Hash function and MACs

- Try to "compress" the message before we are going to sign it.

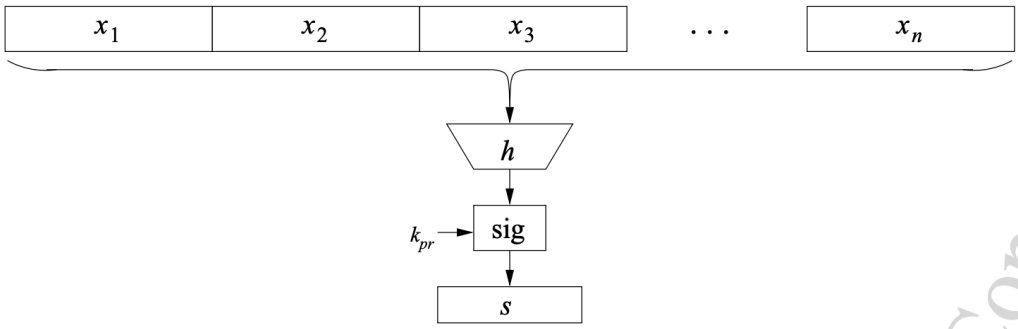
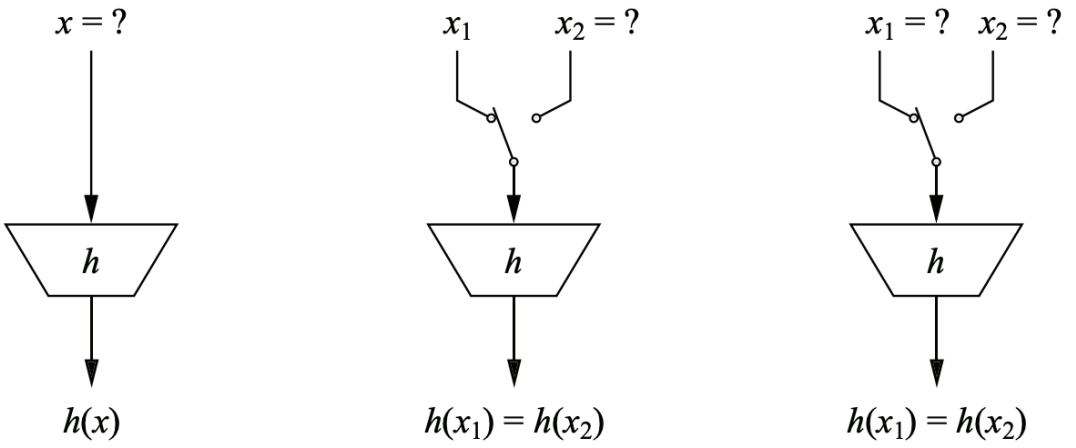


Fig. 11.2 Signing of long messages with a hash function

And the "compression" is done by a hash function

Requirements for Hash function

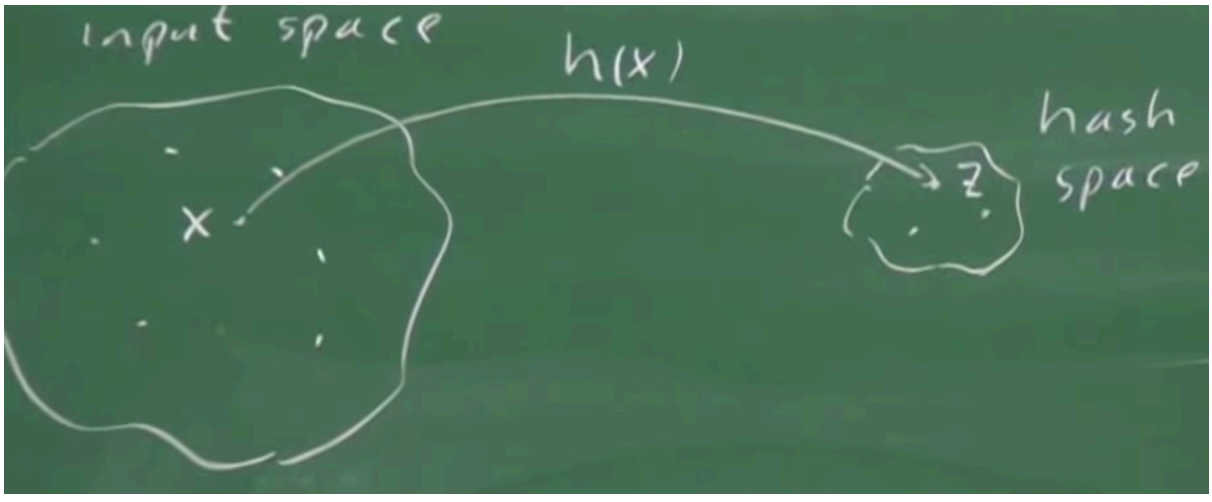
- Arbitrary input length
- Fixed, short output length
- Fast and efficient
- "Preimage Resistance"
- "Second Preimage Resistance" or "weak collision resistance"
- Collision Resistance



(actually, the "preimage" means the input of hash function)

It is easier to do a collision attack than a second preimage resistance

- All hash functions have a collision !!!



