# Applied Cryptography and Network Security

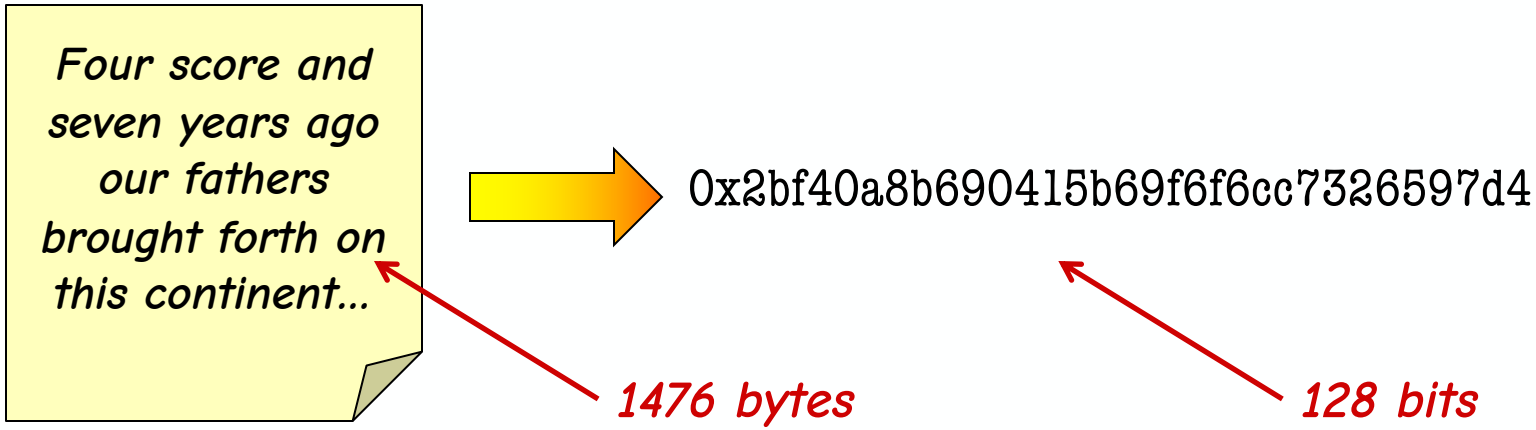
**William Garrison**
bill@cs.pitt.edu
6311 Sennott Square

Hash Function Construction

University of Pittsburgh

# Recall: What is a hash function?

***Definition:*** A hash function is a function that maps a variable-length input to a fixed-length code




Hash functions are sometimes called message digest functions

- SHA (e.g., SHA-1, SHA-256, SHA-3) stands for the secure hash algorithm
- MD5 stands for message digest algorithm (version 5)

# Recall: Cryptographic hash functions should have the following three properties

Assume that we have a hash function $H: \{0,1\}^* \rightarrow \{0,1\}^m$

1. Preimage resistance: Given a hash output value $z$, it should be infeasible to calculate a message $x$ such that $H(x) = z$
   - i.e., $H$ is a one-way function
   - Ideally, computing $x$ from $z$ should take $O(2^m)$ time

2. Second preimage resistance: Given a message $x$, it is infeasible to calculate a 2nd message $y$ such that $H(x) = H(y)$
   - Note that this attack is always possible given infinite time
   - Ideally, this attack should take $O(2^m)$ time

3. Collision resistance: It is infeasible to find two messages $x$ and $y$ such that $H(x) = H(y)$
   - Ideally, this attack should take $O\left(2^{\frac{m}{2}}\right)$ time

# Today, we'll look at how a few hash functions actually work!

It is perhaps unsurprising that hash functions utilize compression functions that are iterated many times

- Compression: Implied by the ability to map a large input to a small output
- Iteration: Helps "spread around" input perturbations

A compression function is $C: \{0,1\}^n \rightarrow \{0,1\}^m$, where $m < n$

- This is not lossless compression!
  - All inputs compress, consider pigeonhole
- Typically, a hash function maintains $m$ bits of state which are output as the digest at the end

Iteration does not need to be reversible!

