padding)

- Let input M be l bits long
    - Find smallest non-negative k such that

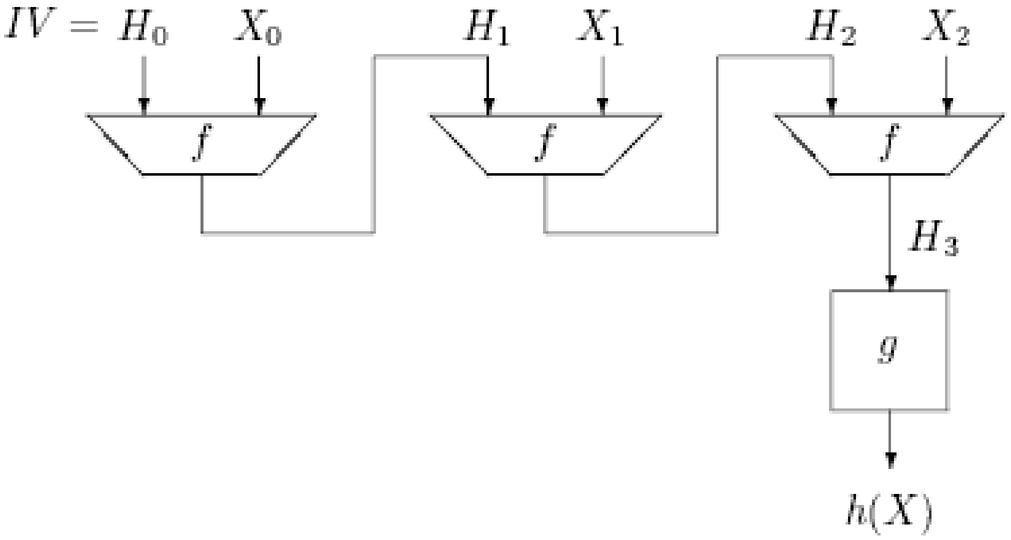
$$k + l + 65 = 0 \bmod 512$$

    - Append $k + 1$ bits consisting of single 1 and k zeros
    - Append 64-bit representation of l
- Example: $M = 101010$ with $l = 6$
    - $k = 441$
    - 64-bit representation of 6 is $000 \cdots 00110$
    - 512-bit padded message

$$\underbrace{101010}_{M}\ 1\ \underbrace{00000 \cdots 00000}_{441 \text{ zeros}}\ \underbrace{00 \cdots 00110}_{l}.$$

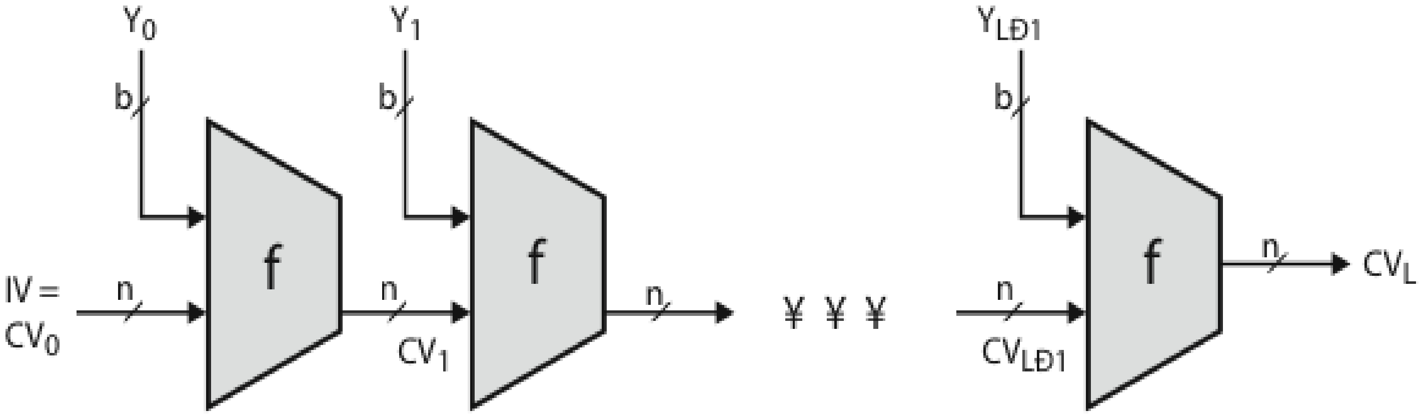

Arbitrary-length input
Cutting input into blocks
Fixed length input block
Repeatedly used Compression Function
Fixed length output
Optional Output Transformation
Output

HF



$$\begin{aligned} H_0 &= IV \ , \\ H_{i+1} &= f(H_i, X_i) \quad \text{for } 0 \leq i < t \ , \\ h(X) &= g(H_t) \ . \end{aligned}$$

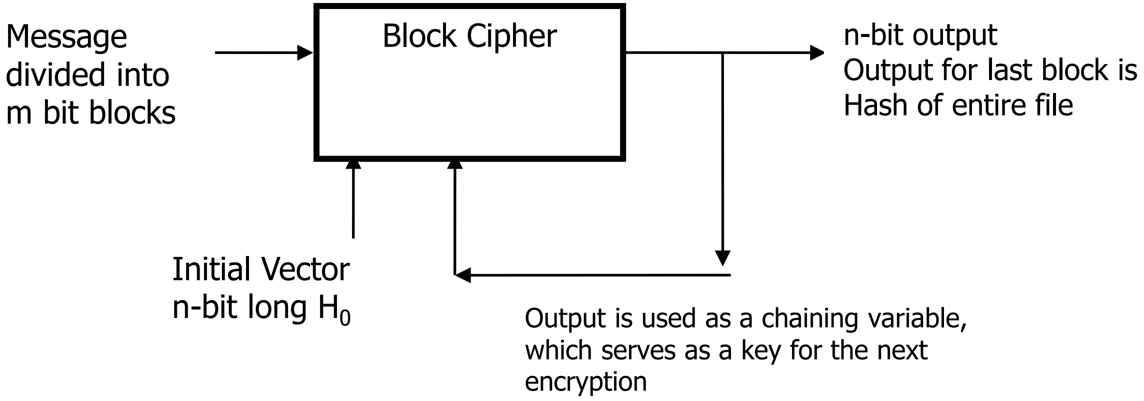
HF



IV	=	Initial value	L	=	number of input blocks
CV_i	=	chaining variable	n	=	length of hash code
Y_i	=	ith input block	b	=	length of input block
f	=	compression algorithm			

HF construction – *using Block Cipher*

- Block cipher (standard or dedicated) in CBC
- Hash function H: $\{0,1\}^* \rightarrow \{0,1\}^n$
- Block cipher encryption E: $\{0,1\}^n \times \{0,1\}^k \rightarrow \{0,1\}^n$
- $H_i = H_{i-1} \oplus M_i$

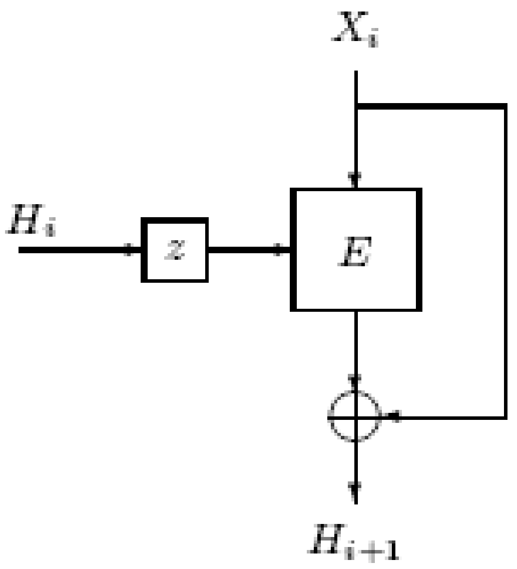

Block Cipher
Message divided into m bit blocks
n-bit output
Output for last block is Hash of entire file
Initial Vector
n-bit long H_0
Output is used as a chaining variable, which serves as a key for the next encryption