  - the input space(can be infinite according to the first requirement) is much larger than the output space(which is finite)

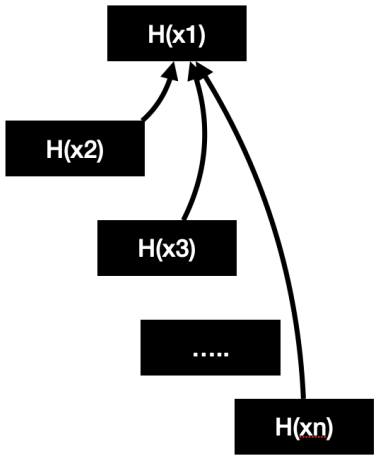
- say that we have a Hash function whose output is 3 bits long

  then as a hacker, we can calculate all possible for the hash space

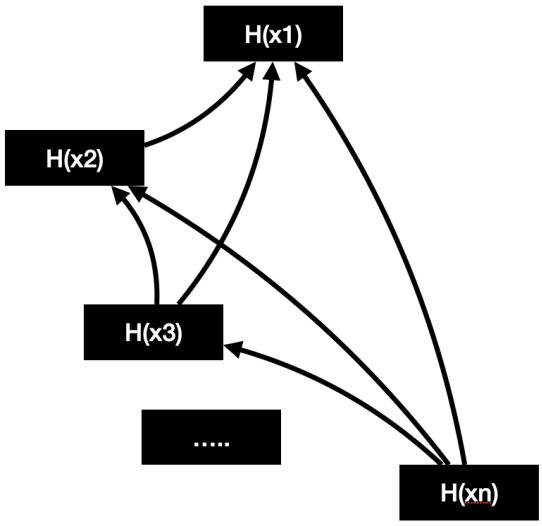
$$
\begin{aligned}
h(x_1) &= 000 = 0 \\
h(x_2) &= 001 = 1 \\
h(x_3) &= 010 = 2 \\
h(x_4) &= 011 = 3 \\
h(x_5) &= 100 = 4 \\
h(x_6) &= 101 = 5 \\
h(x_7) &= 110 = 6 \\
h(x_8) &= 111 = 7
\end{aligned}
$$

  Then the x_9, not matter which one it collide with, should be the collision we want to find, and we have $x_{10}, x_{11}, \dots$

- second preimage attack

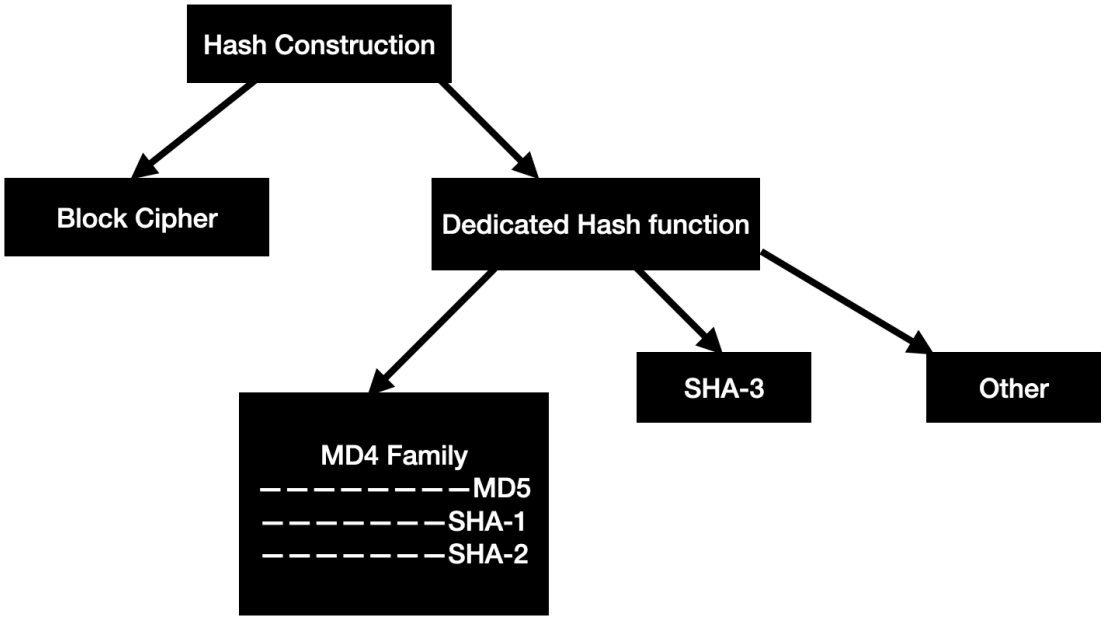


- collision attack



According to **Birthday paradox**, it only need $2^{\frac{n+1}{2}}\sqrt{\ln\left(\frac{1}{1-\lambda}\right)}$ inputs to find a collision with probability of λ

How do we build a hash function?