- In fact we don't want it to be!
  - (Unlike block ciphers)
- No need for Feistel structure, and all bits can be mangled without all steps being 1-to-1

# Many hash functions are based on the Merkle-Damgård construction

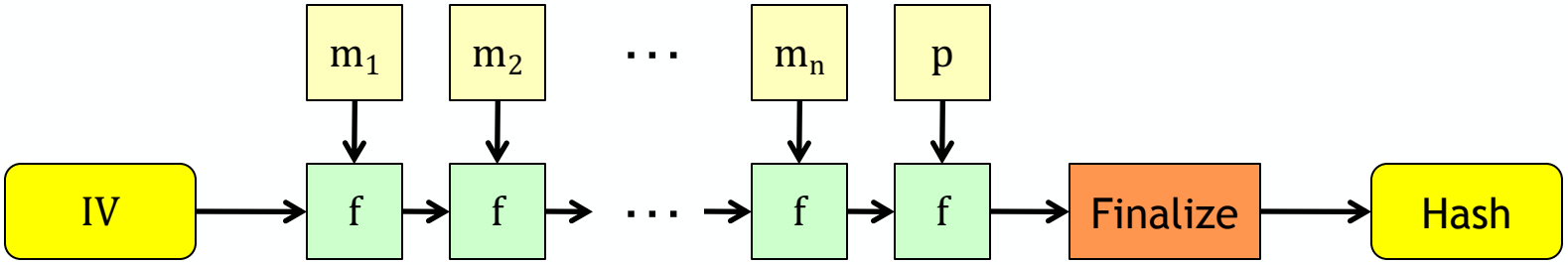
The Merkle-Damgård construction is a "template" for constructing cryptographic hash functions

- Proposed in the late '70s
- Named after Ralph Merkle and Ivan Damgård

Essentially, a Merkle-Damgård hash function does the following:

1. Pad the input message if necessary
2. Initialize the function with a (static) IV
3. Iterate over the message blocks, applying a compression function f
4. Finalize the hash block and output

*Why is a static IV ok?*




Merkle and Damgård (separately) showed that the resulting hash function is secure if the compression function is collision resistant

# A high-level overview of MD4

Input: A message of bit length $\leq 2^{64} - 1$

Output: A 128-bit digest

Steps:

- Pad message to a multiple of 512 bits
- Initialize four 32-bit words of state
- For each 512-bit chunk:
  - Break into sixteen 32-bit words
  - Complete 48 operations in 3 rounds
  - Each operation uses one word of input
  - Different constants and non-linear F used in each round
  - Add the results to the previous state
- Concatenate four 32-bit words of state and output as the digest

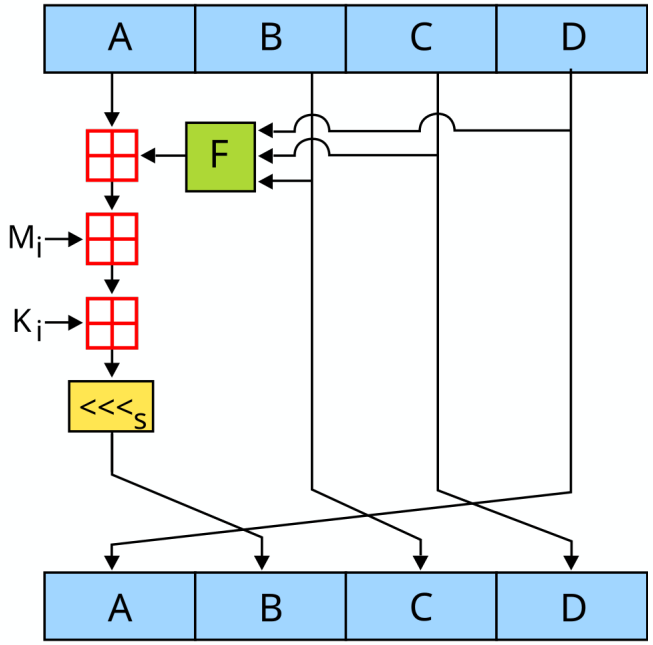



Image from Wikipedia

# Initialization and Padding

Initialize variables:

A = 0x67452301

B = 0xEFCDAB89

C = 0x98BADCFE

D = 0x10325476

Note: These variables comprise the internal state of MD4. They are continuously updated by the compression function and are used to construct the final 128-bit hash value.

