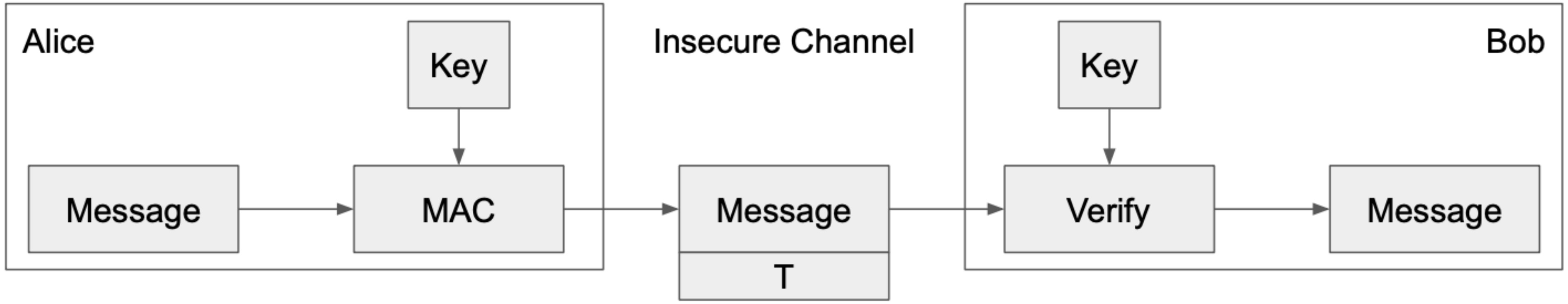
Message authentication code (MAC)

- generated by an algorithm that creates a small fixed-sized block
  - depending on both message and some key
  - not be reversible
  - $MAC_M = F(K_{AB}, M)$
- appended to message as a signature
- receiver performs same computation on message and checks it matches the MAC
- provides assurance that message is unaltered and comes from sender

MACs: Usage

- Alice wants to send *M* to Bob, but doesn't want David to tamper with it
- Alice sends *M* and *T* = MAC(*K*, *M*) to Bob
- Bob receives *M* and *T*
- Bob computes MAC(*K*, *M*) and checks that it matches *T*
- If the MACs match, Bob is confident the message has not been tampered with (integrity)



MACs: Definition

- Two parts:
  - KeyGen() → *K*: Generate a key *K*
  - MAC(*K*, *M*) → *T*: Generate a tag *T* for the message *M* using key *K*
    - Inputs: A secret key and an arbitrary-length message
    - Output: A fixed-length **tag** on the message
- Properties
  - **Correctness**: Determinism
    - Note: Some more complicated MAC schemes have an additional Verify(*K*, *M*, *T*) function that don't require determinism, but this is out of scope
  - **Efficiency**: Computing a MAC should be efficient
  - **Security**: existentially unforgeable under chosen plaintext attack

Existentially unforgeable

- A secure MAC is **existentially unforgeable**: without the key, an attacker cannot create a valid tag on a message
  - David cannot generate MAC(K, M') without K
  - David cannot find any $M' \neq M$ such that MAC(K, M') = MAC(K, M)

Example: HMAC

- issued as RFC 2104 [1]
- has been chosen as the mandatory-to-implement MAC for IP Security
- Used in Transport Layer Security (TLS) and Secure Electronic Transaction (SET)

[1] “HMAC: Keyed-Hashing for Message Authentication”, RFC 2104, https://datatracker.ietf.org/doc/html/rfc2104

HMAC(K, M)

- will produce two keys to increase security
- If key is longer than the desired size, we can hash it first, but be careful with using keys that are too much smaller, they have to have enough randomness in them
- Output $H[(K^+ \oplus opad) \,||\, H[(K^+ \oplus ipad) \,||\, M]]$

Example: HMAC

- HMAC(K, M):
  - Output $H[(K^+ \oplus opad) \mid\mid H[(K^+ \oplus ipad) \mid\mid M]]$
- Use K to derive two different keys
  - *opad* (outer pad) is the hard-coded byte **0x5c** repeated until it’s the same length as K^+
  - *ipad* (inner pad) is the hard-coded byte **0x36** repeated until it’s the same length as K^+
  - As long as *opad* and *ipad* are different, you’ll get two different keys
  - For paranoia, the designers chose two very different bit patterns, even though they theoretically need only differ in one bit