Hash Construction
Block Cipher
Dedicated Hash function
SHA-3
Other
MD4 Family
MD5
SHA-1
SHA-2

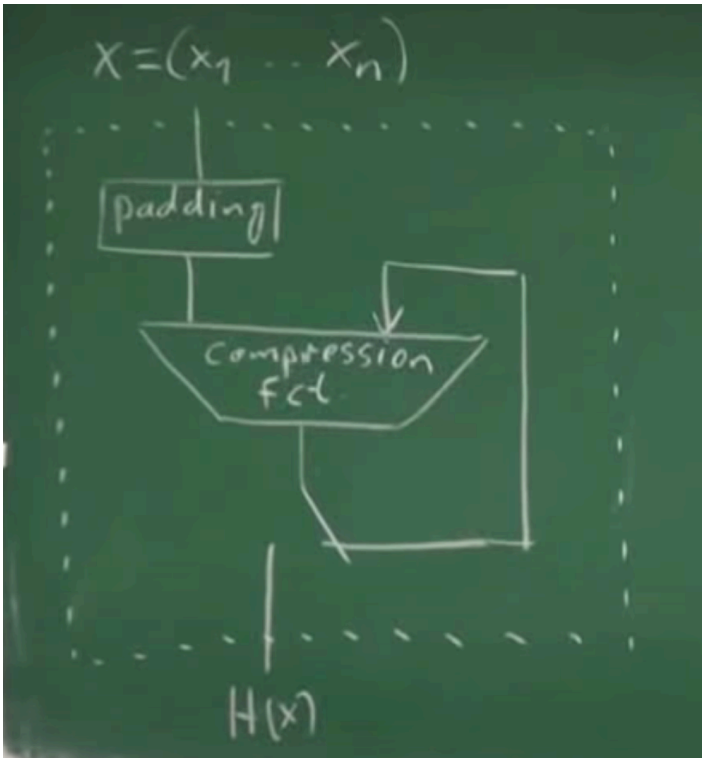
MD4 family

Algorithm	Output length	Input length(block length)	Number of rounds	Collision Found or not
MD5	128	512	64	Yes
SHA-1	160	512	80	not yet
SHA-224(SHA-2)	224	512	64	No
SHA-256(SHA-2)	256	512	64	No
SHA-384(SHA-2)	384	1024	80	No
SHA-512(SHA-2)	512	1024	80	No

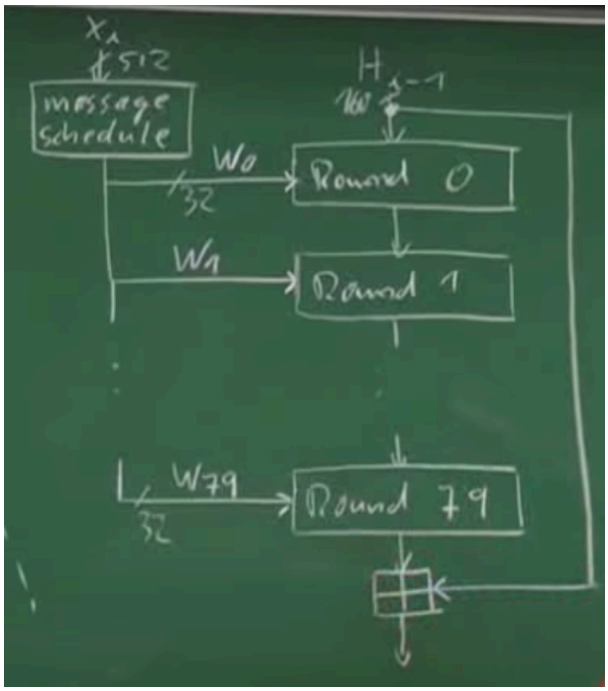
SHA(Security Hash Algorithm)