Hash Function Using A Block Cipher

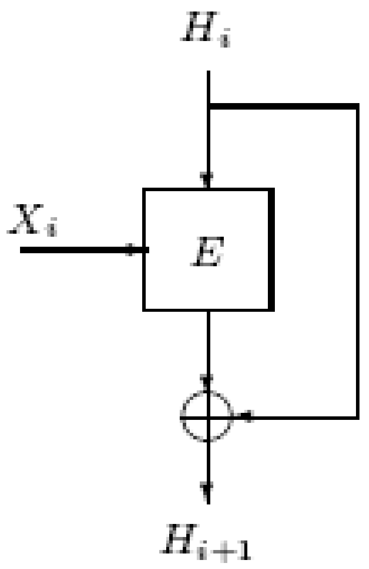
$$H_i = E_{H_{(i-1)}}(B_i),$$

$$H(M) = E_{H_{(n-1)}}(B_n)$$

Using Block cipher



(a) Matyas-Meyer-Oseas



(b) Davies-Meyer

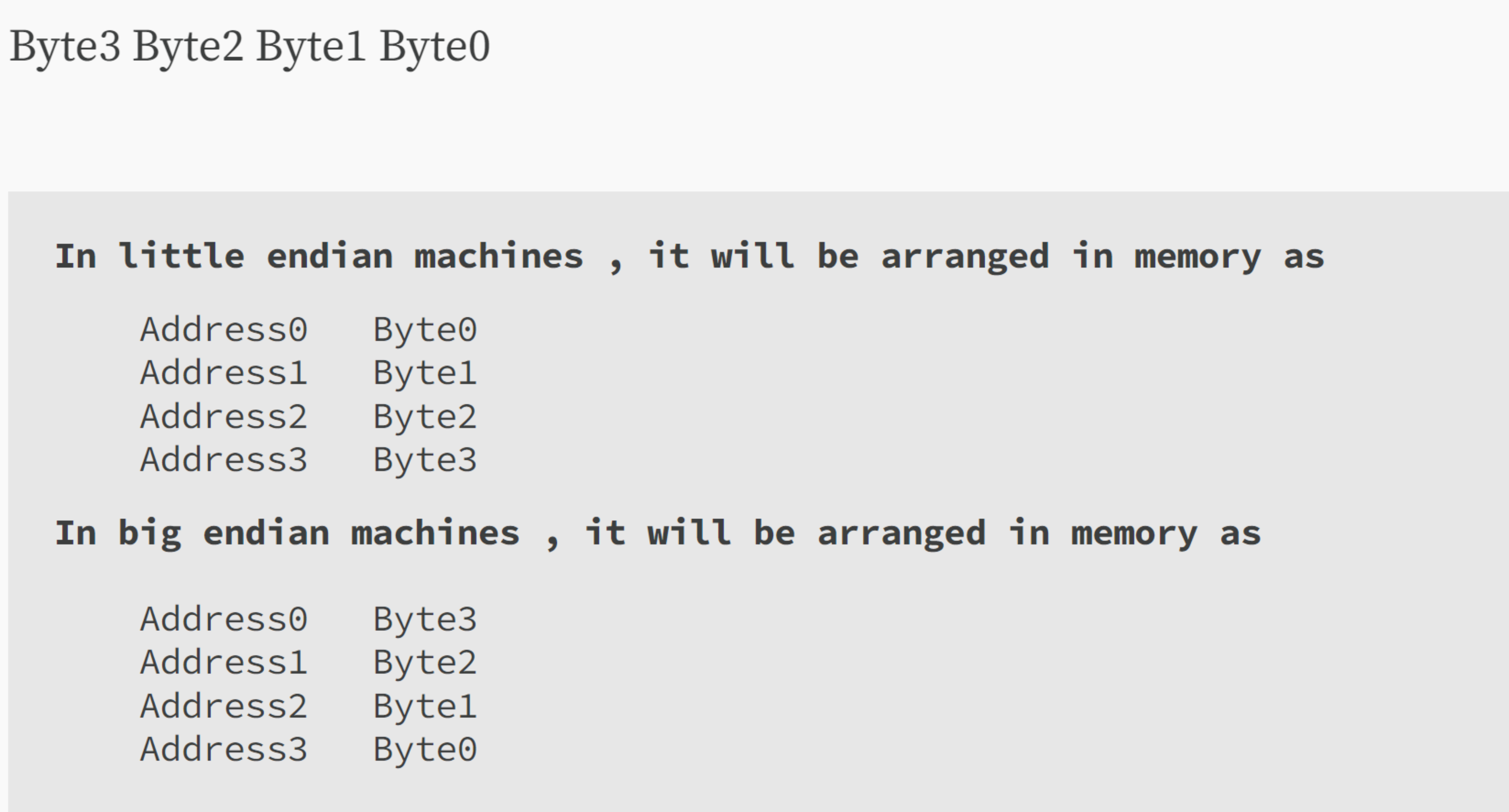
HF construction

- Dedicated function with optimized performance
- Operates on iterative compression function
- based on 32-bit Registers/buffers, S-Boxes
- with multiple rounds of computations

Essential parameters

- Message pre-processing
- Chaining variable and hash output
- Collision-resistance of the compression function
- Word-orientation - Little-endian or big-endian conversion
- Sequential structure
- Message expansion

Little Endian and Big Endian format



Byte3 Byte2 Byte1 Byte0

```
In little endian machines , it will be arranged in memory as

    Address0    Byte0
    Address1    Byte1
    Address2    Byte2
    Address3    Byte3

In big endian machines , it will be arranged in memory as

    Address0    Byte3
    Address1    Byte2
    Address2    Byte1
    Address3    Byte0
```

Hash Functions Security

- **cryptanalytic attacks** exploit structure
- analytic attacks on iterated hash functions
  - typically focus on collisions in compression function *f*
  - like block ciphers, HF is often composed of rounds
  - attacks exploit properties of round functions

Some Cryptographic hash functions

- MD5 (digest of 128-bit) -- Rivest
- SHA-1 (160-bit), NIST FIPS 180-1
- National Institute for Standard & Technology (SHA2)

FIPS 180-2 → SHA-256, SHA-384, SHA-512

- SHA3 – new NIST selection (slide will come later)
- RIPE-MD (160-bit) – (Dobbertin, Preneel)
- MASH (based on modular arithmetic) – (Girault)
- Whirlpool (up to 512 bit, based on a dedicated block-cipher), project NESSIE (Barreto, Rijmaen)
- NESSIE - New European Schemes for Signatures, Integrity and Encryption
- HMAC (with embeded hash functions (MD5, SHA-1) and a secret key K), FIPS

SHA

- The **Secure Hash Algorithms** are a family of cryptographic hash functions published by the National Institute of Standards and Technology (NIST) as a U.S. Federal Information Processing Standard (FIPS)
- **SHA-0**: A retronym applied to the original version of the 160-bit hash function published in 1993 under the name "SHA".
- It was withdrawn shortly after publication due to an undisclosed "significant flaw" and replaced by the slightly revised version SHA-1.

SHA family

- **SHA-1**: A 160-bit hash function which resembles the earlier MD5 algorithm.
- This was designed by the National Security Agency (NSA) to be part of the Digital Signature Algorithm.
- Cryptographic weaknesses were discovered in SHA-1, and the standard was no longer approved for most cryptographic uses after 2010.

SHA family (SHA-2 – FIPS 180-4)

- **SHA-2**: A family of two similar hash functions, with different block sizes, known as ***SHA-256*** and *SHA-512*.
- They differ in the word size; SHA-256 uses 32-bit words where SHA-512 uses 64-bit words.
- There are also truncated versions of each standard, known as *SHA-224*, *SHA-384*, *SHA-512/224* and *SHA-512/256*. These were also designed by the NSA.

SHA family

- **SHA-3**: A hash function formerly called ***Keccak***, chosen in 2012 after a public competition among non-NSA designers. It supports the same hash lengths as SHA-2, and its internal structure differs significantly from the rest of the SHA family.

SHA FIPS

- The corresponding standards are
- FIPS PUB 180 (original SHA),
- FIPS PUB 180-1 (SHA-1),
- FIPS PUB 180-2 (SHA-1, SHA-256, SHA-384, and SHA-512).
- NIST has updated Draft FIPS Publication 202, SHA-3 Standard separate from the Secure Hash Standard (SHS).

MD-5 Hash function [Rivest 1992 – RFC 1321]

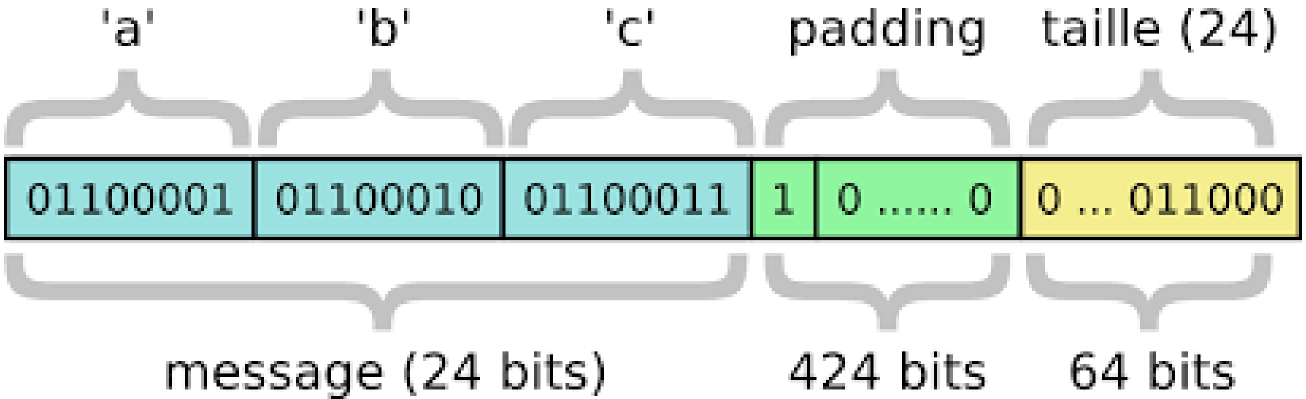
- MD5 Produces hash of length 128 bits

•**Padding**

•The message is "padded" (extended) so that its length (in bits) is being a multiple of 512 bits long.

