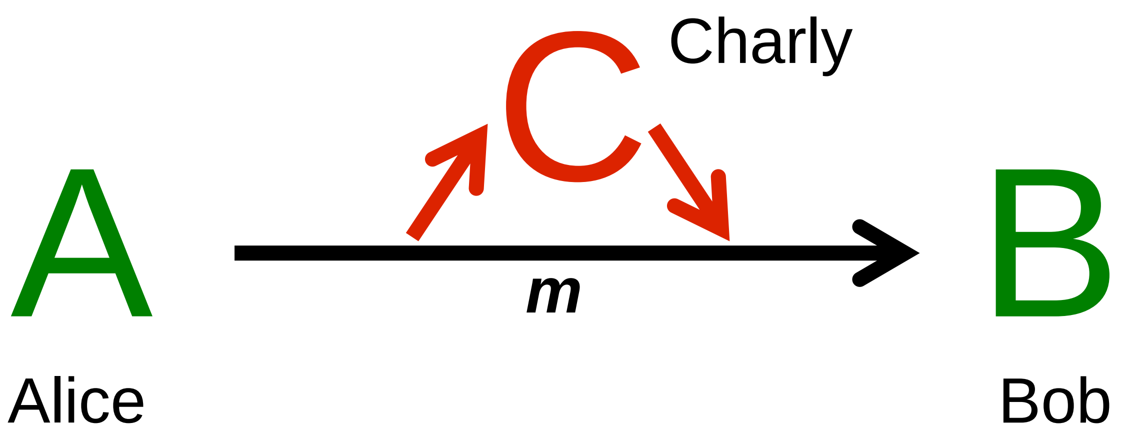
Network Security

Cryptography

Prof. Dr. Stefan Heiss

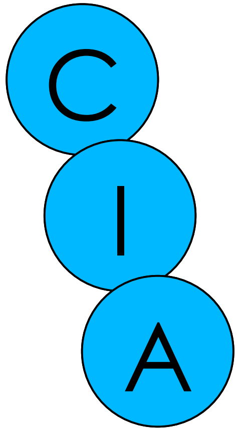
TECHNISCHE HOCHSCHULE OSTWESTFALEN-LIPPE
UNIVERSITY OF APPLIED SCIENCES AND ARTS

- **Eavesdropping, Sniffing**
- **Impersonation, Spoofing, Unauthorized Access**
- **Replaying attacks**
- **Denial of Services (DoS)**
- **Misuse of resources**



IT-Security: Aims

- **Confidentiality**
- **Integrity**
- **Availability**
- **Authenticity**
- **Non-Repudiation**
- **Access Control**



IT-Security: Mechanisms

Confidentiality	Encryption
Integrity	Message Digest, MAC, Digital Signature
Availability	Network Filter, Firewall, Robust Impl.
Authenticity	MAC, Key (physical token), Biometric identification
Non-Repudiation	Digital Signature
Access Control	Secure Configurations, Best Security Practices, Security awareness of users, Policies

Cryptographic Algorithms

- Cryptographic secure Pseudo Random Number Generators (PRNGs)
- Message Digests (Cryptographic Hash Functions)

- Symmetric Ciphers
- MACs (Message Authentication Codes)