Overview of SHA-1

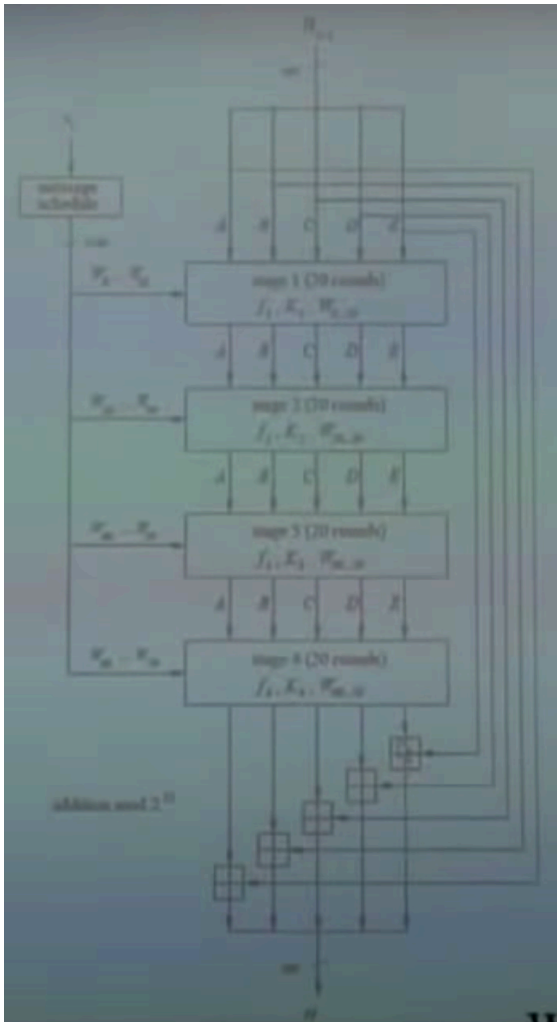
- Aribitrary input length, 512 bits long output
- SHA-1 use a Merkle–Damgård construction

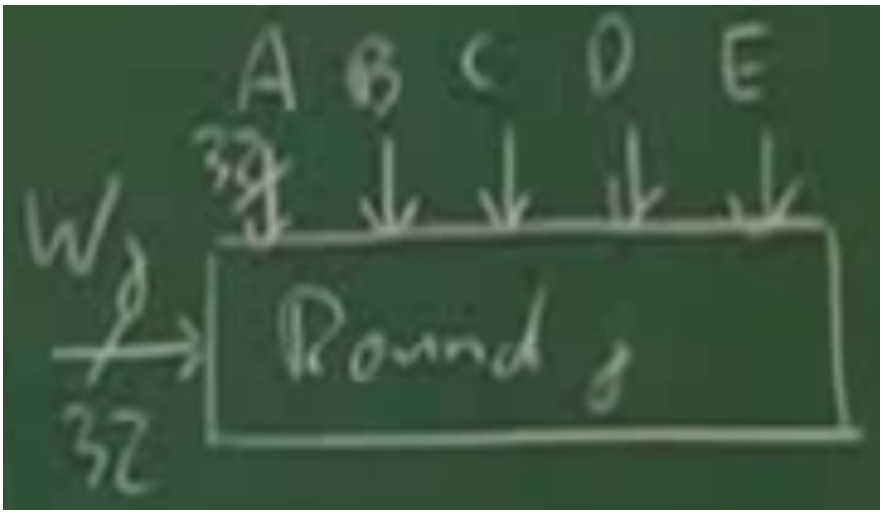


- inside the compression function, we have something like this:
  - the message serve as the key in the block cipher



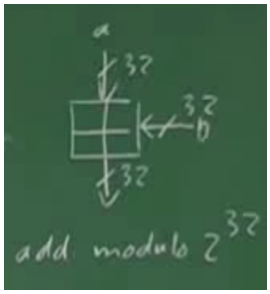
- 80 rounds are splited into 4 stage, each one with 20 rounds