•Padding is performed as follows: a single "1" bit is appended to the message, and then "0" bits are appended so that the length in bits of the padded message becomes congruent to 448, modulo 512.

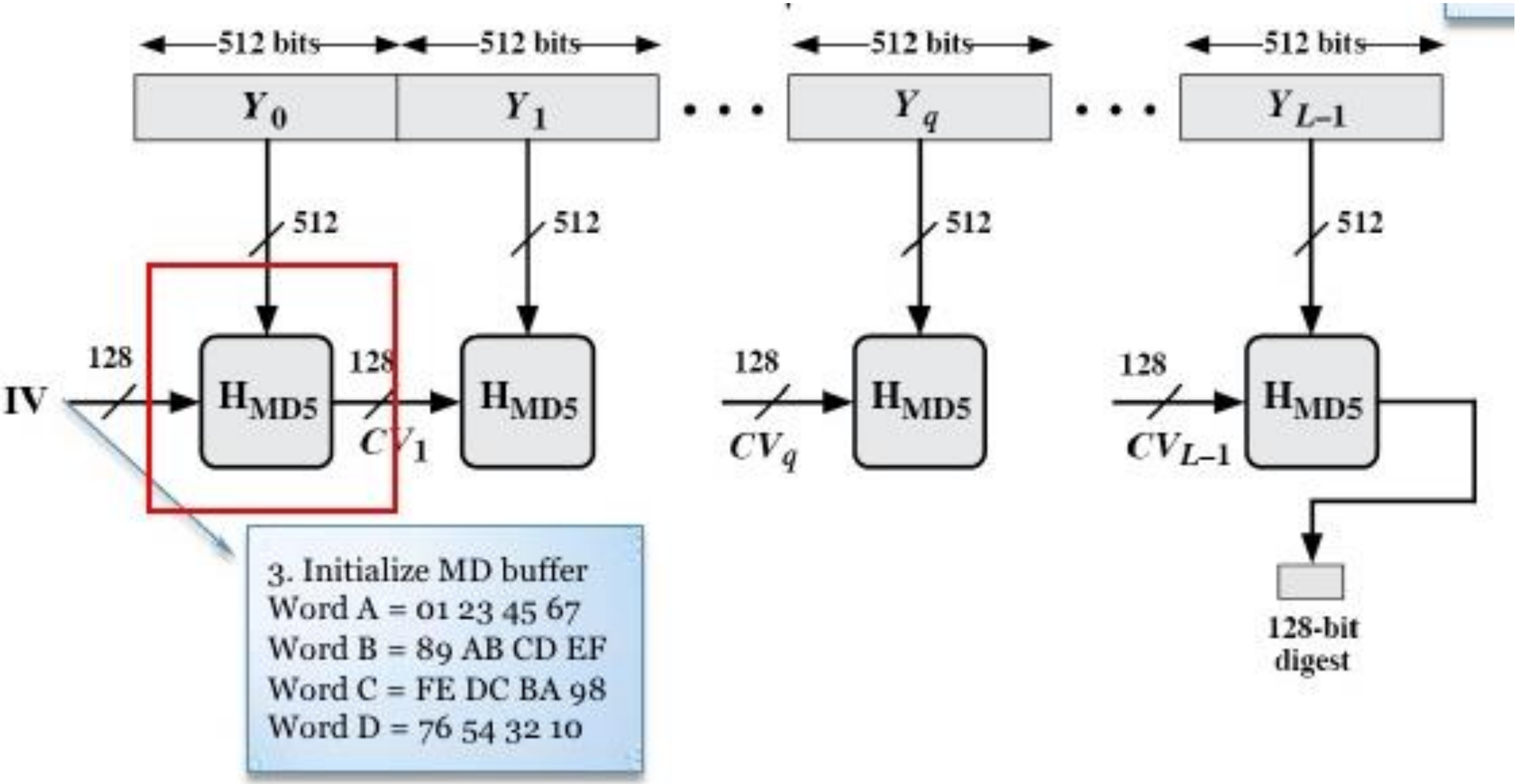
•Last 64 bits are added as binary representation of msg length.



MD-5

- Let message has a length (after padding) that is an exact multiple of 16 (32-bit) words.
- Let M[0 ... N-1] denote the words of the resulting message, where N is a multiple of 16.
- A four-word buffer (A,B,C,D) is used to compute the message digest. Here each of A, B, C, D is a 32-bit register.
- These registers are initialized to the following values in hexadecimal, low-order bytes first):
- word A: 01 23 45 67
- word B: 89 ab cd ef
- word C: fe dc ba 98
- word D: 76 54 32 10

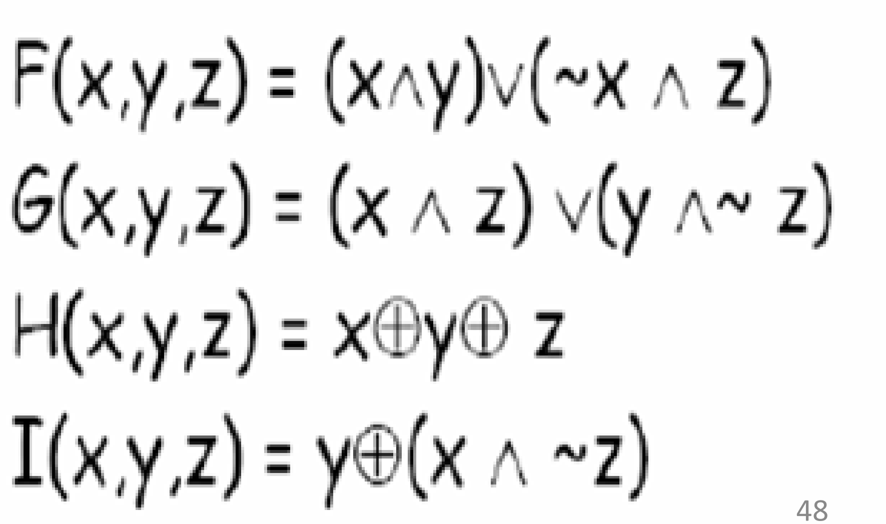
MD5



3. Initialize MD buffer
Word A = 01 23 45 67
Word B = 89 AB CD EF
Word C = FE DC BA 98
Word D = 76 54 32 10

MD5 - **Message Digest Computation**

- **Four auxiliary functions -** Four auxiliary functions are used, each take as input three 32-bit words and produce as output one 32-bit word. //F,G,H,I



$$F(x,y,z) = (x \wedge y) \vee (\sim x \wedge z)$$

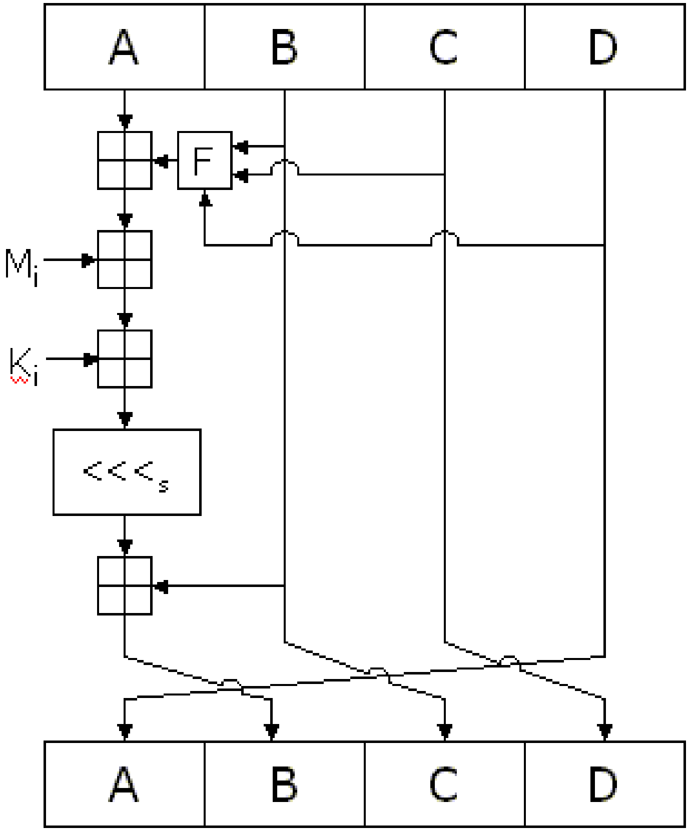
$$G(x,y,z) = (x \wedge z) \vee (y \wedge \sim z)$$

$$H(x,y,z) = x \oplus y \oplus z$$

$$I(x,y,z) = y \oplus (x \wedge \sim z)$$

MD5 – 1 operation

- F – non linear function (F,G,H,I)
- Total such 64 operations
- Grouped in 4 rounds (of 16 each)



MD-5

- Let T[i] denote the i-th element of the table, computed using sine function.

