WEEK 3

Hashing

Introduction: The Problems of Integrity

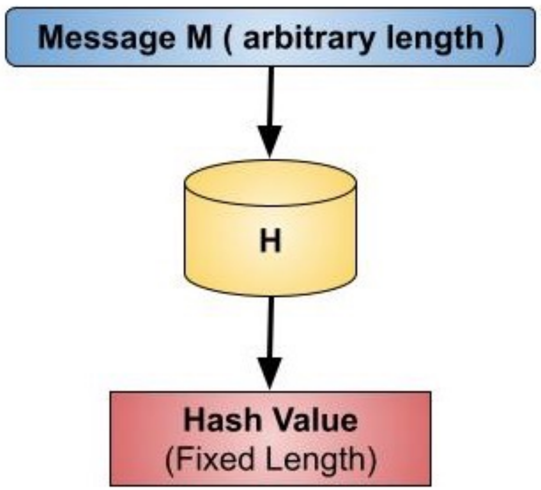
Question #1:

Imagine you're downloading a critical security patch for a server. It's **2GB**.

How do you know for certain that the file you received is the exact one the developer released? **How do you know** it wasn't corrupted during download or, worse, maliciously **modified** by an attacker?

Digital Fingerprint: Hashing as the solution

- A hash function takes any data (text, a file, a picture) and produces a unique, fixed-size string of text called a hash, digest, or checksum.
- h = H(M)   M=message, H = hash function, h= hash value
- hash(”I’m a student in cybersecurity") ->
  d7a8fbb307………2d02d0bf37c9e592
- hash(" I’m a student in cyber security") ->
  e4c4d8f3bf ……… 6367f1f9045bb976



Why We Depend on Hashing: Core Use Cases

- Storing Passwords Securely
- Hash Tables Fast look up of the data
- Build Caching
- Proof of work
  - Cryptocurrencies, or bitcoin,or blockchain
- Verifying Integrity
  - File Downloads
  - Digital Forensics/malware
  - Protect the data

Hash

Simple hash							Password hash				
md5	SHA1	SHA224	SHA226	SHA512	SHA3	etc...	bcrypt	Argon2	scrypt	PBKDF2	etc..

Hash Function: Rule

1. Same input must be same output and Same length
2. No Hash Collision
3. Irreversible output
    1. Hash(“Hello”) = 45as554as4a5s4a65s4a65s6as6
    2. Hash(“aaaaaaaaa”)=8a7s6da56as89as76a8s7a61
    3. Hash(“hello”)=9a95a9s5a9s9n4ggfg4123

Hash Function: Properties of Secure

1. **Pre-Image Resistance:** given a hash value h, it should be different to find any message m such that h = hash(m).
2. **2nd pre-image resistance:** Given an input m_1, it should be difficult to find a different input m_2 such that $hash(m_1) = hash(m_2)$
3. **Collision Resistance:** Hard to find two different inputs with the same hash

Collisions have been found it's broken (END)

Hash Function: Properties of Secure

- Hash Functions and Attack Techniques
  - **Collision Attacks:** Aim to find two different inputs that produce the same hash value. Secure hash functions are designed to make finding collisions difficult.
  - **Preimage Attacks:** Aim to find an input that matches a given hash value.
  - **Second Preimage Attacks:** Aim to find a different input that produces the same hash value as a given input.
  - **Rainbow Table Attacks:** Use precomputed hash values to crack passwords.

Hash Function: Ranbow Table

A rainbow table is a precomputed database of passwords and their hash values.

Attackers use it to quickly reverse a hash into the original password without guessing each time.

No.	Password	MD5 Hash
1	123456	e10adc3949ba59abbe56e057f20f883e
2	password	5f4dcc3b5aa765d61d8327deb882cf99
3	12345678	25d55ad283aa400af464c76d713c07ad
4	qwerty	d8578edf8458ce06fbc5bb76a58c5ca4
5	abc123	e99a18c428cb38d5f260853678922e03
6	letmein	0d107d09f5bbe40cade3de5c71e9e9b7
7	monkey	d0763edaa9d9bd2a9516280e9044d885
8	1234	81dc9bdb52d04dc20036dbd8313ed055
9	football	37b4e2d82900d5e94b8da524fbeb33c0
10	iloveyou	23d5d5f8d54d50e555b9e90627a9f64d
11	admin	21232f297a57a5a743894a0e4a801fc3
12	welcome	5f4dcc3b5aa765d61d8327deb882cf99 *(same as "password")*
13	111111	96e79218965eb72c92a549dd5a330112
14	sunshine	5d41402abc4b2a76b9719d911017c592
15	princess	e38ad214943daad1d64c102faec29de4

Hash Function: Hash Collisions

- **Birthday Attacks(Birthday Paradox)** A hash collision happens when two different inputs produce the same hash value just like two people sharing a birthday.