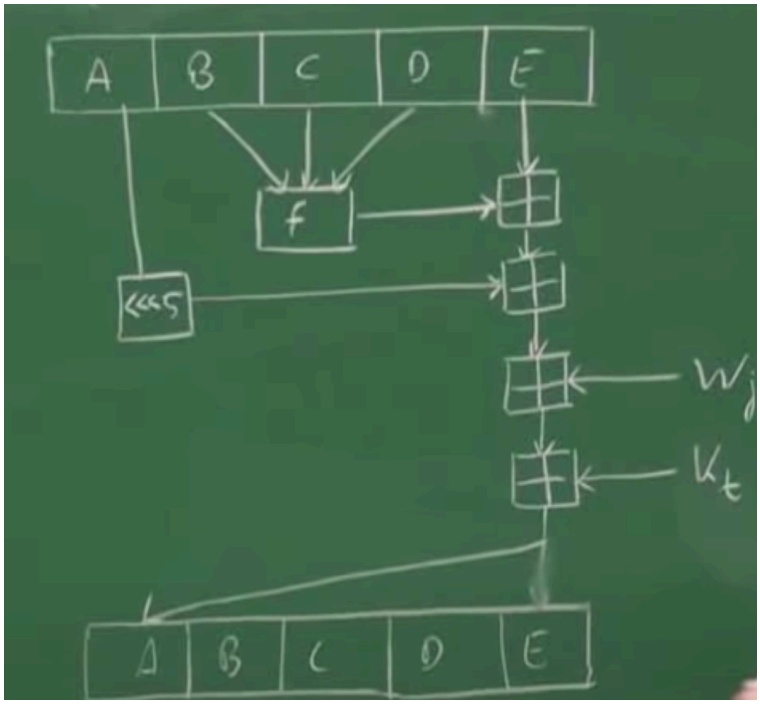


$5 \times 32 = 160$ bits input, 32 message bits input

- the addition here is a addition modulo 2^{32}, which means we drop the carry bit

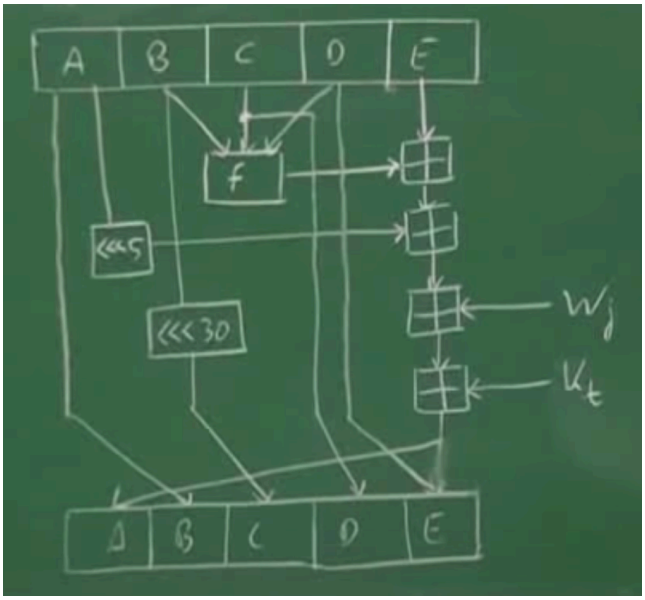


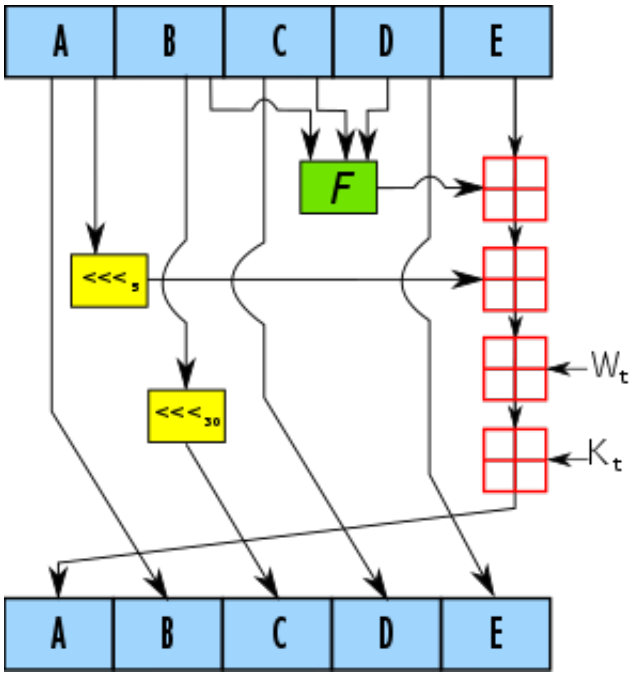
- Inside the round, we do something like fistel encryption, use block A,B,C,D to encrypt block E(w_j is 32 bits long message(serve as the key,k_t is stage signal, change with the stage)



the output:

```
A,B,C,D,E = encrypted_E,A,rotated_B,C,D
```





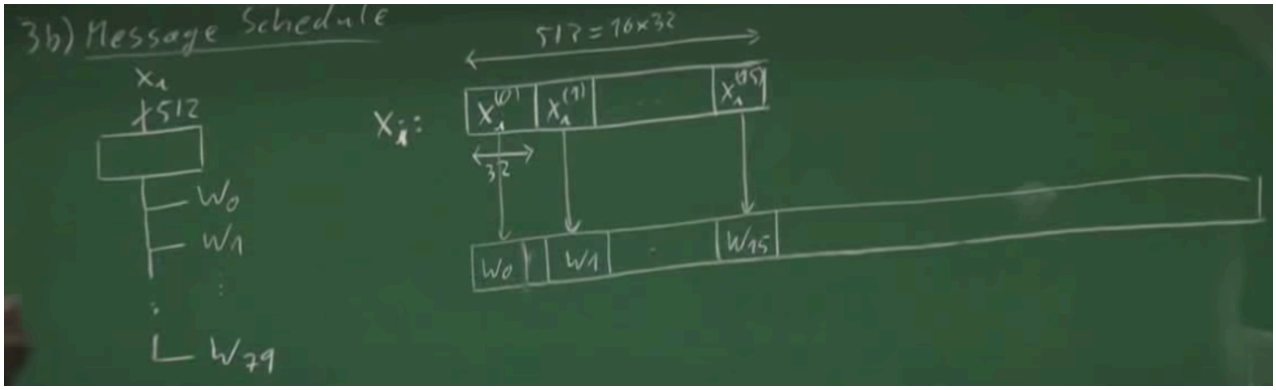
- rotation here do not drop bits, the overflow bits are concatened

```
0b01010111 << 1 = 0b10101110
0b01010111 << 2 = 0b01011101
....
```