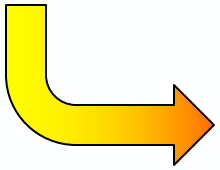
Pre-processing:

append the bit '1' to the message

append $0 \leq k < 512$ bits '0', so that the resulting message length (in bits)

is congruent to $448 \equiv -64 \pmod{512}$

append length of message (before pre-processing), in bits, as 64-bit big-endian integer



*Example:*

0xDEADBEEF → 0xDEADBEEF8000 ... 0020

*32 bits*

*$32_{10}$ = 0x20*

# Each iteration uses one word of input

Each operation does the following:

- $B' = (A + F(B, C, D) + M_i + K) \lll s$
- $C' = B$
- $D' = C$
- $A' = D$
- $M_i$ is word $i$ of this chunk

*After all 48 operations, A,B,C,D are added to their previous values*

Each round is comprised of 16 operations, one per input word

- Round 1: $F(B, C, D) = BC \vee (\neg B)C$, $K = 0$
  - $s$ is 3, 7, 11, 19, then repeats (4×)
  - Words are used in normal order $(0, 1, 2, 3, \ldots)$
- Round 2: $F(B, C, D) = BC \vee BD \vee CD$, $K = \sqrt{2} = 0x5A827999$
  - $s$ is 3, 5, 9, 13, then repeats (4×)
  - Words used in order $0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15$
- Round 1: $F(B, C, D) = B \oplus C \oplus D$, $K = \sqrt{3} = 0x6ED9EBA1$
  - $s$ is 3, 9, 11, 15, then repeats (4×)
  - Words used in order $0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15$

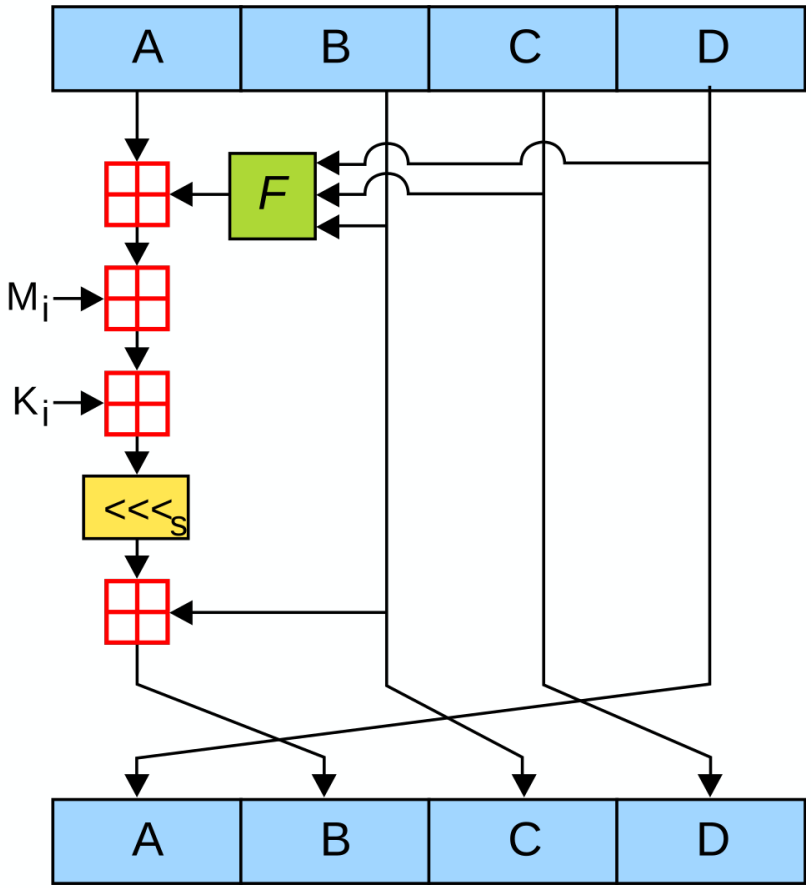
# MD5 is very similar to MD4 with some additional complexity

MD5 is based on the same Merkle-Damgård construction

- … and uses the same initialization and padding

Differences:

- 4 rounds of 16 operations each
- Four new non-linear functions
  - $F_1(B,C,D) = (B \wedge C) \vee (\neg B \wedge D)$
  - $F_2(B,C,D) = (B \wedge D) \vee (C \wedge \neg D)$
  - $F_3(B,C,D) = B \oplus C \oplus D$
  - $F_4(B,C,D) = C \oplus (B \vee \neg D)$
- Different orders for using words (as with MD4, order changes each round)
- Constants are sines, not $\int$