- Do the following: /* Process each 16-word block. */

```
        for i = 0 to N/16 -1 do     /* Copy block i into X. */
        for j = 0 to 15 do
        Set X[j] to M[i*16+j].
         end /* of loop on j */
```

- /* Save A as AA, B as BB, C as CC, and D as DD. */

```
         AA = A    BB = B    CC = C    DD = D
```

/*prog cont.*/

64 values – from sine function

- **for**
- i **from** 0 **to** 63
- k[i] := floor(abs(sin(i + 1)) × (2 **pow** 32))
- **end for**

64 values – from sine function

```
k[ 0.. 3] := { 0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee }
k[ 4.. 7] := { 0xf57c0faf, 0x4787c62a, 0xa8304613, 0xfd469501 }
k[ 8..11] := { 0x698098d8, 0x8b44f7af, 0xffff5bb1, 0x895cd7be }
k[12..15] := { 0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821 }
k[16..19] := { 0xf61e2562, 0xc040b340, 0x265e5a51, 0xe9b6c7aa }
k[20..23] := { 0xd62f105d, 0x02441453, 0xd8a1e681, 0xe7d3fbc8 }
k[24..27] := { 0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed }
k[28..31] := { 0xa9e3e905, 0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a }
k[32..35] := { 0xfffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c }
k[36..39] := { 0xa4beea44, 0x4bdecfa9, 0xf6bb4b60, 0xbebfbc70 }
k[40..43] := { 0x289b7ec6, 0xeaa127fa, 0xd4ef3085, 0x04881d05 }
k[44..47] := { 0xd9d4d039, 0xe6db99e5, 0x1fa27cf8, 0xc4ac5665 }
k[48..51] := { 0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039 }
k[52..55] := { 0x655b59c3, 0x8f0ccc92, 0xffeff47d, 0x85845dd1 }
k[56..59] := { 0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0x4e0811a1 }
k[60..63] := { 0xf7537e82, 0xbd3af235, 0x2ad7d2bb, 0xeb86d391 }
```

MD-5 – round 1 – F function

- /* Round 1. */ /* Let [abcd k s i] denote the operation
  a = b + ((a + F(b,c,d) + X[k] + T[i]) <<< s). */

- /* Do the following 16 operations. */
  [ABCD 0 7 1] [DABC 1 12 2] [CDAB 2 17 3]
  [BCDA 3 22 4] [ABCD 4 7 5] [DABC 5 12 6]
  [CDAB 6 17 7] [BCDA 7 22 8] [ABCD 8 7 9]
  [DABC 9 12 10] [CDAB 10 17 11] [BCDA 11 22 12] [ABCD 12 7 13]
  [DABC 13 12 14] [CDAB 14 17 15] [BCDA 15 22 16]

A B C D are updated……

Round 2 – G function

- /* Round 2. */
- /* Let [abcd k s i] denote the operation
- **a = b + ((a + G(b,c,d) + X[k] + T[i]) <<< s).** */

- /* Do the following 16 operations. */

[ABCD 1 5 17] [DABC 6 9 18] [CDAB 11 14 19] [BCDA 0 20 20] [ABCD 5 5 21] [DABC 10 9 22] [CDAB 15 14 23] [BCDA 4 20 24] [ABCD 9 5 25] [DABC 14 9 26] [CDAB 3 14 27] [BCDA 8 20 28] [ABCD 13 5 29] [DABC 2 9 30] [CDAB 7 14 31] [BCDA 12 20 32]

Round 3 – H function

- /* Round 3. */
- /* Let [abcd k s t] denote the operation

a = b + ((a + H(b,c,d) + X[k] + T[i]) <<< s). */

- /* Do the following 16 operations. */

[ABCD 5 4 33] [DABC 8 11 34] [CDAB 11 16 35] [BCDA 14 23 36] [ABCD 1 4 37] [DABC 4 11 38] [CDAB 7 16 39] [BCDA 10 23 40] [ABCD 13 4 41] [DABC 0 11 42] [CDAB 3 16 43] [BCDA 6 23 44] [ABCD 9 4 45] [DABC 12 11 46] [CDAB 15 16 47] [BCDA 2 23 48]

Round 4 – I function

- /* Round 4. */
- /* Let [abcd k s t] denote the operation

  a = b + ((a + I(b,c,d) + X[k] + T[i]) <<< s). */
- /* Do the following 16 operations. */

  [ABCD 0 6 49] [DABC 7 10 50] [CDAB 14 15 51] [BCDA 5 21 52] [ABCD 12 6 53] [DABC 3 10 54] [CDAB 10 15 55] [BCDA 1 21 56] [ABCD 8 6 57] [DABC 15 10 58] [CDAB 6 15 59] [BCDA 13 21 60] [ABCD 4 6 61] [DABC 11 10 62] [CDAB 2 15 63] [BCDA 9 21 64]

MD-5 - result

- /* Then perform the additions – update with the original vlaues*/
- A = A + AA B = B + BB C = C + CC D = D + DD
- /* end of loop on i */
- Output = ABCD /*concatenated – 128 bit*/

Secure Hash Algorithm (SHA)

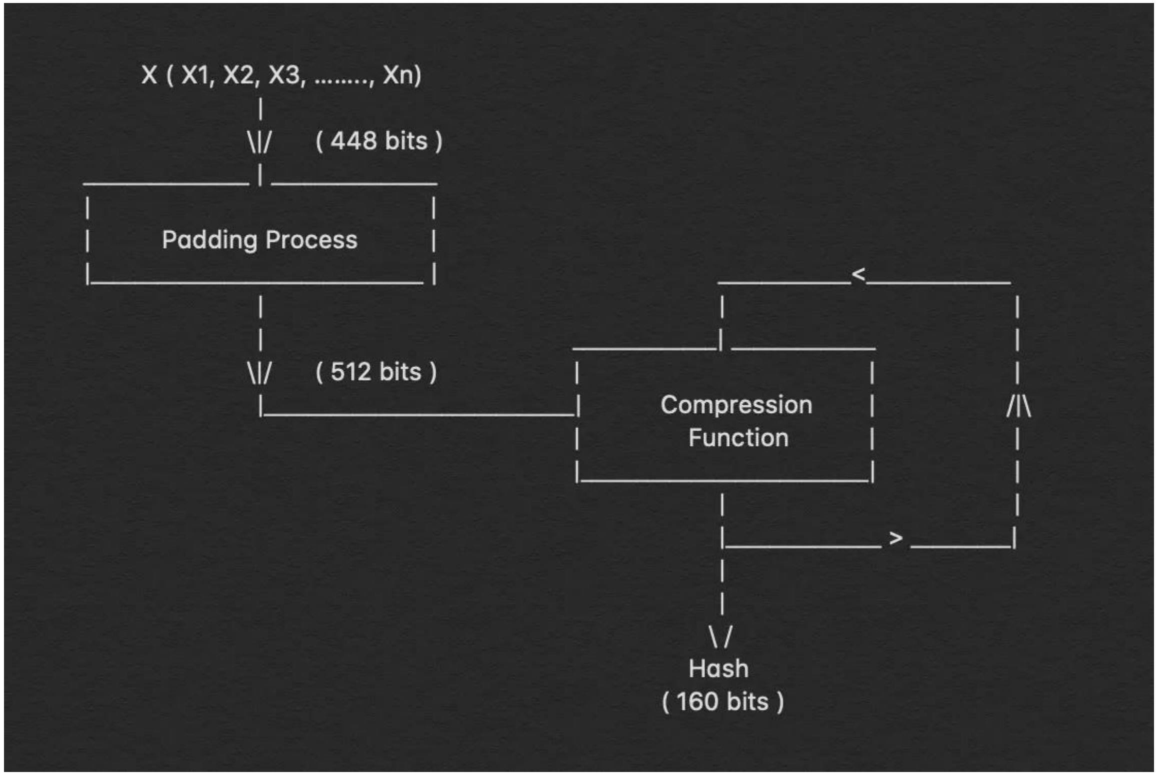
- SHA-1 or Secure Hash Algorithm 1 is a cryptographic hash function which takes an input and produces a 160-bit (20-byte) hash value.
- SHA originally designed by NIST (National Institute of standards and technology) and published as a Federal Information Processing Standard (FIPS 180) in 1993.
- revised in 1995 as FIPS 180-1 and referred to as SHA-1, also Internet RFC3174
- The algorithm is SHA, the standard is SHS – secure hash standard