- the f is actual f_t, it depends on different stages

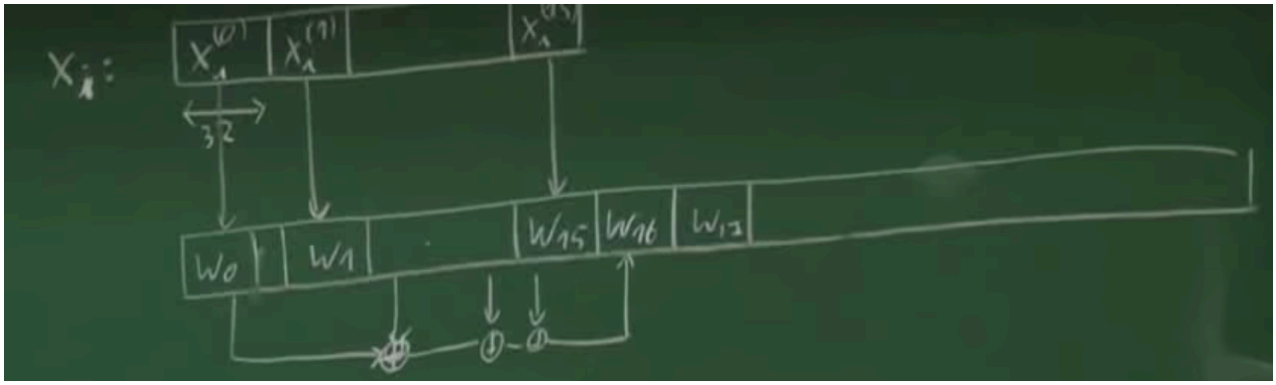
Stage	Round	Constant K_t	Function $F_t(B, C, D)$
1	0-19	0x5A827999	$(B \wedge C) \vee (\neg B \wedge D)$
2	20-39	0x6ED9EBA1	$B \oplus C \oplus D$
3	40-59	0x8F1BBCDC	$(B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$
4	60-79	0xCA62C1D6	$B \oplus C \oplus D$

Message Schedule



- each 512 block bits go through 80 rounds, yield 32 bits each round
- the first 16 words(32 bits) are just a duplication of the origin one.
- for the rest of them

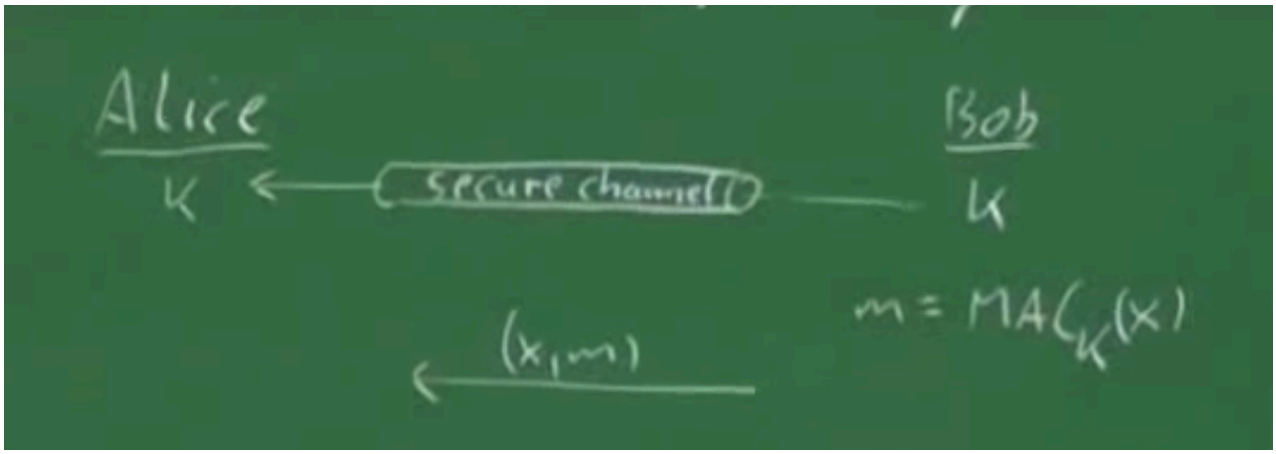
$$w_j = w_{j-16} \oplus w_{j-14} \oplus w_{j-8} \oplus w_{j-3}, 16 \leq j \leq 79$$