# SHA-1 follows many of these conventions, but with a 160-bit state/output

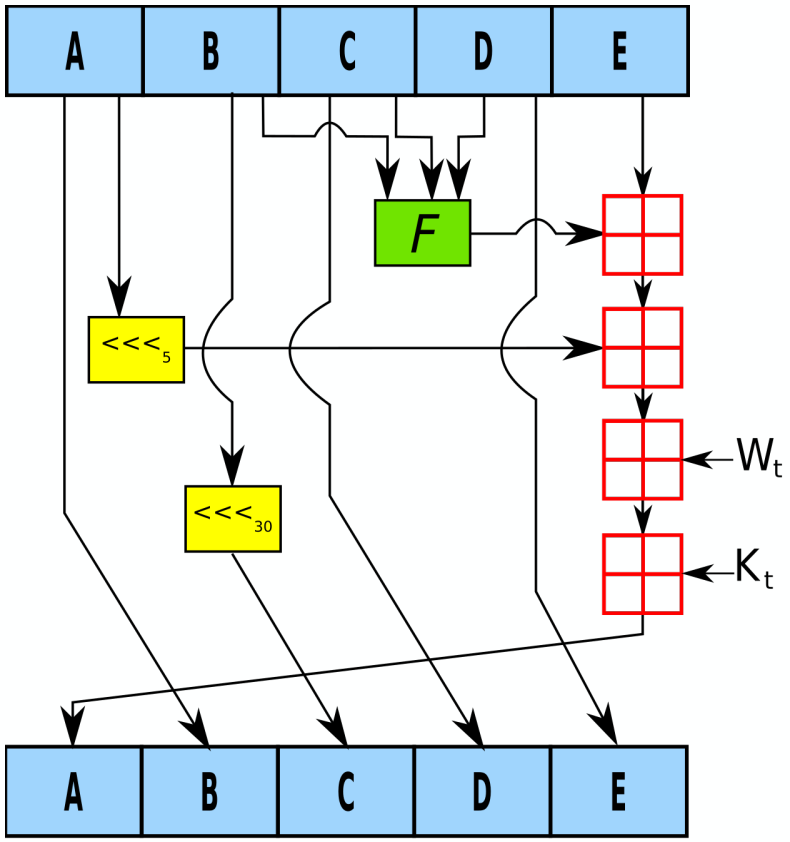
Initialization values:

- $h_0$ = 0x67452301
- $h_1$ = 0xEFCDAB89
- $h_2$ = 0x98BADCFE
- $h_3$ = 0x10325476
- $h_4$ = 0xC3D2E1F0

Padding: Same as MD4 and MD5

For each 512-bit chunk:

- Break it into 16 words
- The 16 words from the input are expanded to 80 words
- 80 operations grouped into 4 rounds
- F, as before, is the non-linear function, and changes between rounds



# Initializing the compression function

Process the message in successive 512-bit chunks:

break message into 512-bit chunks

for each chunk

    break chunk into sixteen 32-bit big-endian words w[i], 0 ≤ i ≤ 15

    Extend the sixteen 32-bit words into eighty 32-bit words:

    for i from 16 to 79

        w[i] = (w[i-3] xor w[i-8] xor w[i-14] xor w[i-16]) <<< 1

    Initialize hash value for this chunk:

    a = h0

    b = h1

    c = h2

    d = h3

    e = h4

# Main body of the compression function

Main loop:

*Note: Sometimes, we treat state as a bit vector...*

```
for i from 0 to 79
        if 0 ≤ i ≤ 19 then
                f = (b and c) or ((not b) and d); k = 0x5A827999
        else if 20 ≤ i ≤ 39
                f = b xor c xor d; k = 0x6ED9EBA1
        else if 40 ≤ i ≤ 59
                f = (b and c) or (b and d) or (c and d); k = 0x8F1BBCDC
        else if 60 ≤ i ≤ 79
                f = b xor c xor d; k = 0xCA62C1D6

        temp = (a <<< 5) + f + e + k + w[i]
        e = d; d = c; c = b <<< 30; b = a; a = temp
```

*... but other times, it is treated as an unsigned integer*

Add this chunk's hash to result so far:

```
h0 = h0 + a; h1 = h1 + b; h2 = h2 + c; h3 = h3 + d; h4 = h4 + e
```

# Finalizing the result

Produce the final hash value (big-endian):

output = h0 || h1 || h2 || h3 || h4

*"||" denotes concatenation*

Interesting note:

- $k_1$ = 0x5A827999 = $2^{30} \times \sqrt{2}$
- $k_2$ = 0x6ED9EBA1 = $2^{30} \times \sqrt{3}$
- $k_3$ = 0x8F1BBCDC = $2^{30} \times \sqrt{5}$
- $k_4$ = 0xCA62C1D6 = $2^{30} \times \sqrt{10}$

*Question: Why might it make sense to choose the k values for SHA-1 in this manner?*