- Asymmetric Ciphers
- Digital Signatures
- Key derivation algorithms / schemes

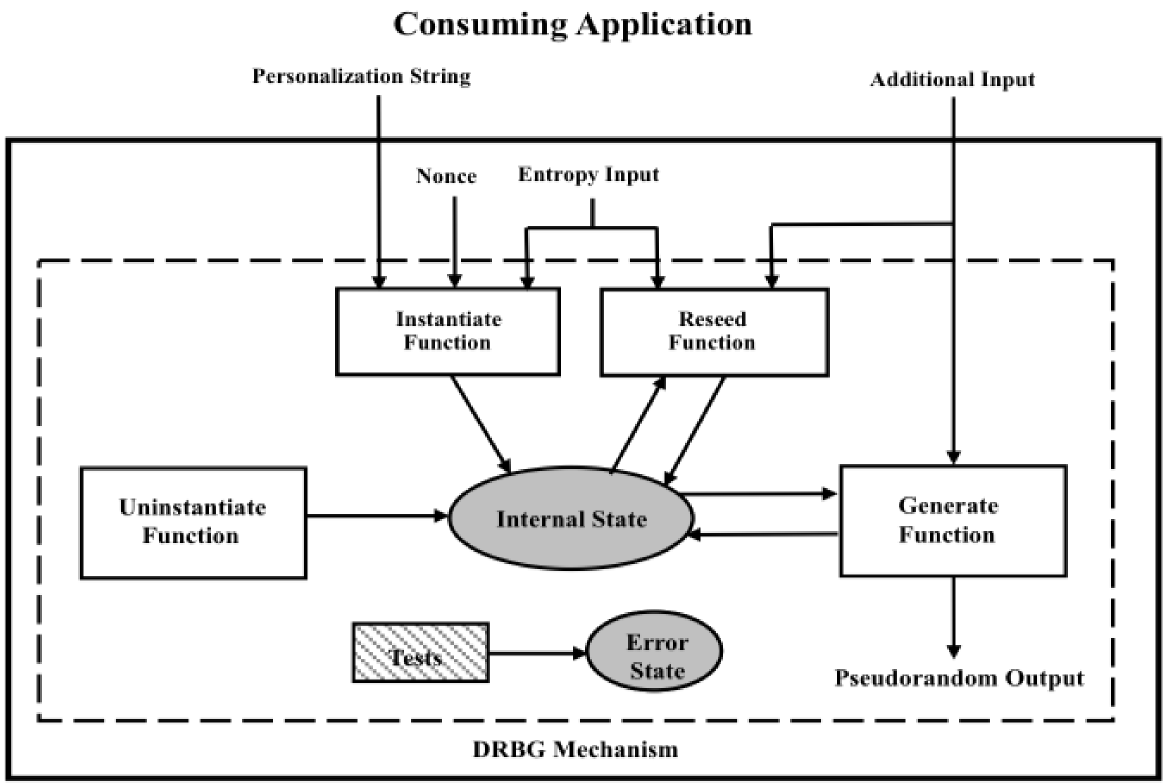
- **Kerckhoffs von Nieuwenhof (1835-1903):**
  - The security of a cryptographic algorithm should not depend on its nondisclosure.
  - Today's best practice: Only use and implement well-known algorithms that have been thoroughly investigated by the community of international distinguished cryptographers. (E.g.: Contest for election of AES)
  - Do not rely on “Security by obscurity” !

Cryptographic Secure PRNGs

- JCA implementations: **SecureRandom**
- Recommendation for Random Number Generation Using Deterministic Random Bit Generators, NIST Special Publication 800-90A, June 2015

  http://dx.doi.org/10.6028/NIST.SP.800-90Ar1

  - DRBG Functional Model:



SecureRandom_PerformanceDemo

```
public class SecureRandom_PerformanceDemo {

  static Random rng = new SecureRandom();
  static byte[] b = new byte[1];

  public static void main(String[] args) throws Exception {
    for( int i = 0; i < 100; i++ ) {
      long t1 = System.nanoTime();
      rng.nextBytes(b);
      rng.nextLong();
      System.out.println(System.nanoTime() - t1);
    }
  }
}
```

Standard Random() implementations are NO CSPRNGs !

- Example: Java‘s [Random]() class implements the PRNG based on the following linear congruential formula:

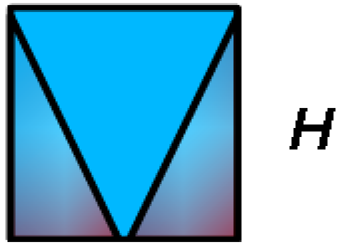
```java
public static long nextSeed(long seed) {
  return (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
}
```

  - $x_{n+1} = (25214903917 \cdot x_n + 11) \mod 2^{48}$
  - (Only the bits from (int)(seed >>> 16) are return via the Random API.)

```java
public static void findNextIntValue(long r1, long r2) {
  long seed = (r1 << 16);
  while( (nextSeed(seed) >>> 16) != r2 ) {
    ++seed;
  }
  System.out.println("Next value: " +
   Long.toHexString( nextSeed(nextSeed(seed)) >>> 16) );
}
```