MACs

message authentication codes, which is also call "cryptography checksum"

Symmetric MACs



Alice only need to recompute the MACs with received `x` and compare it with the received one.

```
# Alice
mac = MACs(k,x)
print(mac == received_mac)
```

If this check out Alice knows that the message `x` must come from some one who knows the key.

Properties of MACs

- Arbitrary input length
- Fixed output length
- Message authentication : the message must come from someone who you want it to come from
- Integrity : manipulation will be dectected by doing the final check.
- Non-reproduction is NOT given.(Alice can generate a `x` and then say Bob send this to her)

HMAC(Hash based MAC)

```
mac = HMAC(k,x)
# how we build HMAC ?
# + means concate here
def HMAC1(k,x):
    return hash(bytes(k) + bytes(x)) # secret
prefix
def HMAC2(k,x):
    return hash(bytes(x)+bytes(k)) # secret
suffix
```

Unfortunately, both of them are weak

Hash length extension attack

video

tool

- when we compute a hash value, we split it into blocks

```
# just Python style pseudo code
def hash(k,x):
    inp = bytes(k) + bytes(x)
    next_round_input =
initial_value_wich_is_given
    while inp:
        next_round_input = compress(inp[:512]
, next_round_input)
        inp = inp[512:]
    return next_round_input

# as an attacker, we can extend the input

my_input = bytes(k) + bytes(x) +
bytes(my_message)
    next_round_input = hash(k,x)
    while my_input:
```

```
        next_round_input = compress(inp[:512]
, next_round_input)
        inp = inp[512:]
    return next_round_input
```

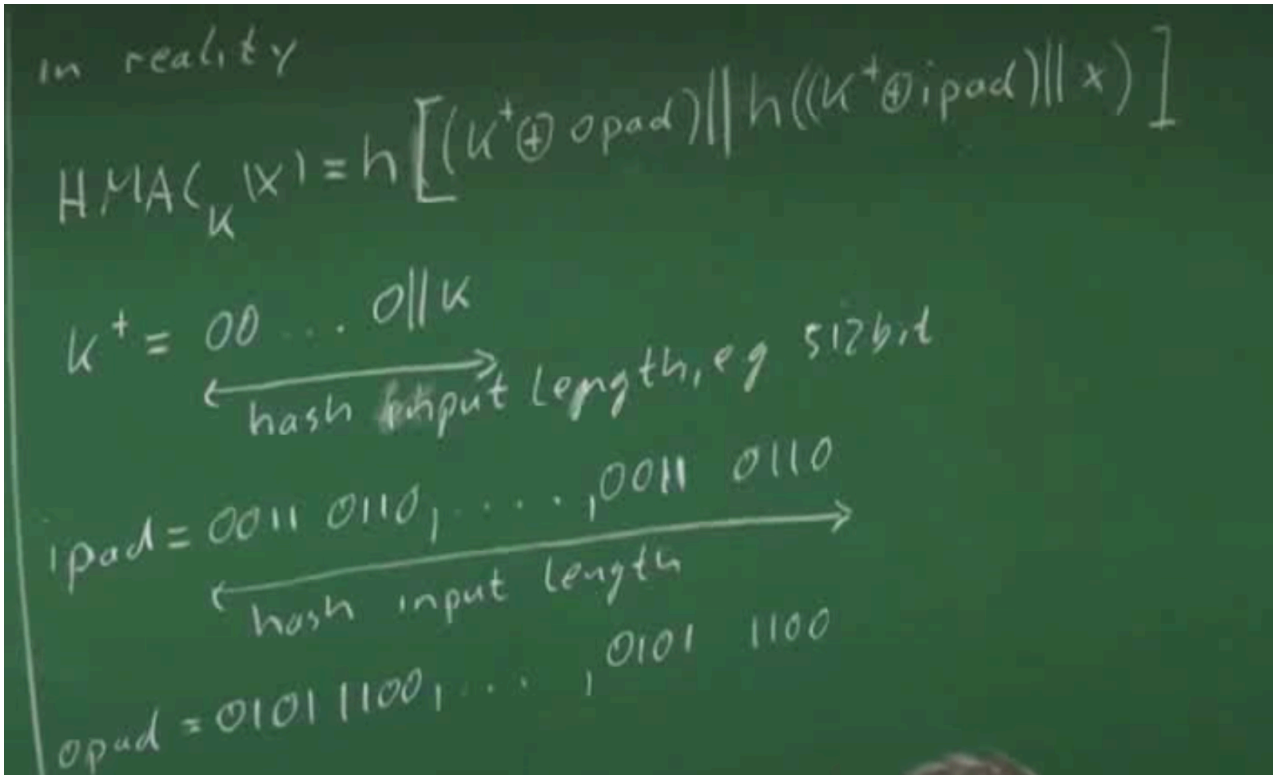
- we can prevent this kind of attack by padding the length of the original message