SHA-1 (Secure Hash Algorithm)

- an iterated hash function with a 160-bit message digest. (5 registers)
- Padding same as MD-5
- SHA-1 is built from word-oriented operations on bitstrings, where a word consists of 32 bits (or eight hexadecimal characters (nibble)).
- The operations used in SHA-1 are as follows:
- X ٨ Y bitwise “and” of X and Y
- X ٧ Y bitwise “or” of X and Y
- X xor Y bitwise “xor” of X and Y
- ¬X bitwise complement of X
- X+Y integer addition modulo 2^{32}
- ROTLs(X) circular left shift of X by s position ($0 \leq s \leq 31$)

Padding and Blocks

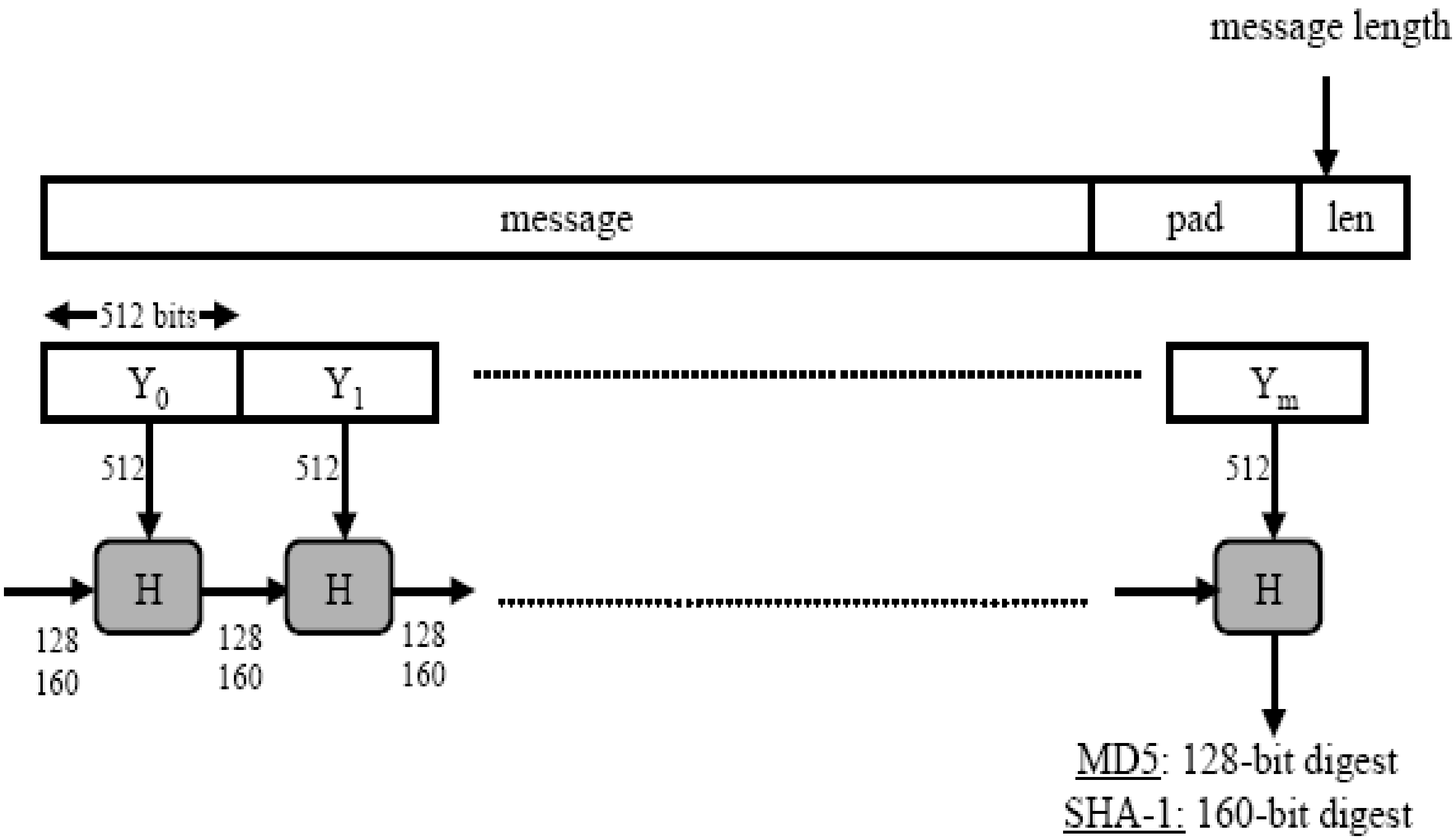
- Same as MD5
- y = M1 || M2 || …. || Mn.



Pre-processing

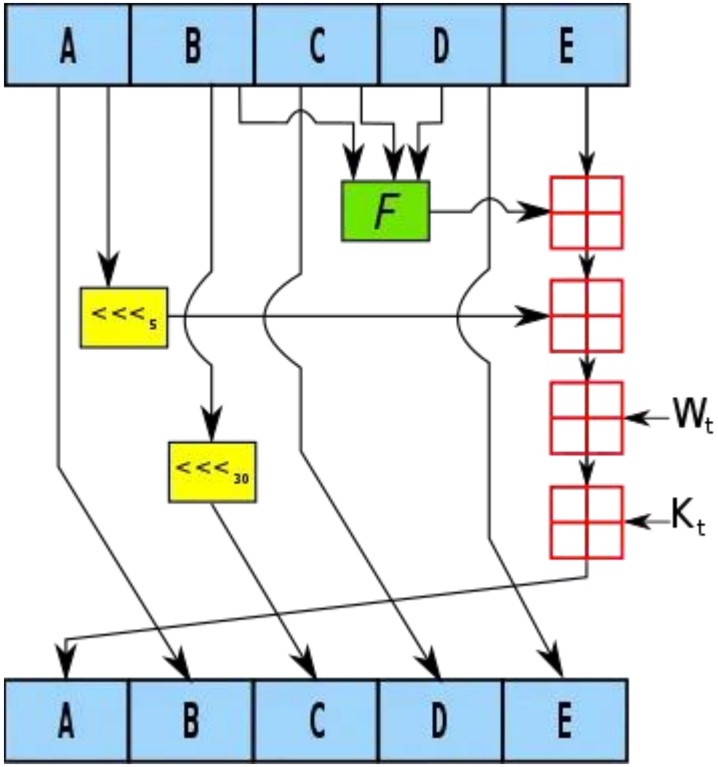
- append the bit 1 to the message
- append $0 \le k < 512$ bits 0, so that the resulting message length (in *bits*) is congruent to $448 \equiv -64 (\mathrm{mod} 512)$
- append length of message (before pre-processing), in bits, as 64-bit big-endian integer
- This pre-processing is a kind of padding, which makes sure that the input size is a multiple of 512 bits

MD-5 and SHA-1


message length
message
pad
len
512 bits
Y_0
Y_1
Y_m
512
512
512
H
H
H
128
160
128
160
128
160
MD5: 128-bit digest
SHA-1: 160-bit digest

Initial Values

- A = 0x67452301
  B = 0xEFCDAB89
  C = 0x98BADCFE
  D = 0x10325476
  E = 0xC3D2E1F0



SHA-1 functions

- Define the function f0,…f79 as follows :

f_t (B,C,D) =

- $(B \wedge C) \vee ((\neg B) \wedge D)$ if $0 \leq t \leq 19$
- B xor C xor D if $20 \leq t \leq 39$
- $(B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$ if $40 \leq t \leq 59$
- B xor C xor D if $60 \leq t \leq 79$.
- Each function f_t takes three words B, C and D as input, and produces one word as output.