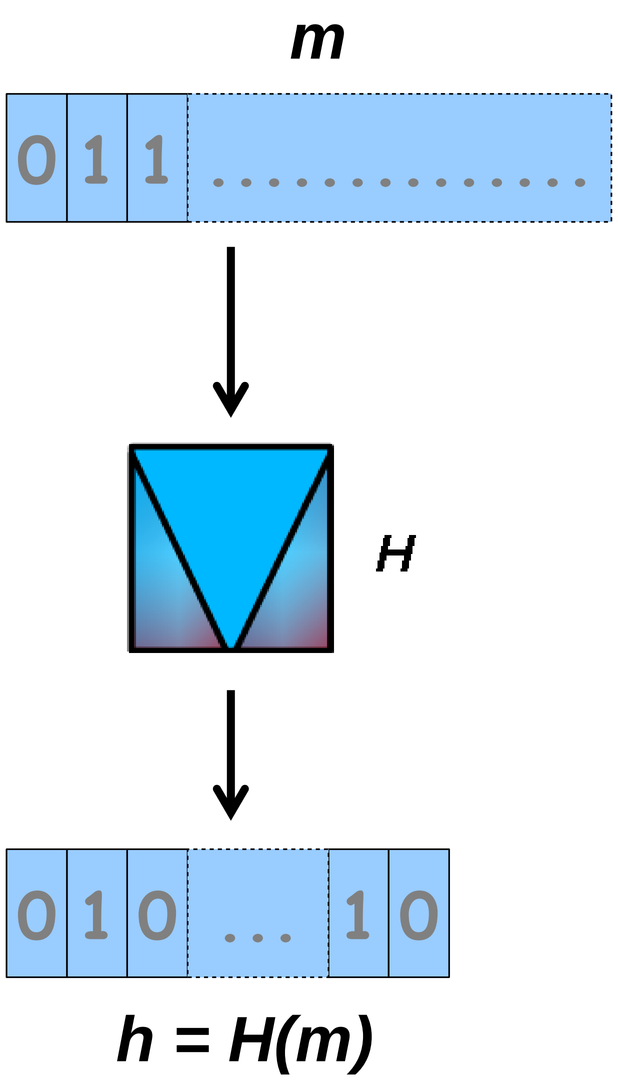
Message Digests

- ***Cryptographic Hash Functions***
- ***Digital Fingerprints***
- [JCA](#) implementations: [MessageDigest](#)



A Message Digest (Cryptographic Hash Function H is a mapping of the set of all binary sequences of finite length $m = (m_1, m_2, m_3, \ldots)$ to the set of binary sequences of some fixed length n:

$$H(m) = (h_1, h_2, \ldots, h_n) \in (F_2)^n$$

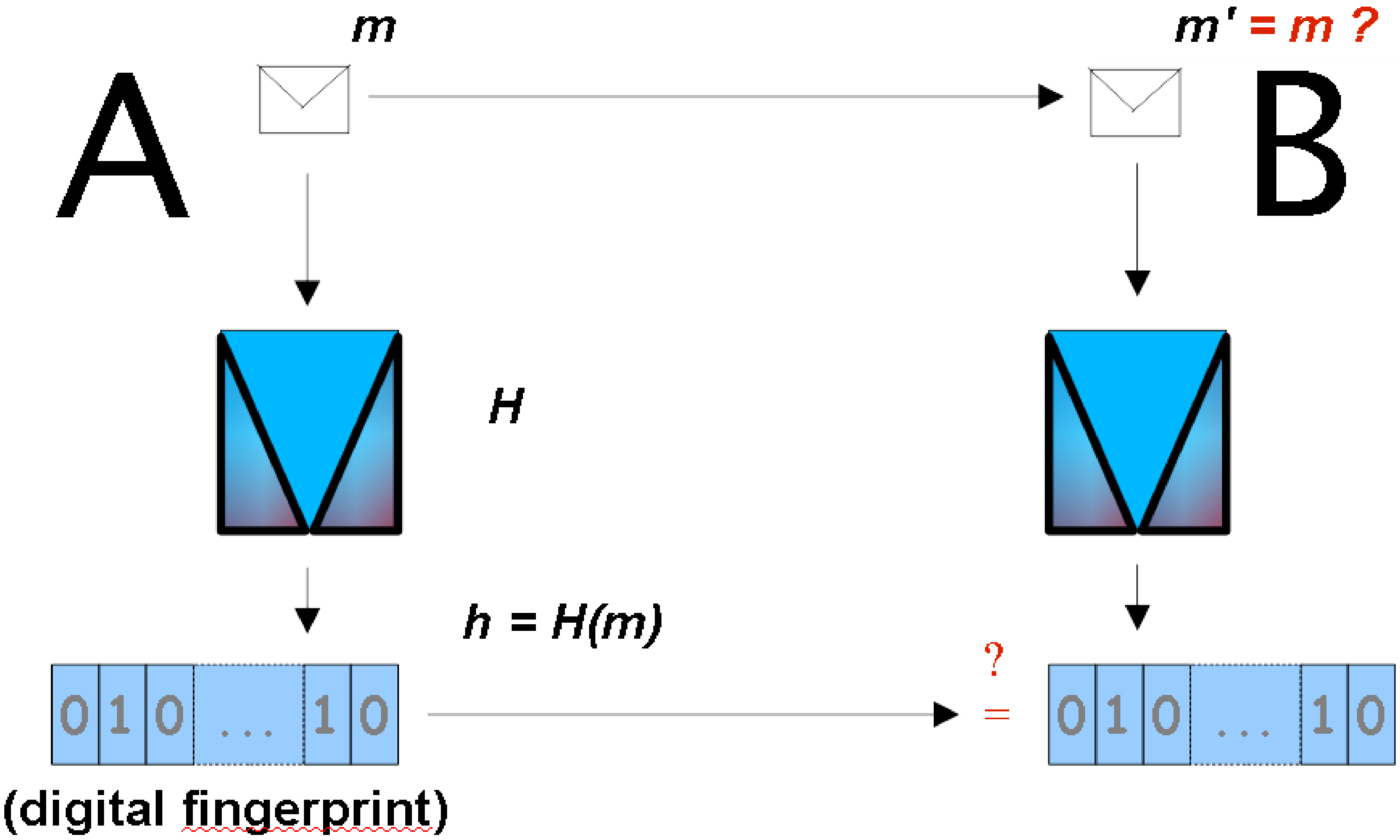


- **Preimage resistence**

  Given a sequence $(h_1,h_2,\ldots,h_n) \in (F_2)^n$, it is practically impossible to find a sequence $(s_1,s_2,s_3,\ldots)$ with $H(s_1,s_2,s_3,\ldots) = (h_1,h_2,\ldots,h_n)$.

- **Collision resistence**

  It is practically not possible to find two sequences $(s_1,s_2,s_3,\ldots)$ and $(t_1,t_2,t_3,\ldots)$ with $H(s_1,s_2,s_3,\ldots) = H(t_1,t_2,t_3,\ldots)$.


m
A
m ' = m ?
B
H
h = H(m)
0 1 0 ... 1 0
?
=
0 1 0 ... 1 0
(digital fingerprint)