HMAC

$H[(K^{+} \oplus opad) \,||\, H[(K^{+} \oplus ipad) \,||\, M]]$

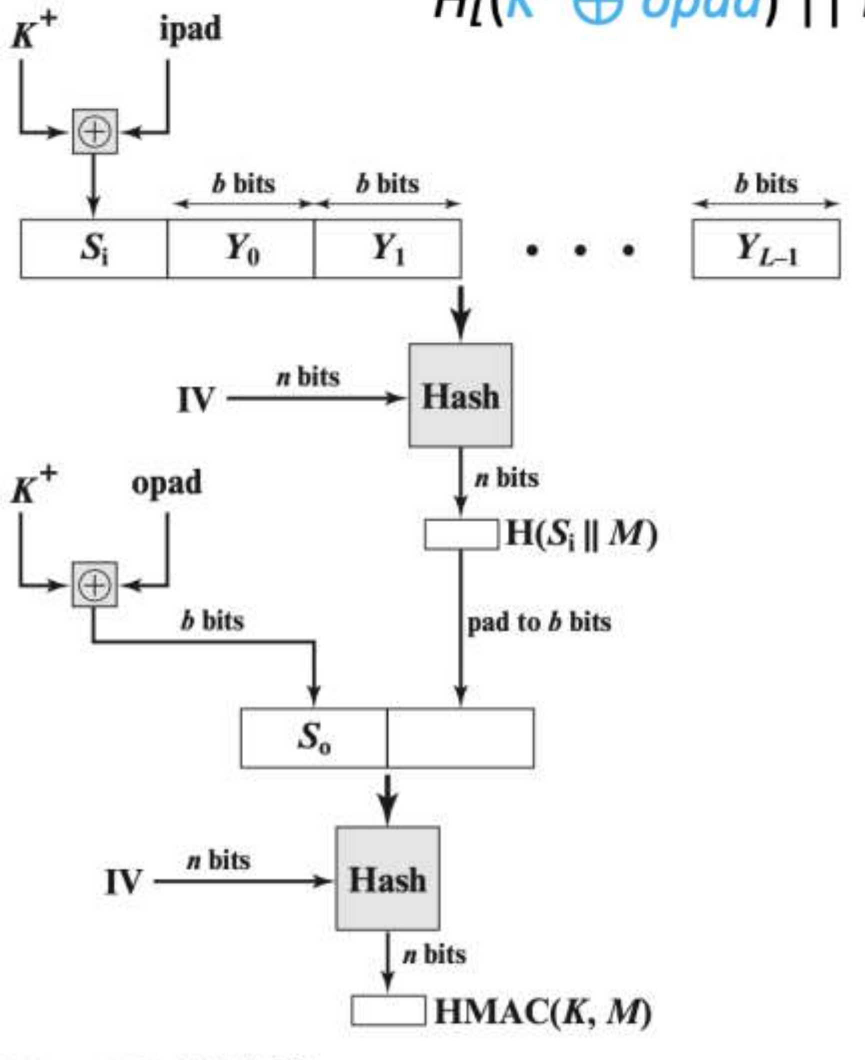


Figure 3.6 HMAC Structure

A	B	A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

$$K^{+} = \begin{cases} H(K) & K \text{ is larger than block size} \\ K & \text{otherwise} \end{cases}$$

ipad = 00110110 , repeat b/8 times
opad = 01011100, repeat b/8 times

HMAC procedure

$$H[(K^+ \oplus opad) \,||\, H[(K^+ \oplus ipad) \,||\, M]]$$

- Step 1: Append zeros to the left end of *K* to create a *b*-bit string K^+ (e.g., if *K* is of length 160 bits and *b* = 512, then *K* will be appended with 44 zero bytes);
- Step 2: XOR (bitwise exclusive-OR) K^+ with ipad to produce the *b*-bit block S_i;
- Step 3: Append *M* to S_i;
- Step 4: Apply H to the stream generated in step 3;
- Step 5: XOR K^+ with opad to produce the *b*-bit block S_o;
- Step 6: Append the hash result from step 4 to S_o;
- Step 7: Apply H to the stream generated in step 6 and output the result.