- a =:"TEXTCOLLBYfGiJUETHQ4hAcKSMd5zYpgqf1YRDhkmxHkhPWptrkoyz28wnI9V0aHeAuaKnak"
- b =:"TEXTCOLLBYfGiJUETHQ4hEcKSMd5zYpgqf1YRDhkmxHkhPWptrkoyz28wnI9V0aHeAuaKnak"

- MD5 : faad49866e9498fc1719f5289e7a0269

Hash Function: MD5

- **MD5 Algorithm Description**

  We begin by supposing that we have a b-bit message as input, and that we wish to find its message digest. Here b is an arbitrary nonnegative integer; b may be zero, it need not be a multiple of eight, and it may be arbitrarily large. We imagine the bits of the message written down as follows:

  m_0 m_1 ... m_{b-1}

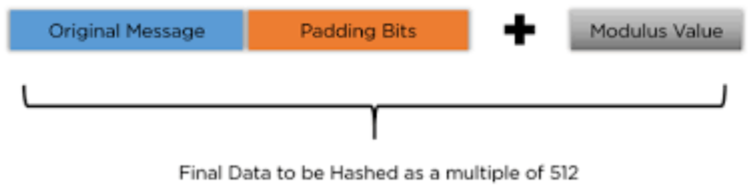
  The following five steps are performed to compute the message digest of the message.

Hash Function: MD5

- **Step1 : Append Padding Bits**

  The message is "padded" (extended) so that its length (in bits) is congruent to 448, modulo 512. That is, the message is extended so that it is just 64 bits shy of being a multiple of 512 bits long. Padding is always performed, even if the length of the message is already congruent to 448, modulo 512. 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.

  In all, at least one bit and at most 512 bits are appended.



Hash Function: MD5

- **Step1 : Append Padding Bits**
  - Append a single 1 bit.
  - Append 0 bits until length ≡ 448 (mod 512).
  - Adds at least 1 bit, at most 512 bits.

String **abc** to **61 62 63** and below

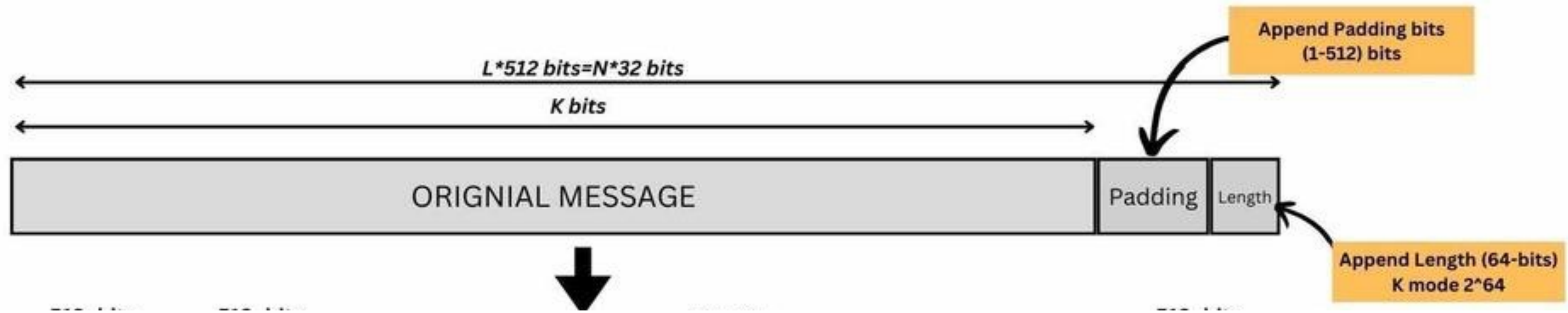
- i.e 01100001 01100001 01100001 1000000 0000000 00000000 000000000 00000000 0000000 000000000000000000000000000000000 000 until 448 bits