Functions - F and K (constant)

- First 20 Rounds
  f(1) = (B **and** C) **or** ((**not** B) **and** D)
  k(1) = 0x5A827999
- Next 20 Rounds
  f(2) = B **xor** C **xor** D
  k(2) = 0x6ED9EBA1
- Next 20 Rounds
  f(3) = (B **and** C) **or** (B **and** D) **or** (C **and**)
  k(3) = 0x8F1BBCDC
- Next 20 Rounds
  f(4) = B **xor** C **xor** D
  k(4) = 0xCA62C1D6

SHA-1

- Four constants are used :
  - K_t = 0x5a827999, for t = 0 to 19
  - K_t = 0x6ed9eba1, for t = 20 to 39
  - K_t = 0x8f1bbcdc, for t = 40 to 59
  - K_t = 0xca62c1d6, for t = 60 to 79

- Message block is transferred from 16 blocks to 80 blocks:
  - $W_t = M_t$, for t=0 to 15
  - $W_t = (W_{t-3} + W_{t-8} + W_{t-14} + W_{t-16})<<<1$, for t=16 to 79

SHA-1

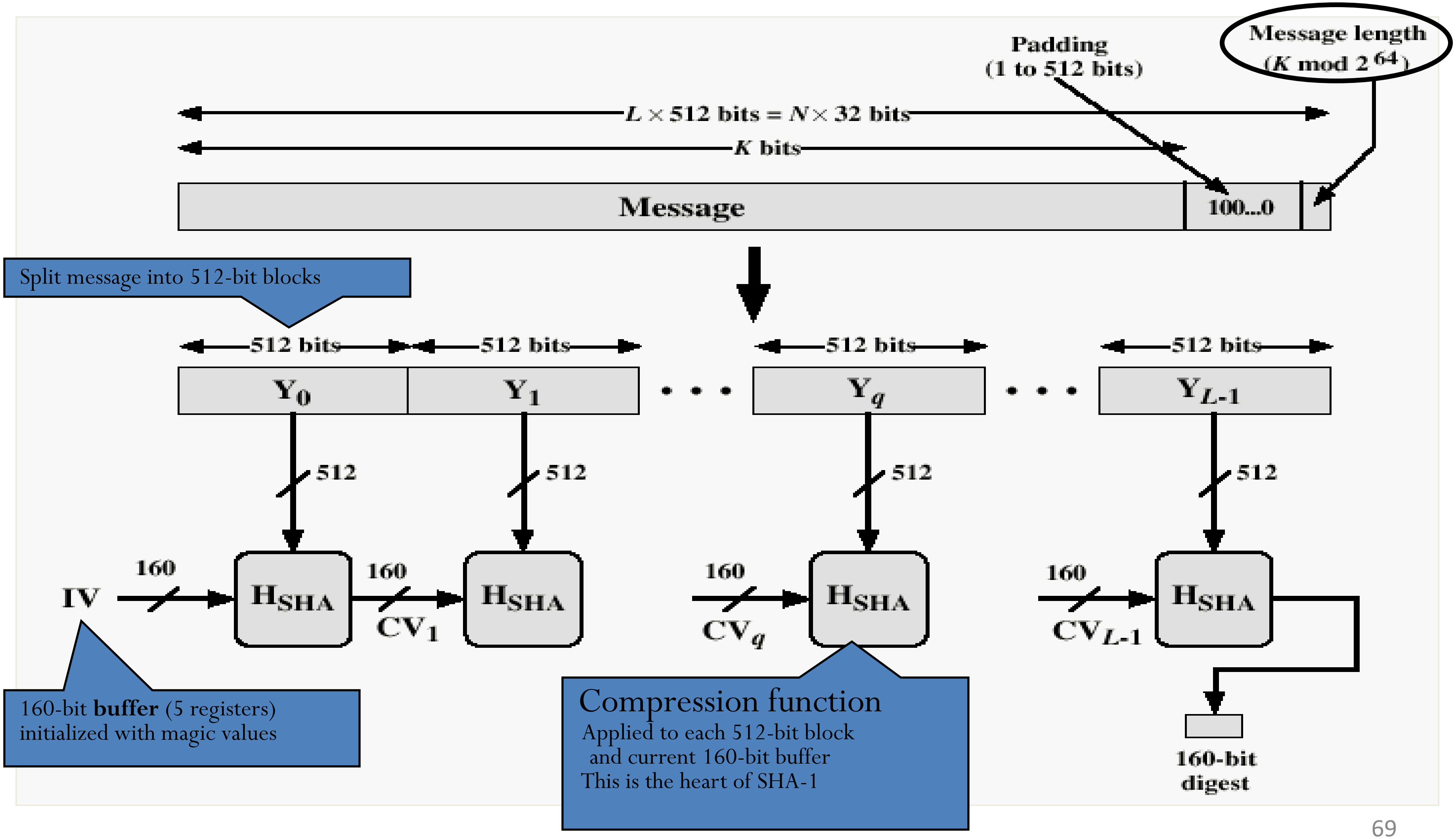
- H0 ← 67452301
- H1 ← EFCDAB89
- H2 ← 98BADCFE
- H3 ← 10325476
- H4 ← C3D2E1F0

- Define the word constants K0,…,K79, which are used in the computation of SHA-1(x), as follows:
- Kt =
- 5A827999 if $0 \leq t \leq 19$
- 6ED9EBA1 if $20 \leq t \leq 39$
- 8F1BBCDC if $40 \leq t \leq 59$
- CA62C1D6 if $60 \leq t \leq 79$

SHA-1

```
for  i ← 1 to n
do
        denote Mi =  W0 || W1||…..||W15, where each Wi is a word
        for t ← 16 to 79
        do Wt ←ROTL1 (Wt-3     Wt-8      Wt-14      Wt-16)
A ← H0
B ← H1
C ← H2
D ← H3
E ← H4
```

Description of SHA-1



Split message into 512-bit blocks

160-bit **buffer** (5 registers)
initialized with magic values

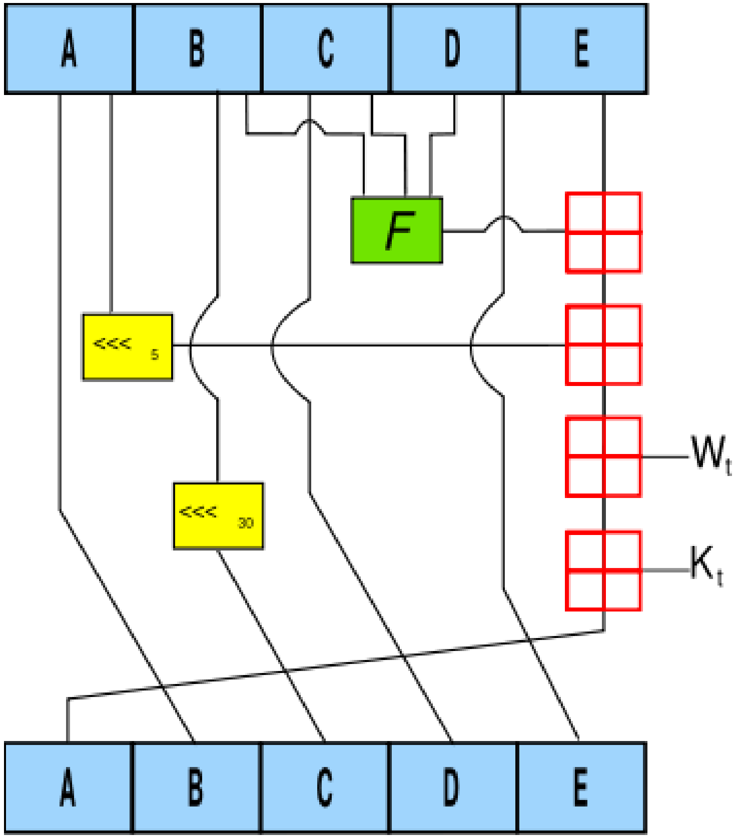
Compression function

Applied to each 512-bit block
and current 160-bit buffer
This is the heart of SHA-1

SHA-1

```
For  t ← 0 to 79
do
        Temp ← ROTL5 (A) + ft (B, C, D) + E +Wt + Kt
        E ← D
        D ← C
        C ← ROTL30 (B)
        B ← A
        A ← temp
H0 ← H0 + A
H1 ← H1 + B
H2 ← H2 + C
H3 ← H3 + D
H4 ← H4 + E
return ( H0 || H1 ||  H2 || H3 || H4)
```