- **Integrity checks**

  Example: Check of MD5 message digest after some file download

- **Protection of secrets**

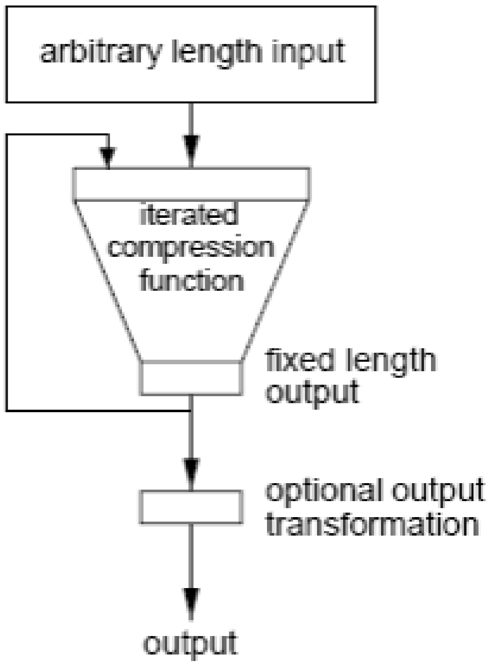
  Example: Password files

- **Construction of PRNGs and stream ciphers**

- **Construction of MAC's (keyed hash)**

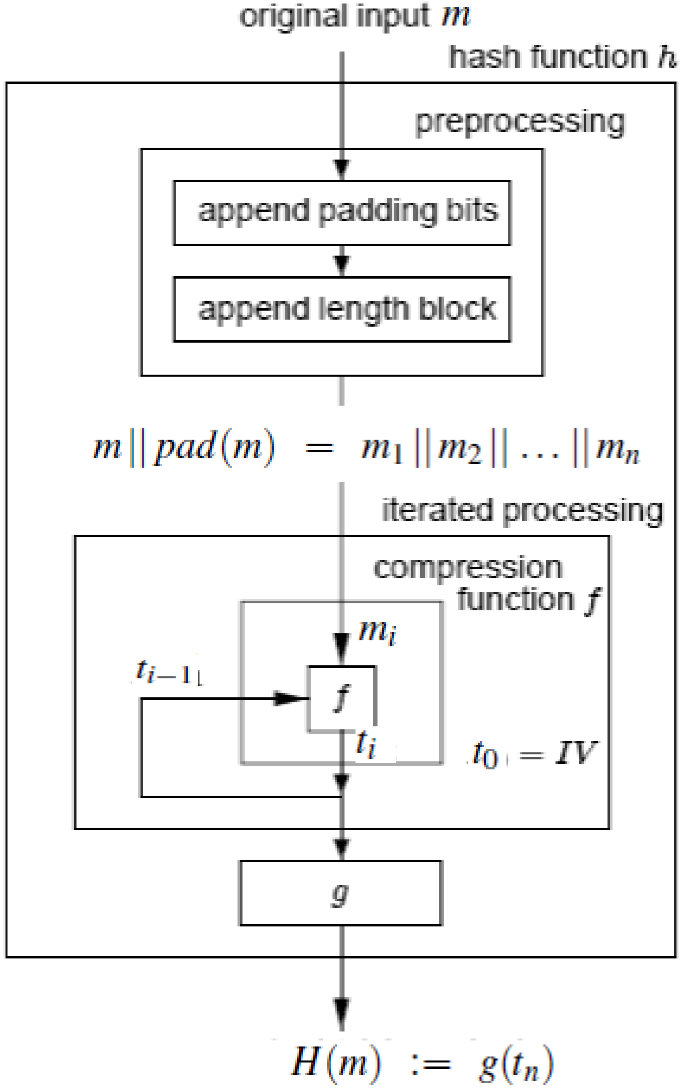
Iterated Hash Functions – General Construction

(a) high-level view



See: Handbook of Applied Cryptography, Chapter 9

(b) detailed view



$l(m_1) = \cdots = l(m_n) = r$

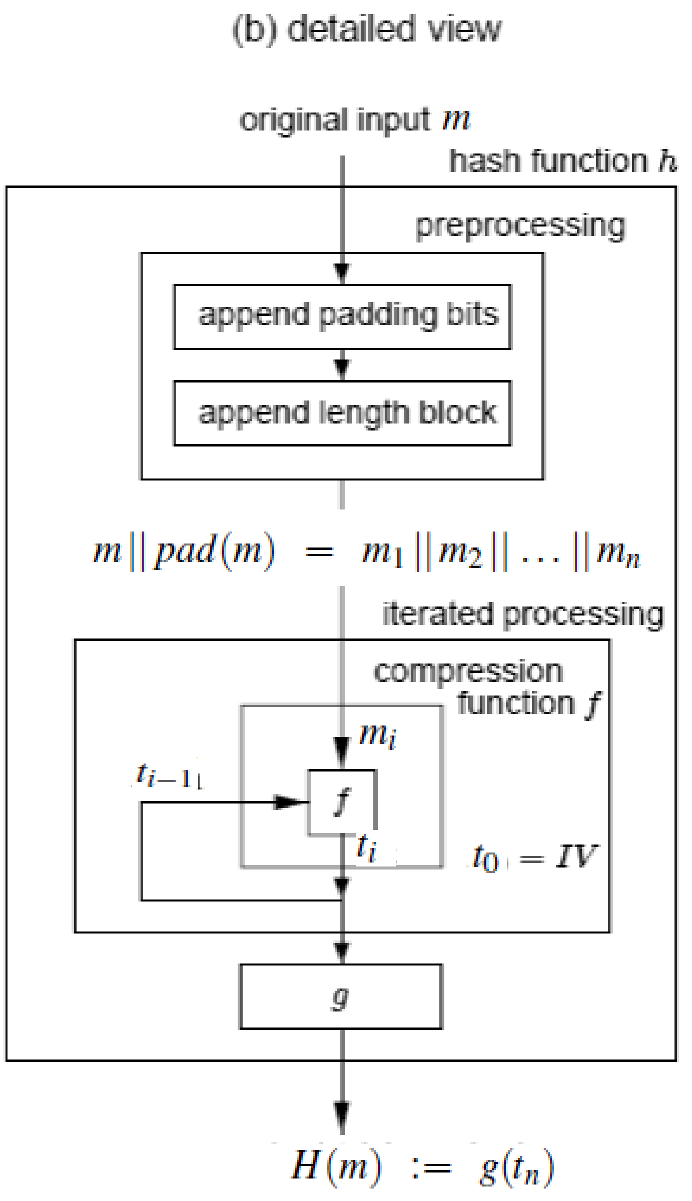
$f : \mathbb{Z}_2^{r+s} \to \mathbb{Z}_2^{s}$