Hash Function: MD5

- **Step2 : Append Length**

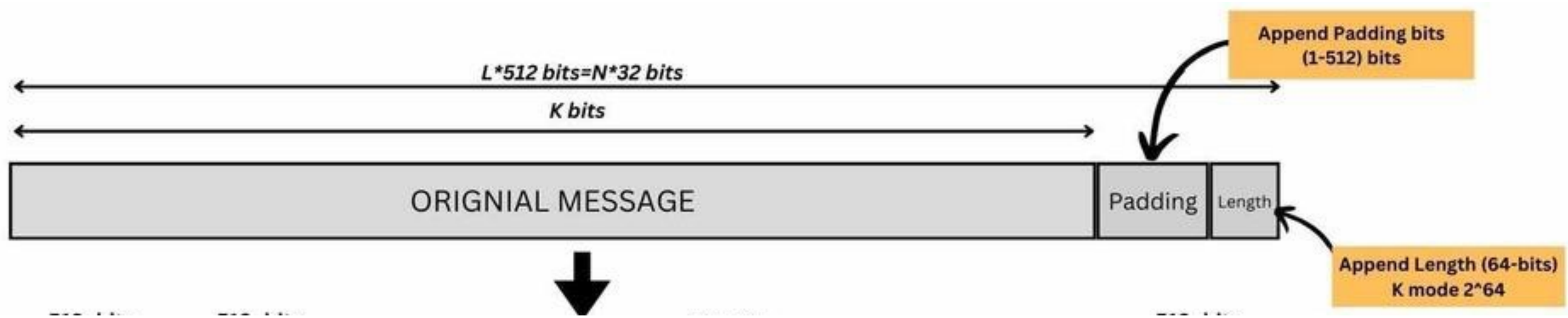
  A 64-bit representation of b (the length of the message before the padding bits were added) is appended to the result of the previous step. In the unlikely event that b is greater than 2^64, then only the low-order 64 bits of b are used. (These bits are appended as two 32-bit words and appended low-order word first in accordance with the previous conventions.)

  At this point the resulting message (after padding with bits and with b) has a length that is an exact multiple of 512 bits. Equivalently, this message has a length 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.



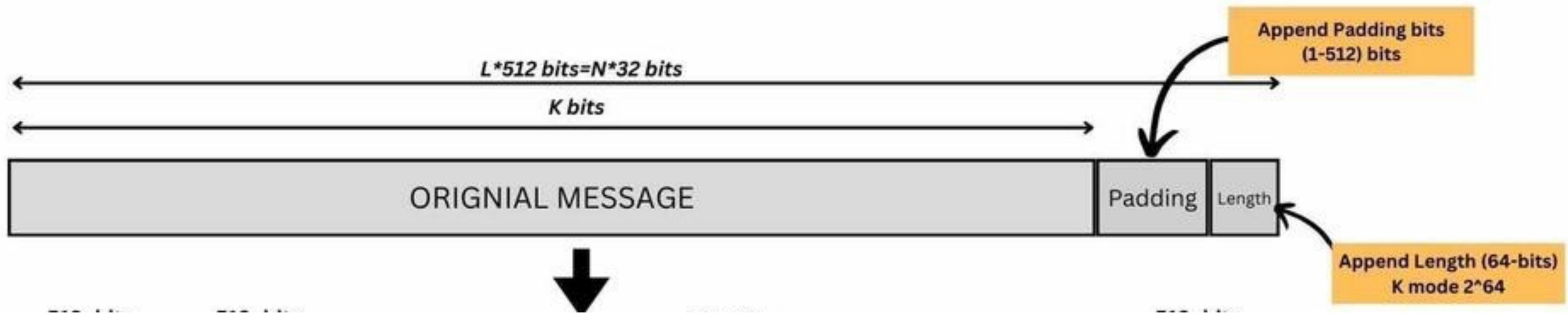
Hash Function: MD5

- **Step1: Padding**
  - i.e 01100001 01100001 01100001 1000000 0000000 00000000 000000000 00000000 0000000 0000000000000000000000000000000000 until 448 bits
- **Step2 : Append Length (64bit)**
  - 00011000 00000000 00000000 00000000 00000000 00000000 00000000 00000000



Hash Function: MD5

- **Step1 + Step2 (512bits)**
  - i.e 01100001 01100001 01100001 1000000 0000000 00000000 000000000 00000000 0000000 00000000000000000000000000000000000 until 448 bits + 64bits 00011000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
  - abc(24 bits) + padding + length 24(24 mod 2^64) = 512bits



Hash Function: MD5

- **Step3 : Initialize MD Buffer**

  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

Hash Function: MD5

- **Step4 : Process Message in 16-Word Blocks**

  We first define four auxiliary functions that each take as input three 32-bit words and produce as output one 32-bit word.