One SHA-1 operation



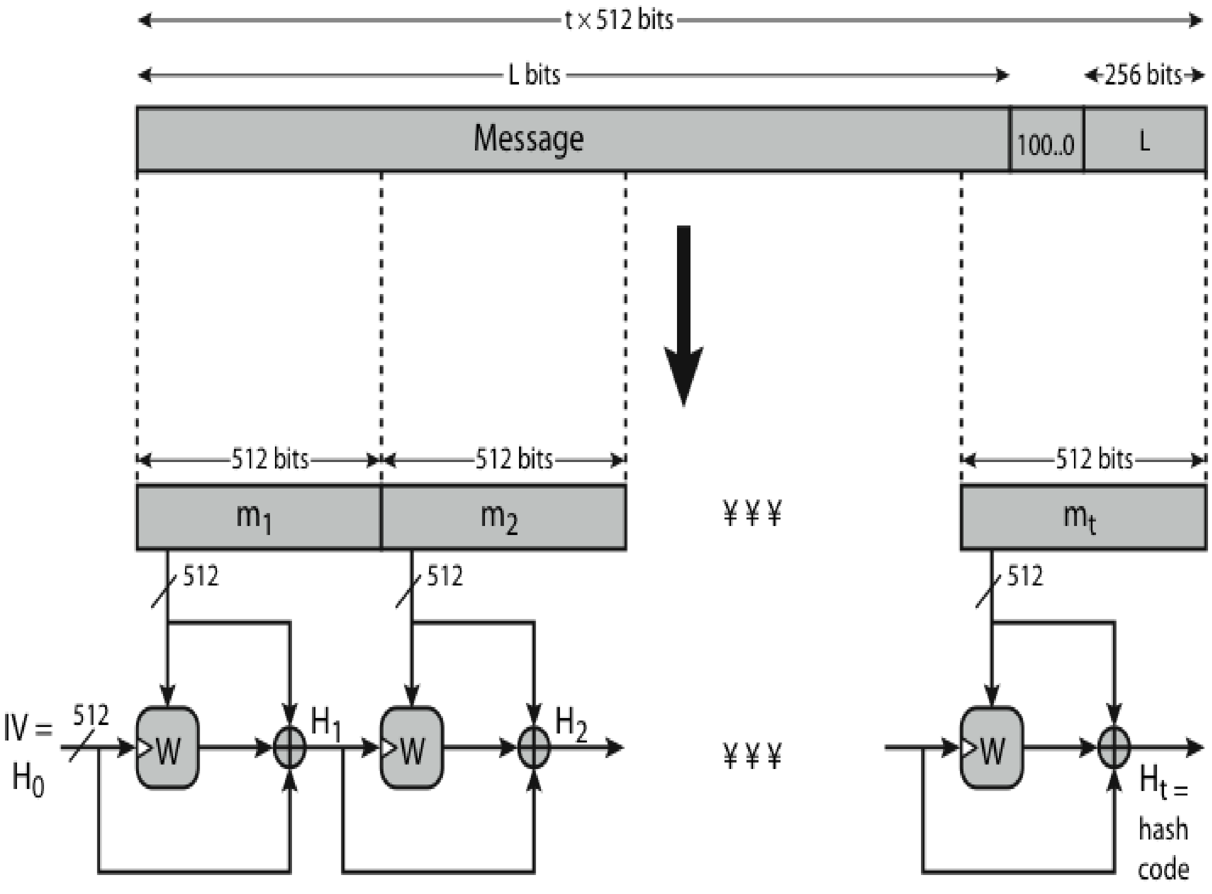
If *t* is the operation number (from 0 to 79), W_t represents the *t* th sub-block of the expanded message, and <<< *s* represents a left circular shift of *s* bits, then the main loop looks like:

```
For t=0 to 79
    TEMP = (a <<< 5) + $f_t$(b,c,d) + e + $W_t$ + $K_t$
    e = d
    d=c
    c=b <<< 30
    b = a
    a = TEMP
```

SHA-1

- SHA-1 is built around an internal "compression function" which takes as input the 160-bit state and a 512-bit message block, and returns a new state.

Whirlpool – 512 bit HF – using AES internals - endorsed by European NESSIE project – updating 512 bit buffer



Note: triangular hatch marks key input

SHA-3 <published in 2015>

- NIST has initiated an effort to develop one or more additional hash algorithms through a public competition – 2007
- to develop a new cryptographic hash algorithm, which converts a variable length message into a short "message digest" that can be used in generating digital signatures, message authentication codes, and many other security applications in the information infrastructure
- First round submissions (64 entries) and conf – Dec 2008 – 51 candidates
- Second round – 14 candidates – July 2009
- Third round – 5 finalists – Dec 2012 – BLAKE, Grøstl, JH, Keccak , Skein

SHA3

- Performance, Security, CryptAnalysis, Diversity
- Winner – Keccak (Sponge functions)!!!

SHA-3

- a tunable security parameter, such as the number of rounds, which would allow the selection of a range of possible security/performance tradeoffs
- a recommended value for each digest size
- is SHA-3 the same as sha256?
- SHA-3 (and its variants SHA3-224, SHA3-256, SHA3-384, SHA3-512), is considered more secure than SHA-2 (SHA-224, SHA-256, SHA-384, SHA-512) for the same hash length. For example, SHA3-256 provides more cryptographic strength than SHA-256 for the same hash length (256 bits).

SHA-2 256 aka SHA-256

- Message Digest Length = 256
- Initial hash value:
- H[0] = 6A09E667
- H[1] = BB67AE85
- H[2] = 3C6EF372
- H[3] = A54FF53A
- H[4] = 510E527F
- H[5] = 9B05688C
- H[6] = 1F83D9AB
- H[7] = 5BE0CD19

SHA-256 Application in Bitcoin Blockchain

- Bitcoin Proof of Work – Difficulty (compute a hash with certain leading bits as zeros)
- Solving a hash involves computing a proof-of-work, called a NONCE, or “number used once”, that, when added to the block, causes the block’s hash to begin with a certain number of zeroes.
- Once a valid proof-of-work is discovered, the block is considered valid and can be added to the blockchain.

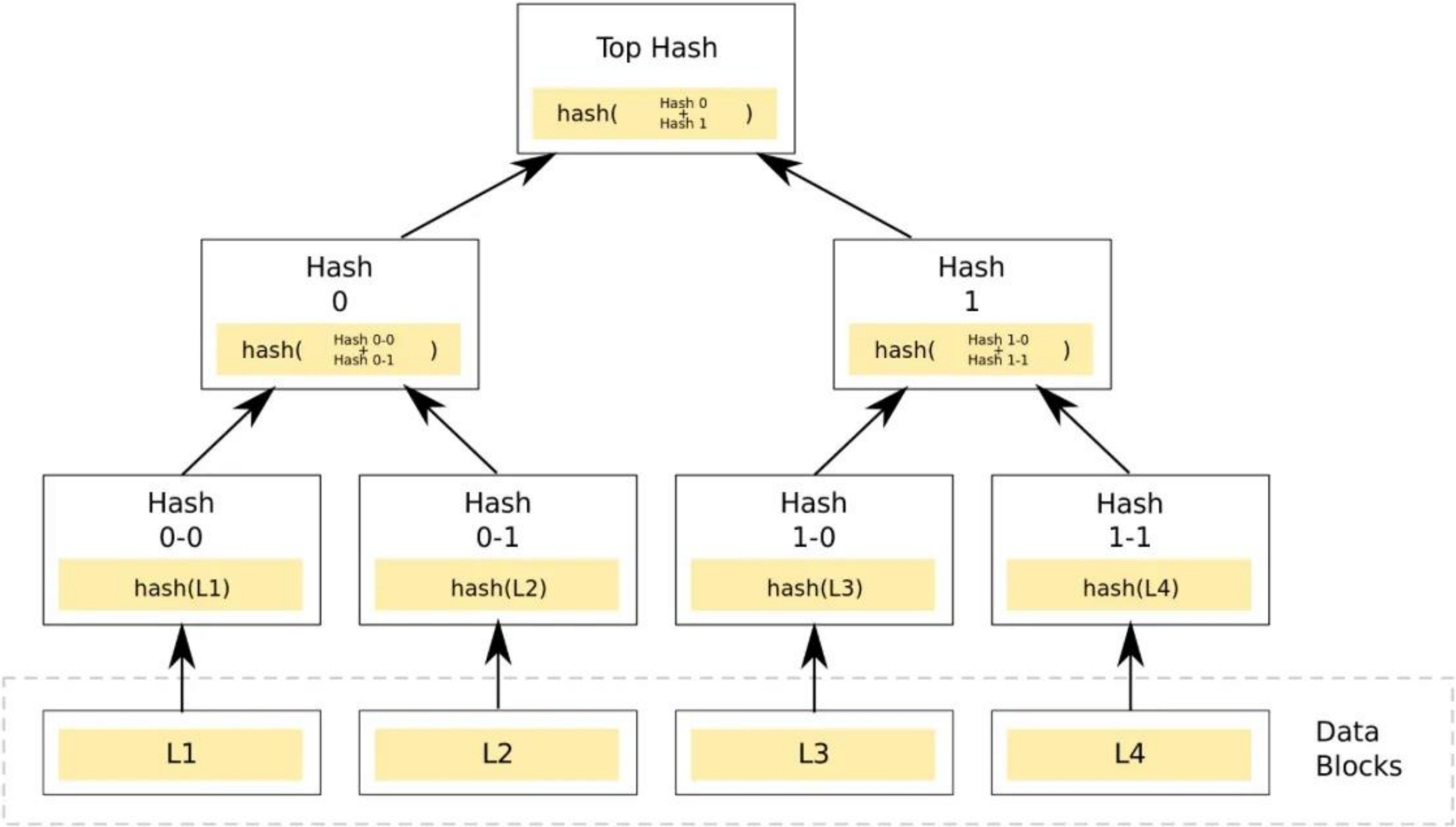
Bitcoin (uses SHA256)