# SHA-2 added more complexity and larger output modes (e.g., SHA2-256)

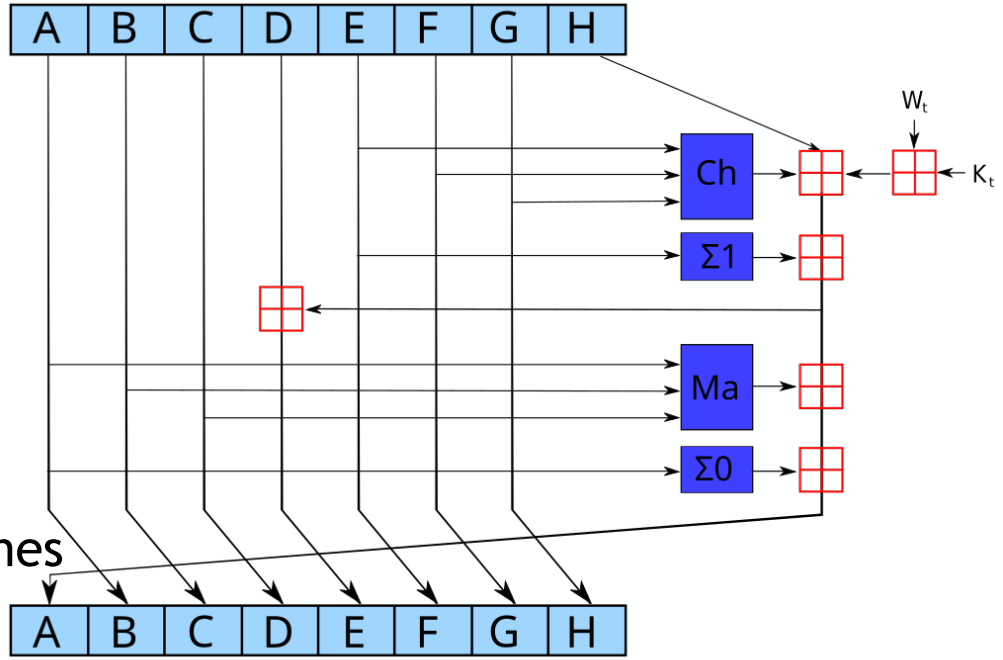
## Initialization values:

- Square roots of the first 8 primes

## Padding: Same as MD4 through SHA-1

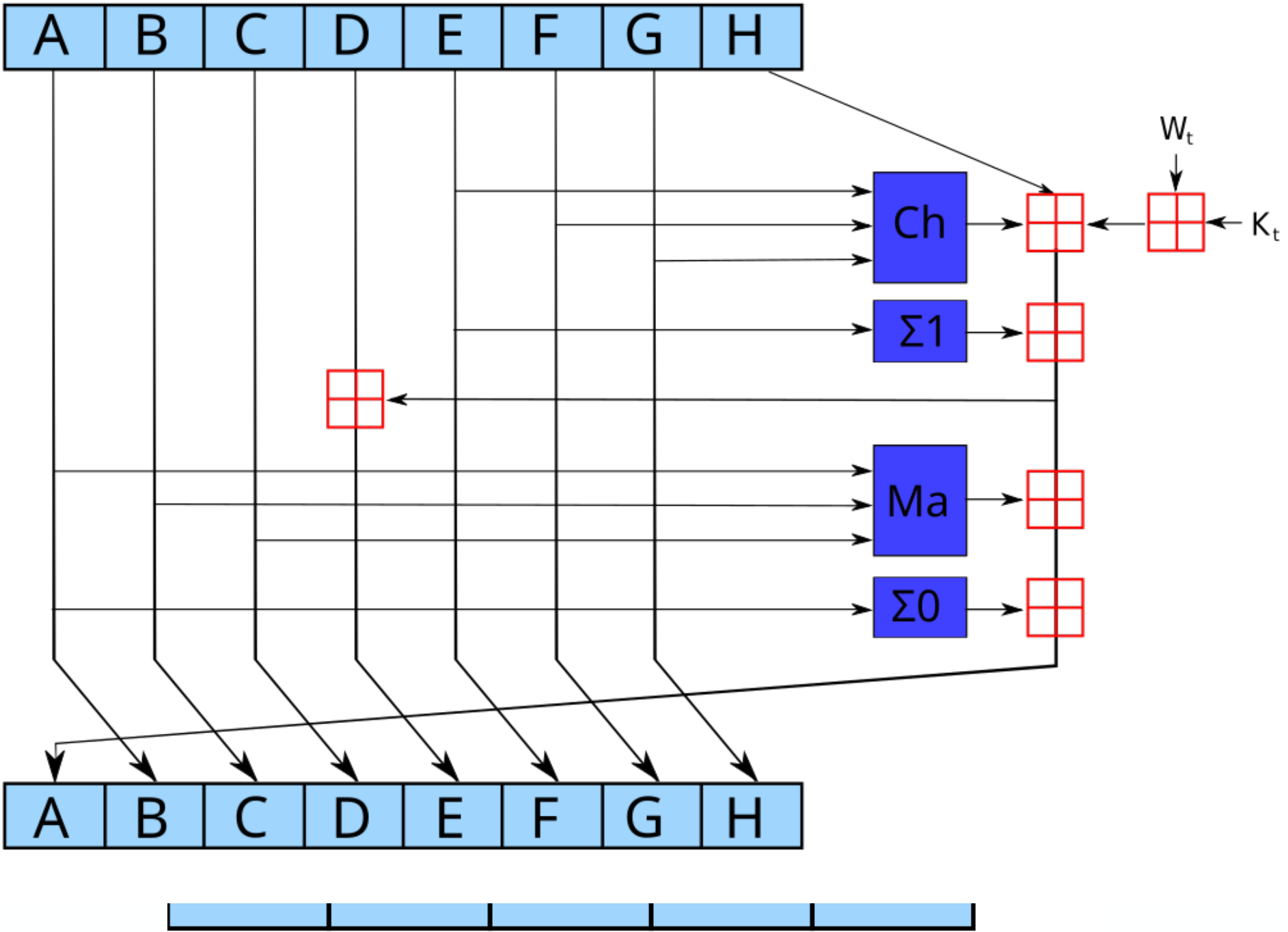
## Round constants:

- Different for each operation
- Cube roots of the first 64 primes




## For each 512-bit chunk:

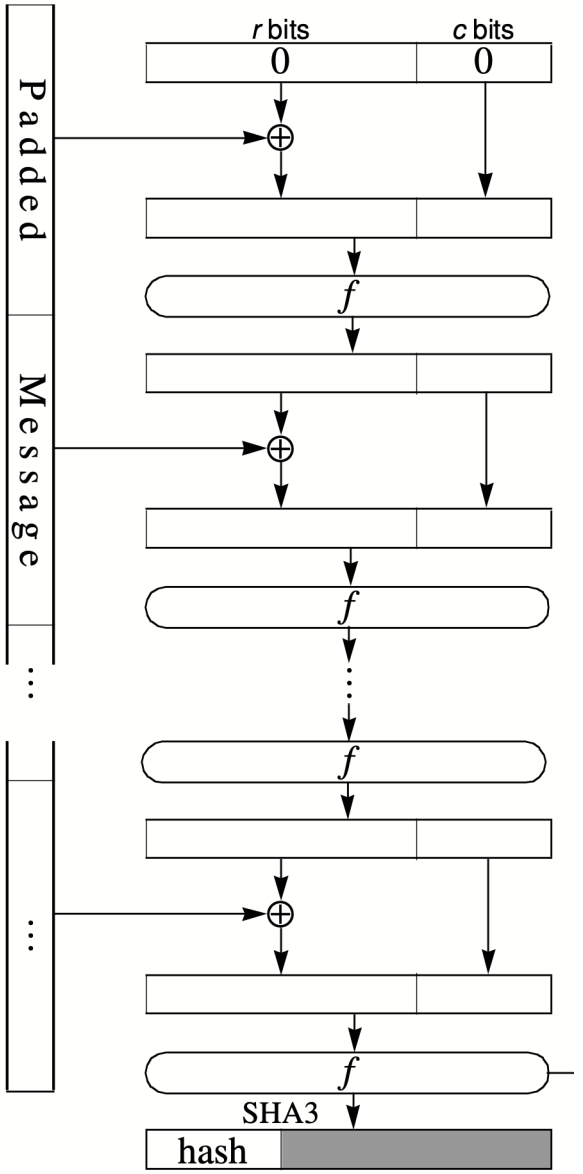
- Break into 16 words and expand to 64 words (for 64 operations)
  - Similar to SHA-1
- The non-linear steps are shown as Ch, Ma, Σ1, and Σ0