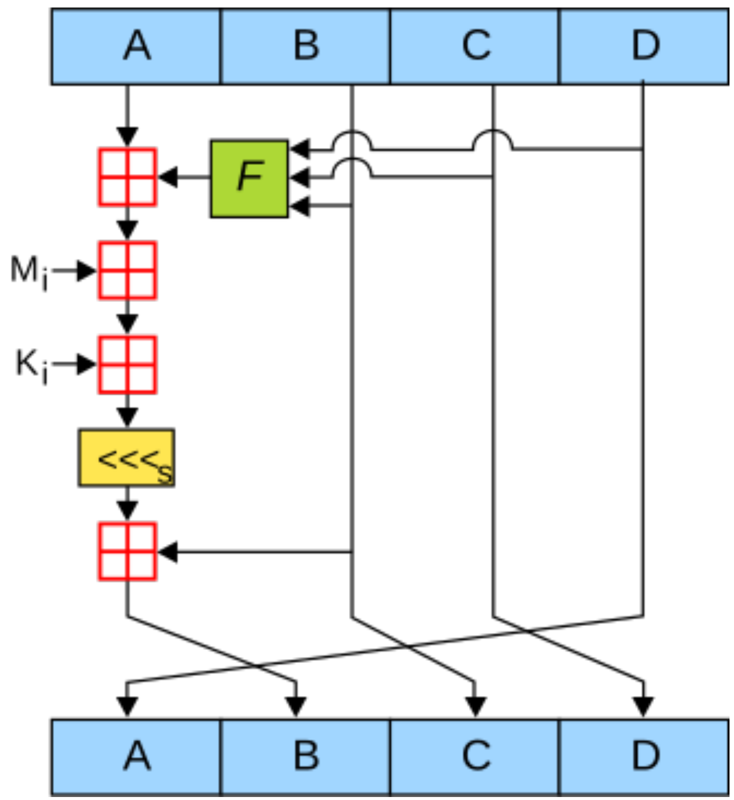
  F(X,Y,Z) = (B ∧ C) ∨ (¬B ∧ D)

  G(X,Y,Z) = XZ v Y not(Z)

  H(X,Y,Z) = X xor Y xor Z

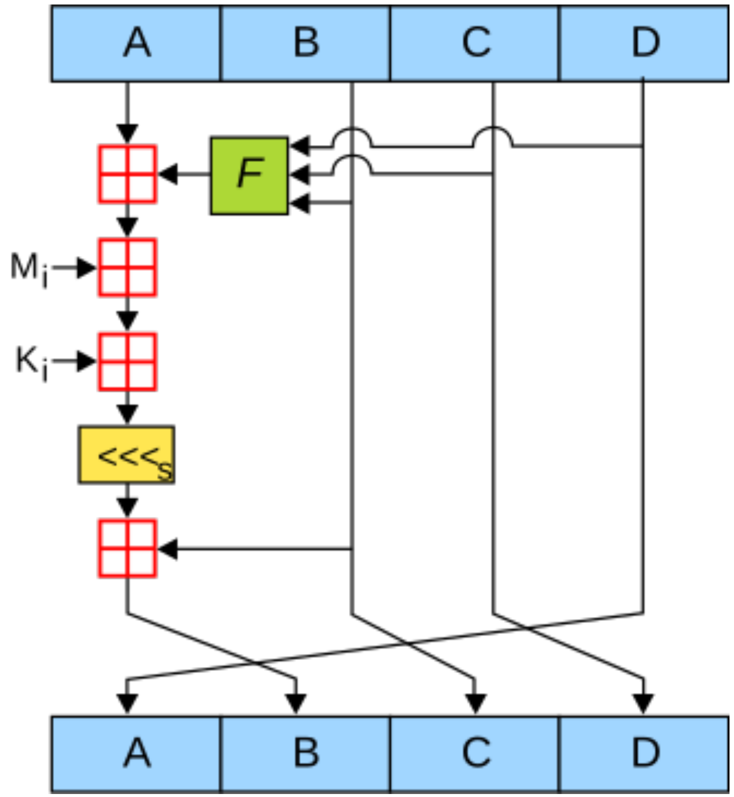
  I(X,Y,Z) = Y xor (X v not(Z))

- In each bit position F acts as a conditional: if X then Y else Z. The function F could have been defined using + instead of v since XY and not(X)Z will never have 1's in the same bit position.) It is interesting to note that if the bits of X, Y, and Z are independent and unbiased, the each bit of F(X,Y,Z) will be independent and unbiased.
  - word A: 92 25 47 e8
  - word B: 66 c8 9b 8f
  - word C: 67 73 12 df
  - word D: 0c ce c8 ee