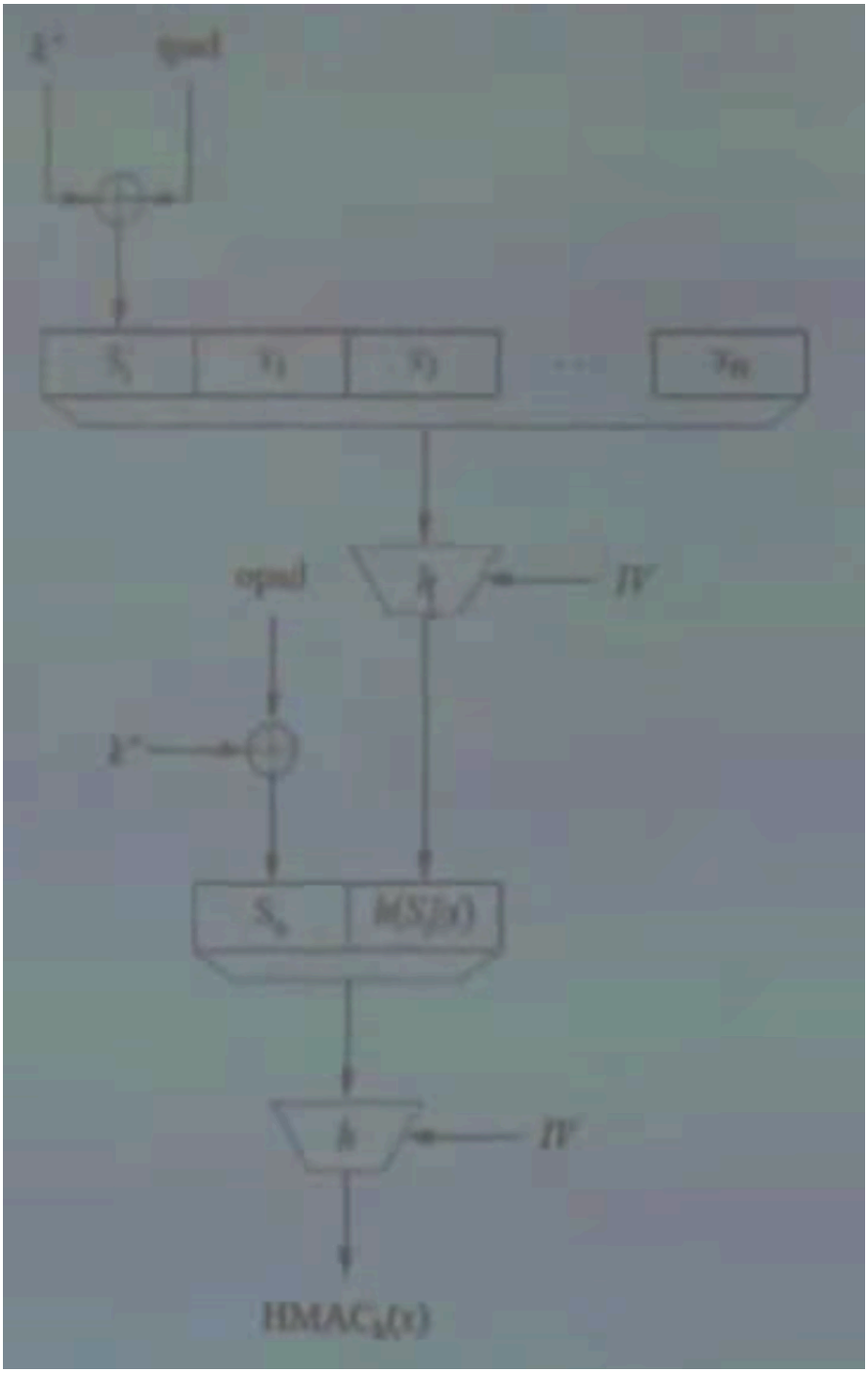
HMAC construction

This is what we do in SSL/TLS

- we use 2 nested secret prefix

```
hmac = HMAC(bytes(k) + bytes(HMAC(
bytes(k)+bytes(x))))
```




opad
opad
IV
IV
$\mathrm{HMAC}_K(x)$