A
B
C
D
E
F
G
H
Ch
Σ1
Ma
Σ0
$W_t$
$K_t$
A
B
C
D
E
F
G
H

# SHA-3 (Keccak) uses a novel sponge construction rather than Merkle-Damgård

Sponge construction with two-part internal state

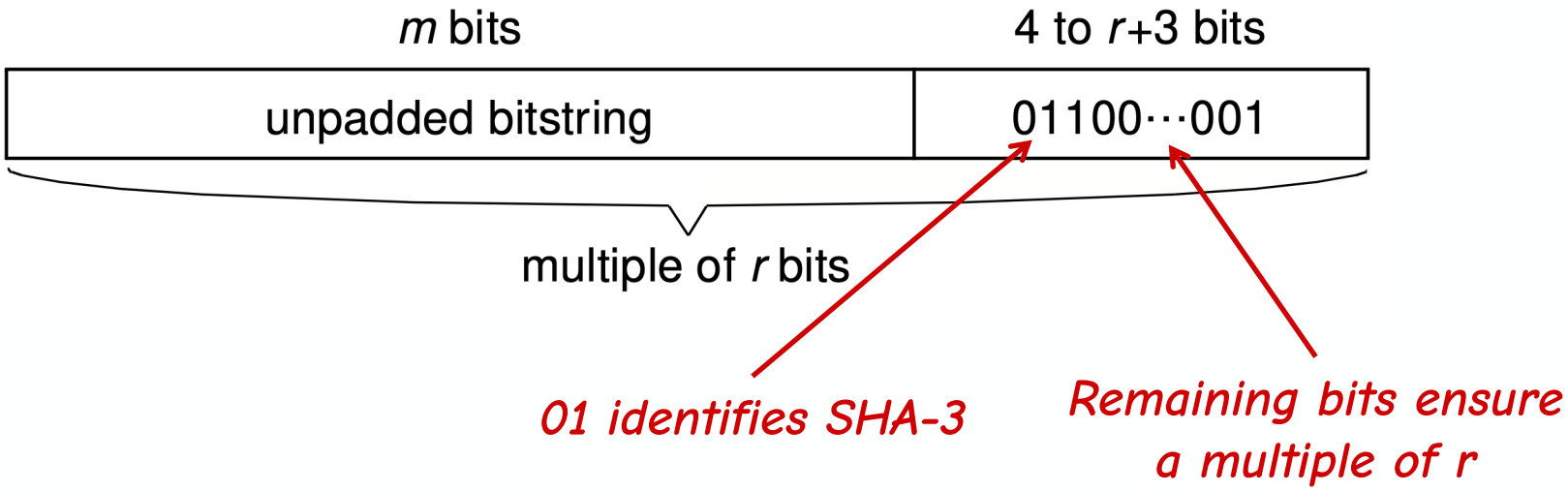
- Total of 1600 bits of state
- Split into $r$ (rate) and $c$ (capacity)
  - Rate: Determines how much input is absorbed at a time
  - Capacity: Determines the security level
  - Capacity is set to twice the output size
- Chunks of data are XOR'd only into $r$
  - This means smaller $c$ is faster
- In between each chunk, function $f$ permutes entire state
- After all chunks are processed, output the first bits of $r$
  - (However many are needed)



# SHA-3 padding

Padding in Keccak is different from MD*, SHA-1, and SHA-2

- Input must be padded to be an even multiple of $r$
- Append 01 to the input, then 100…001 of the appropriate length



# SHA-3 permutation function

The function $f$:

- Imagines the state as a 5×5×64 array of bits
  - Can operate on rows, columns, or lanes
- Uses 5 steps of efficient bitwise operations
- Uses 64-bit words for modern CPUs

Steps:

- θ: Compute the parity of each 5-bit column and XOR into nearby columns
- ρ: Rotate each 64-bit word (lane-wise) by a different triangular number
  - 0, 1, 3, 6, 10, 15, …
- π: Permute the 25 lanes in a fixed pattern
  - Note that ρ and π, combined, mix across all 3 dimensions
- χ: Combine bits along rows (non-linear step!)
- ι: XOR a 7-bit round constant into parts of state
  - The only step that differs per round