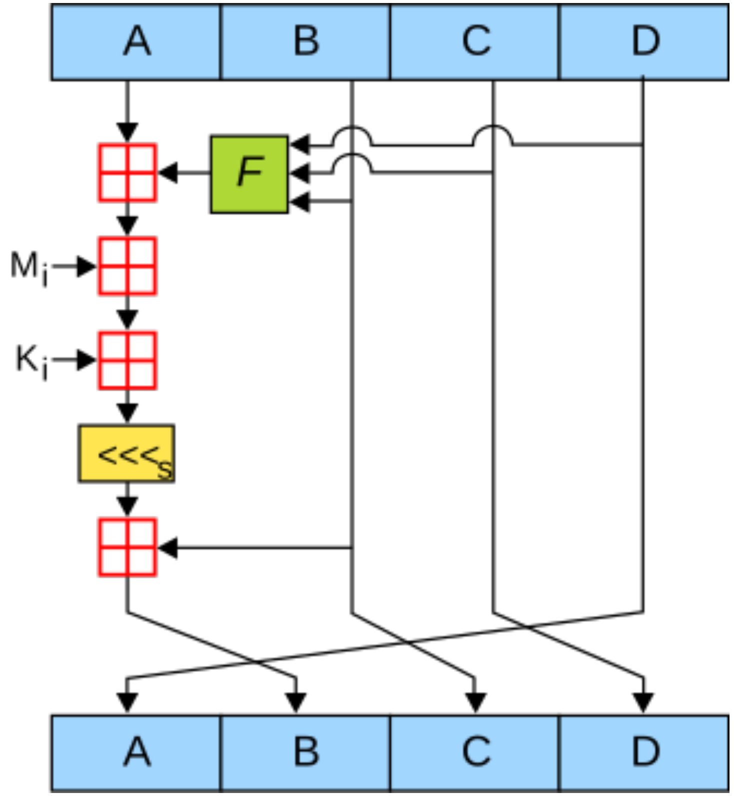
Hash Function: **MD5**

- **Step4 :Process Each 512-bit Block**
  - Split block to 16 × 32-bit words
  - 64 operations using F,G,H,I and T[**i**]
  - Update A,B,C,D



Hash Function: MD5

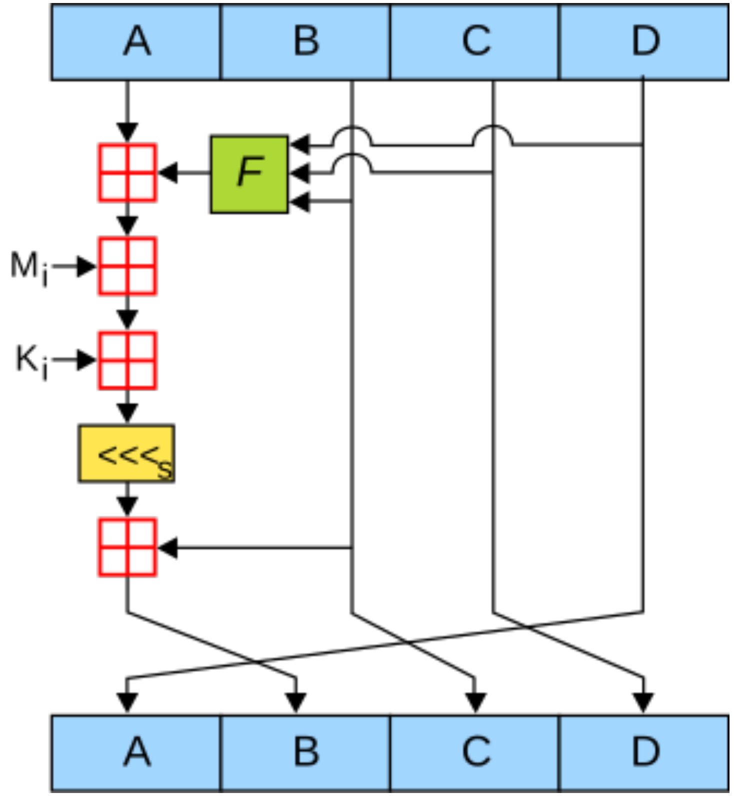
- **Step4 : Process Message in 16-Word Blocks**
  - Split block to 16 × 32-bit words
  - 64 operations using F,G,H,I and T[i]
  - Update A,B,C,D
  - **word** A: 922547e8
  - word B: 66c89b8f
  - word C: 677312df
  - word D: 0ccec8ee



Hash Function: MD5

- **Step 5 : Output**
  - 10010000 00000001 01010000 10011000 00111100 11010010 …(128bits)

  - Digest (32bits in hex): 922547e866c89b8f677312df0ccec8ee



Hash Function: MD5

- **Step 5 : Output**
  - **The message digest produced as output is A, B, C, D.**
  - This completes the description of MD5.