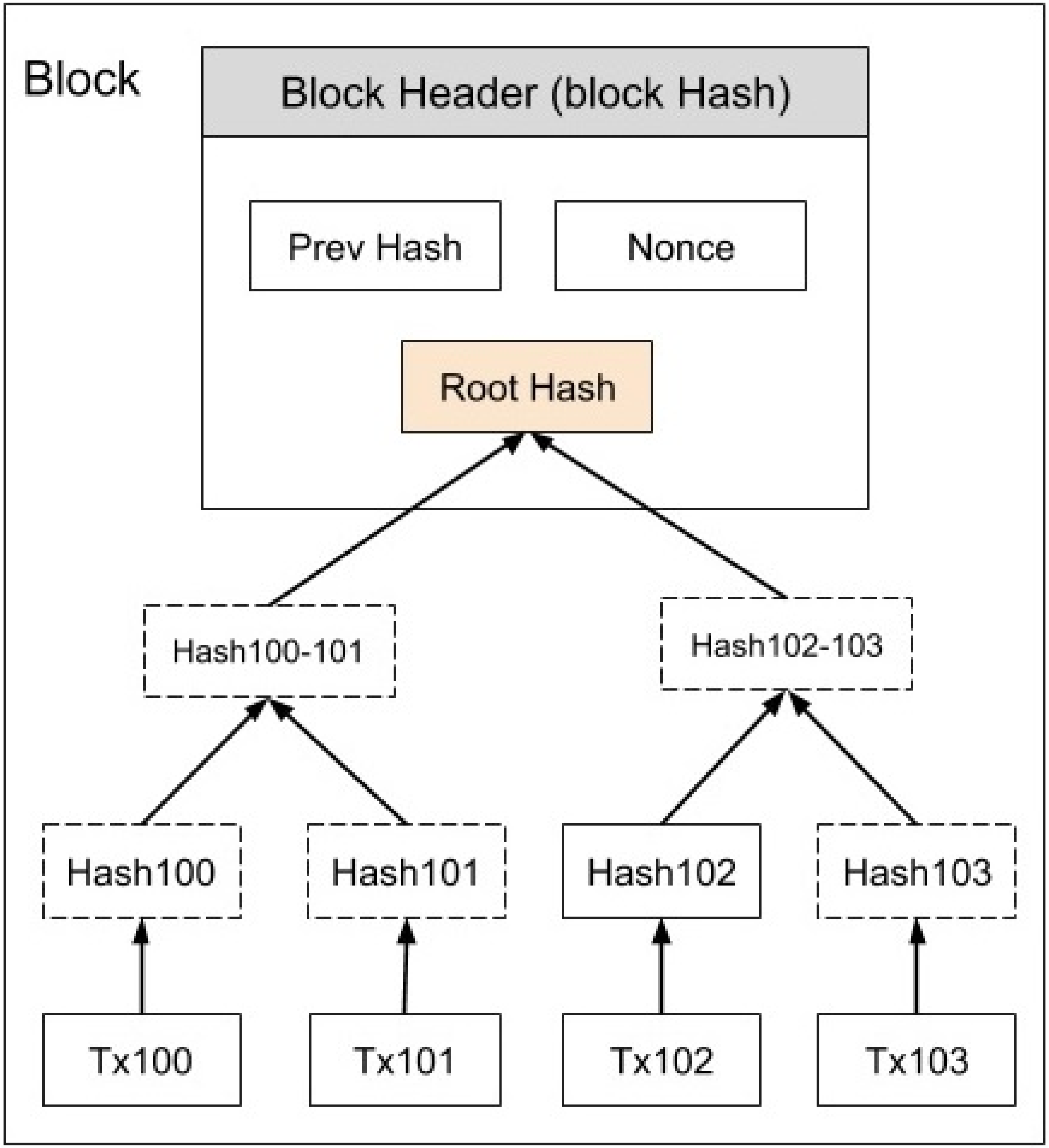
- Current difficulty level is 76 zeros of 256 (39,156,400,059,293) – Feb 22, 2023
- 1728 transactions
- Inputs 5,320, Outputs 6,085
- Fees: 0.11333773 BTC
- Nonce: 1,373,504,660 (0x51de0494)
- E.g. **Bitcoin Block777,735** Mined on February 22, 2023 07:22:30 – Hash after solving a difficulty is
- 000000000000000000065108c78d576253c2501426c50 17f442533a25a0673f5
- Merkle Root is bdc244d801fddb55d2dde668ffe14f3d5ff1e7b549a38aa 6965a9cc67c800ad4

Merkel Tree and Merkel Root

- **Merkle Tree –** it's a binary hash tree; a data structure used for efficiently summarising and verifying the integrity of large data sets
- a kind of inverted tree structure with the root (known as the **Merkle root**) at the top and the leaves at the bottom
- This data structure is used in the bitcoin and blockchain technologies to summarise all the transactions in a block, providing a very efficient process to verify whether a transaction is included in a block

Merkel Tree


Top Hash
hash(Hash 0 + Hash 1)
Hash 0
hash(Hash 0-0 + Hash 0-1)
Hash 1
hash(Hash 1-0 + Hash 1-1)
Hash 0-0
hash(L1)
Hash 0-1
hash(L2)
Hash 1-0
hash(L3)
Hash 1-1
hash(L4)
L1
L2
L3
L4
Data Blocks



Transactions Hashed in a Merkle Tree

Merkel Root

- a binary tree of hash lists where the parent node is the hash of its children, and the leaf nodes are hashes of the original data blocks.
- The leaf nodes have to be even in number for a binary hash tree to work so if the number of leaf nodes is an odd number, then the last leaf node is duplicated to even the count
- Each pair of leaf nodes is concatenated and hashed to form the second row of hashes.
- The process is repeated until a row is obtained with only two hashes
- These last two hashes are concatenated to form the Merkle root

membership of the transaction

- key use case of the Merkle tree is in **simple payment verification** where you can prove that a transaction exists in a block, by showing the Merkle branch of the transaction
- For the Merkle branch, we just need the transaction's position in the block's transaction list and the hashing partners at each level, instead of the complete set of transactions.
- by going up the tree and combining our result with the respective hashing partner at each level, we finally get the Merkle root. Thus we can prove membership of the transaction in the block.

HF design and security issues

- Known Answer Tests (KATs) and Monte Carlo Tests (MCTs)
- Known Answer Test values must be provided with submissions, which demonstrate operation of the SHA-3
- candidate algorithm with varying length inputs, for each of the minimum required hash length values (224, 256, 384, and 512-bits).
- There are three types of KATs that are required for all submissions: 1) Short Message Test, 2) Long Message Test, and 3) Extremely Long Message Test (2^32 bits).

MCT

- The Monte Carlo Test provides a way to stress the internal components of a candidate algorithm.
- A seed message will be provided. This seed is used by a pseudorandom function together with the candidate algorithm to generate 100,000 message digests.
- 100 of these 100,000 message digests, i.e. once every 1,000 hashes, are recorded as checkpoints to the operation of the candidate algorithm

HF Security issues

- collision-finding, first-preimage-finding, second-preimage-finding, length-extension attack, multicollision attack

- How these hash codes satisfy essential properties of a hash function?
- 1. All outputs are equally probable.
- 2. HF should provide security with some computational complexity proof.
- B. Changing a single bit in input text will affect many output bits, therefore manipulating the effect of modification by complementing other bits in input is not easy.
- Compression function f should be fairly secure.

Security tests - compression function

- 1-bit sensitivity test – change only a single bit in an input block (one after another and look for output collisions)
- 2-bit sensitivity test (change two bits – balancing)
- Exhaustive collision detection test (take input message of 1 block, no padding, compute hash, modify input block exhaustively and compute hash, look for collisions)
- at input, any regular pattern)

Tests

- Statistical tests (correlations between input-output) any hash bit is
- → influenced by no. of 0’s or no. of 1’s in the input?
- → influenced by any particular group of input bits?
- Avalanche effect (changing a 1-bit at input changes how many bits in output?)
- Security **- input discovery**, collisions, Any functional weakness – specific input patterns (for all 0’s, all 1’s)

Summary

- Hash function generates a small identifier of a large object (number, message, file, document, etc.)
- Speed of computation – fast (simple Mathematical functions used)
- Computational Complexity increases linearly with digest size and input length (no. of blocks processed)
- Memory requirement (for storing temporary values)
- Published material (IV, Padding technique (bit sequence, byte count))
- Scalability (by increasing block length)
- Security issues more rounds of computation remove predictable output behavior for specific input patterns