# Keccak is designed for multiple uses!

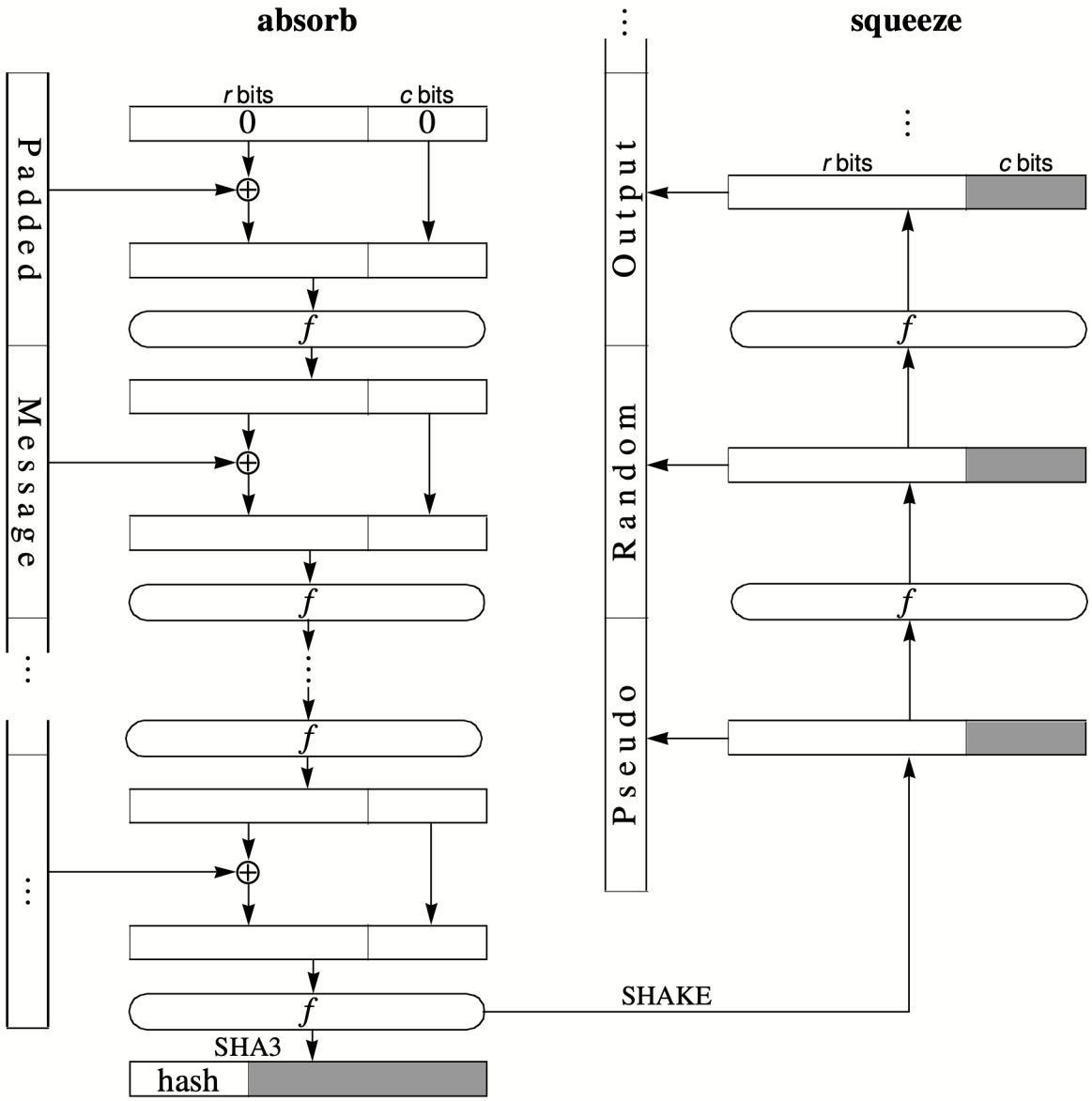
In addition to generating a fixed-length hash output, Keccak can be used (as SHAKE) to generate an arbitrary-length stream of pseudorandom bits from its input

- XOF: Extendable-output function
  - Why might one want this?
- Input stage is "absorb," output stage is "squeeze"
  - For SHA-3, no squeeze needed since $r$ is greater than output size
- Can also be tuned for different security levels

Differences between SHA-3 mode and SHAKE mode:

- Initial padding uses separator 1111 rather than 01
- To output more bits than $r$, run $f$ again and output more!

# The full sponge structure of Keccak (SHA-3 and SHAKE)



# Summary and wrap-up

Today we looked at a lot of hash function details

- Common themes:
  - Efficient operations, heavy repetition
  - Interpreting state differently between steps (linear vs. non-linear)
  - Compression without the need for reversibility
  - Permutations to spread around each bit's impact

We just overviewed about 30 years of hashing history!

- What trends did you notice as we got more modern?

Next: Public-key cryptography