  - Hash of the input string (MD5) :
    922547e866c89b8f677312df0ccec8ee

Hash Function: SHA (Secure hash Algorithm)

- Developed by the U.S. National Security Agency (NSA) and standardized by NIST.
  - Data integrity
  - Message authentication
  - Digital signatures

Generation	Algorithms	Output (bits)	Status
SHA-0	(Original, 1993)	160	Withdrawn (flaws)
SHA-1	(1995)	160	Weak – broken (collisions)
SHA-2	SHA-224, SHA-256, SHA-384, SHA-512	224–512	Secure
SHA-3	SHA3-224, SHA3-256, SHA3-384, SHA3-512	224–512	Secure (new design)

Comparison of SHA functions

Algorithm and variant		Output size (bits)	Internal state size (bits)	Block size (bits)	Rounds	Operations	Security (bits)	Performance on Skylake (median cpb)[45]: Long messages	Performance on Skylake (median cpb)[45]: 8 bytes
MD5 (as reference)		128	128 (4 × 32)	512	4 (16 operations in each round)	And, Xor, Or, Rot, Add (mod 2^{32})	≤ 18 (collisions found)[46]	4.99	55.00
SHA-0		160	160 (5 × 32)	512	80	And, Xor, Or, Rot, Add (mod 2^{32})	< 34 (collisions found)	≈ SHA-1	≈ SHA-1
SHA-1							< 63 (collisions found)[47]	3.47	52.00
SHA-2	*SHA-224* *SHA-256*	224 256	256 (8 × 32)	512	64	And, Xor, Or, Rot, Shr, Add (mod 2^{32})	112 128	7.62 7.63	84.50 85.25
	SHA-384	384	512 (8 × 64)	1024	80	And, Xor, Or, Rot, Shr, Add (mod 2^{64})	192	5.12	135.75
	SHA-512	512					256	5.06	135.50
	SHA-512/224 *SHA-512/256*	224 256					112 128	≈ SHA-384	≈ SHA-384
SHA-3	*SHA3-224* *SHA3-256* *SHA3-384* *SHA3-512*	224 256 384 512	1600 (5 × 5 × 64)	1152 1088 832 576	24[48]	And, Xor, Rot, Not	112 128 192 256	8.12 8.59 11.06 15.88	154.25 155.50 164.00 164.00
	SHAKE128 *SHAKE256*	*d* (arbitrary) *d* (arbitrary)		1344 1088			min(*d*/2, 128) min(*d*/2, 256)	7.08 8.59	155.25 155.50

CADT

Hash Function: SHA Family

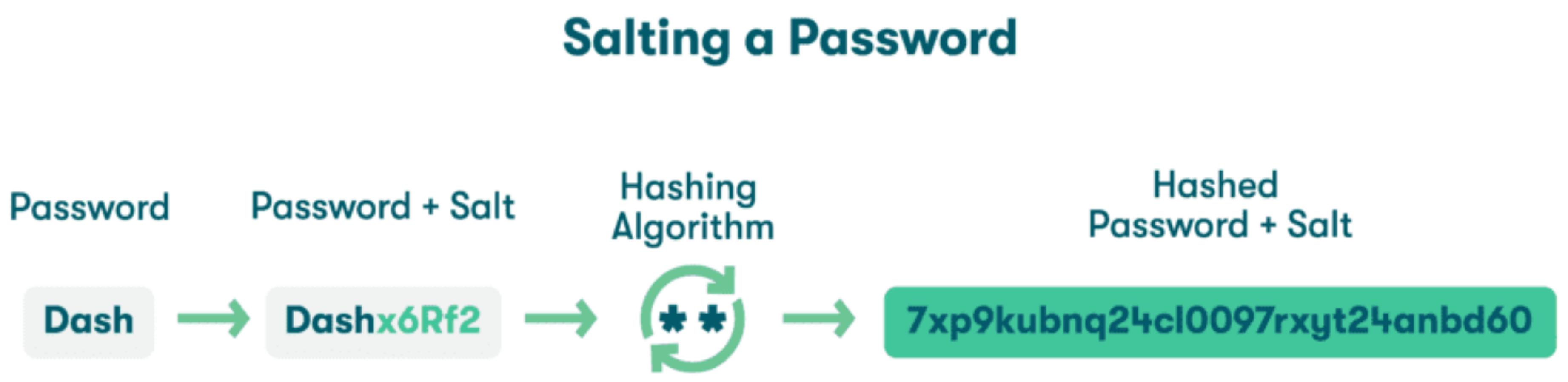
SHA-1 is history, SHA-2 is today, SHA-3 is the future.

Hash Function: Using Salt

A **salt** is a **random value** added to a password **before hashing**.

Why?
Prevents attackers from using **rainbow tables** (precomputed hash lists).
Ensures that **identical passwords produce different hashes.**

Hash Function: Using Salt

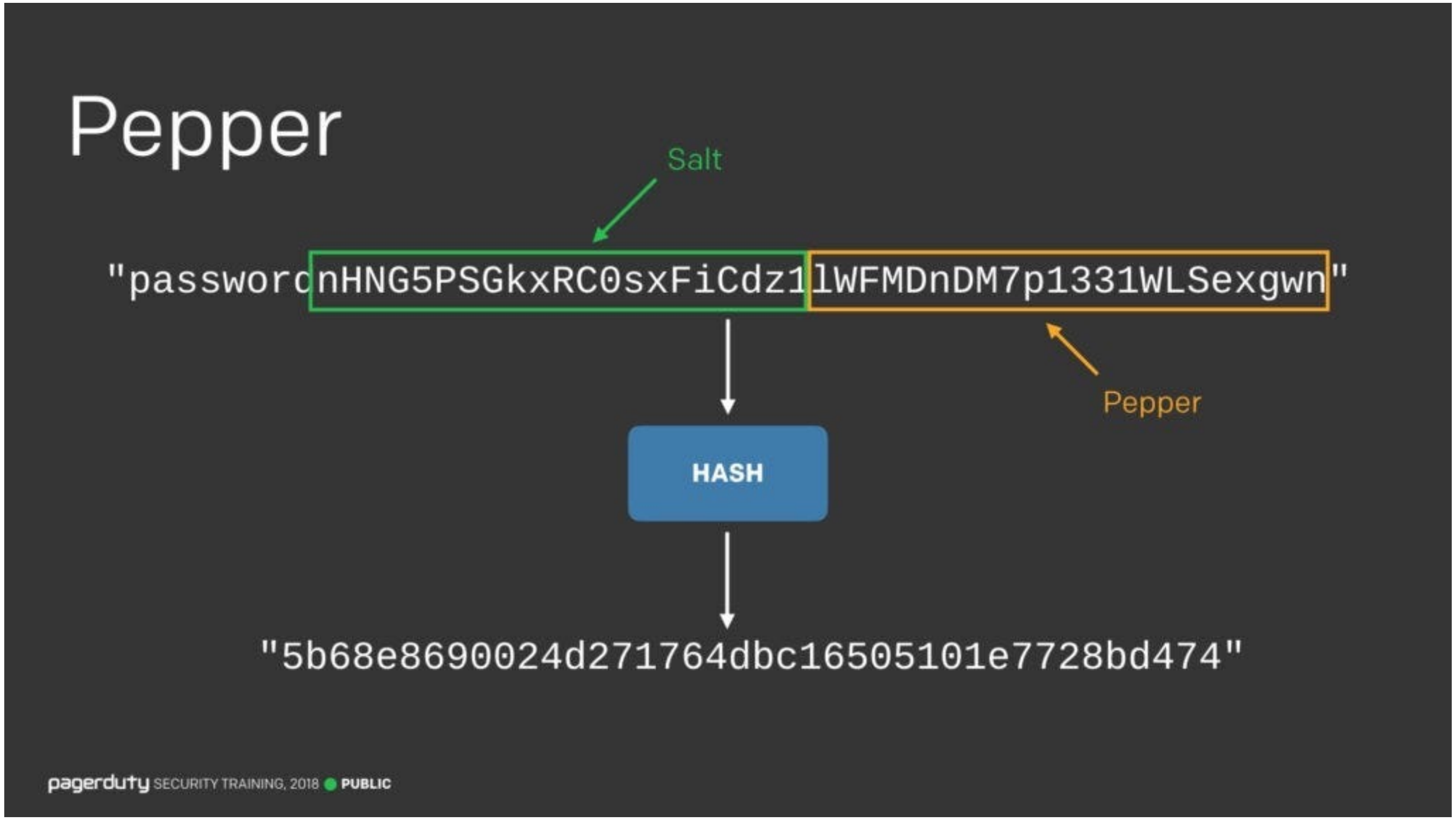


Hash Function: Using Pepper

A pepper is a **secret key/value** added to all passwords before hashing and kept **separate** from the database.

Why we use it?
Pepper protects against database theft
Even if attackers steal hashes + salts, they still need the secret pepper to crack them.

Hash Function: Salt vs Pepper



Salt is public and random value, (can storer in DB)

pepper is **secret key/value (don't store in DB)**

Hash Function: Password Hashing

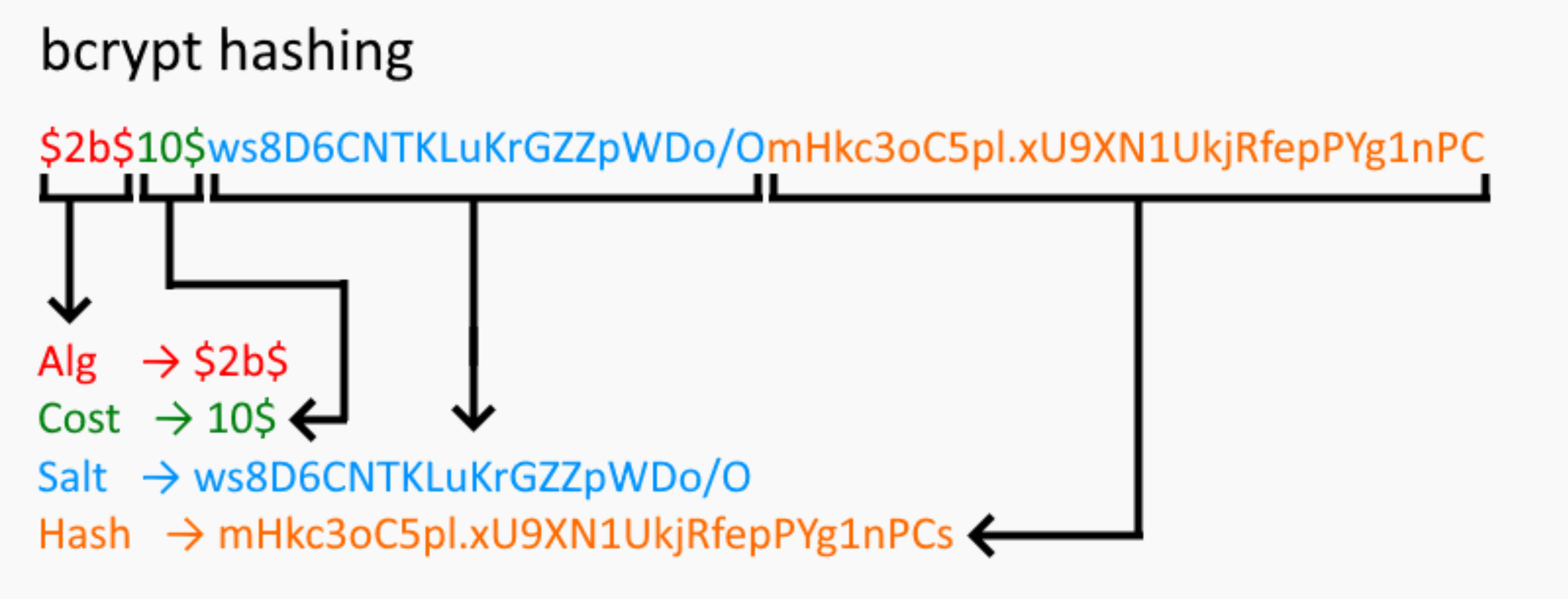
Bcrypt

- Created in **1999**, based on the **Blowfish cipher**.
- Designed *only* for secure password hashing.
- Automatically handles **salts** and allows setting a **cost factor** (work factor).

Structure form : $[version]$[cost]$[22-character-salt][31-character-hash]

Hash Function: Password Hashing

- **Bcrypt**



Hash Function: Password Hashing

Argon2

- Created in **2015**, winner of the **Password Hashing Competition (PHC)**.
- Designed to be **memory-hard** and **GPU-resistant.**
- Uses tunable parameters for **time**, **memory**, and **parallelism**.

Structure form :
$[algorithm]$v=[version]$m=[memory],t=[time],p=[parallelism]$[base64(salt)]$[base64(hash)]

Parameter	Increase	Decrease
m (memory) 64MiB	Uses more RAM → slows GPUs/ASICs	Easier for parallel attackers
t (time)	Longer to compute → higher brute-force cost	Faster but weaker
p (parallel)	Faster on multi-core CPUs	Slower but simpler

Hash Function: Why Simple Hashing Is Weak for Passwords

Attackers can:

- Use **rainbow tables** (precomputed hash lists),
- Use **brute-force** with GPUs,
- Exploit the fact that hashes are fast.

Hash Summary: **Tips**

- Always Salt **before Hashes**
- **Use** Slow Hashes for Passwords (**Bcrypt / Scrypt /**argon2...)
- Store only + salt (not plaintext)
- avoid fast hashes like MD5 or SHA-1 for passwords. (**EoL**)
- Hash != Encryption

- **Don’t** you know how to use hash properly?