$t_i := f(m_i || t_{i-1})$

$g : \mathbb{Z}_2^{s} \to \mathbb{Z}_2^{l}$

Hash Functions – Important Examples

H	$l = l(H(m))$	$r = l(m_i)$	$s = l(t_i)$	$min\{l(pad)\}$
MD5	128	512	128	65
SHA-1	160	512	160	65
SHA-224	224	512	256	65
SHA-256	256	512	256	65
SHA-512/224	224	1024	512	129
SHA-512/256	256	1024	512	129
SHA-384	384	1024	512	129
SHA-512	512	1024	512	129
SHA3-224	224	1152	1600	4
SHA3-256	256	1088	1600	4
SHA3-384	384	832	1600	4
SHA3-512	512	576	1600	4



$l(m_1) = \dots = l(m_n) = r$

$f : \mathbb{Z}_2{}^{r+s} \to \mathbb{Z}_2{}^{s}$

$t_i := f(m_i||t_{i-1})$

$g : \mathbb{Z}_2{}^{s} \to \mathbb{Z}_2{}^{l}$

- **Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD**

  Xiaoyun Wang, Dengguo Feng, Xuejia Lai, Hongbo Yu, August 2004

  http://eprint.iacr.org/2004/199.pdf

- **The first collision for full SHA-1**

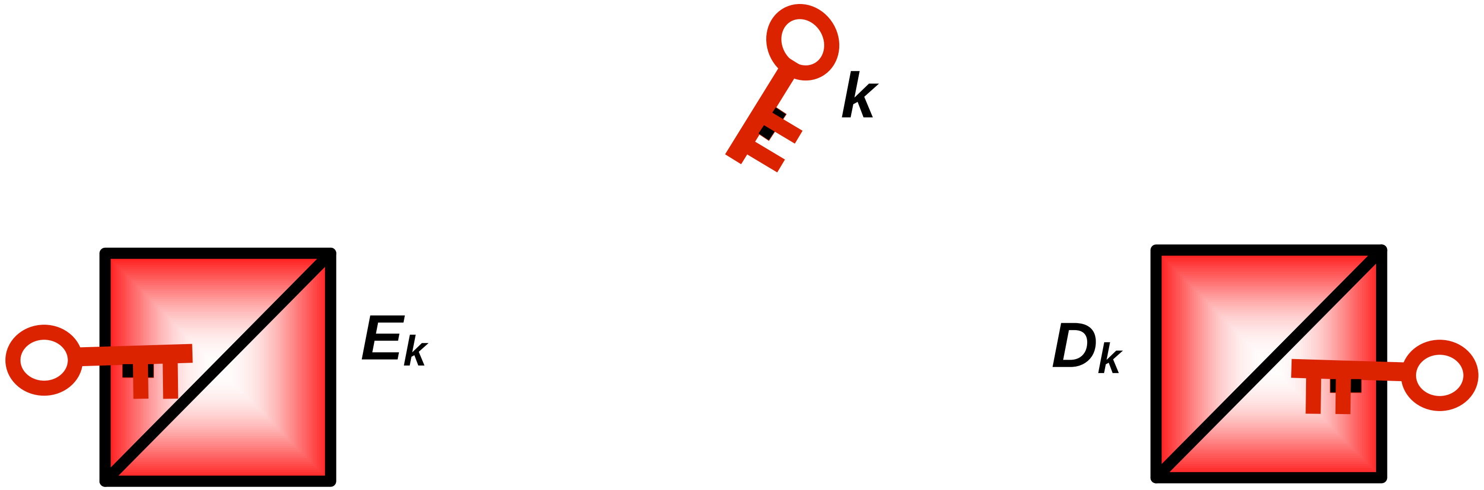
  Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, Yarik Markov, 2017
  https://shattered.io/

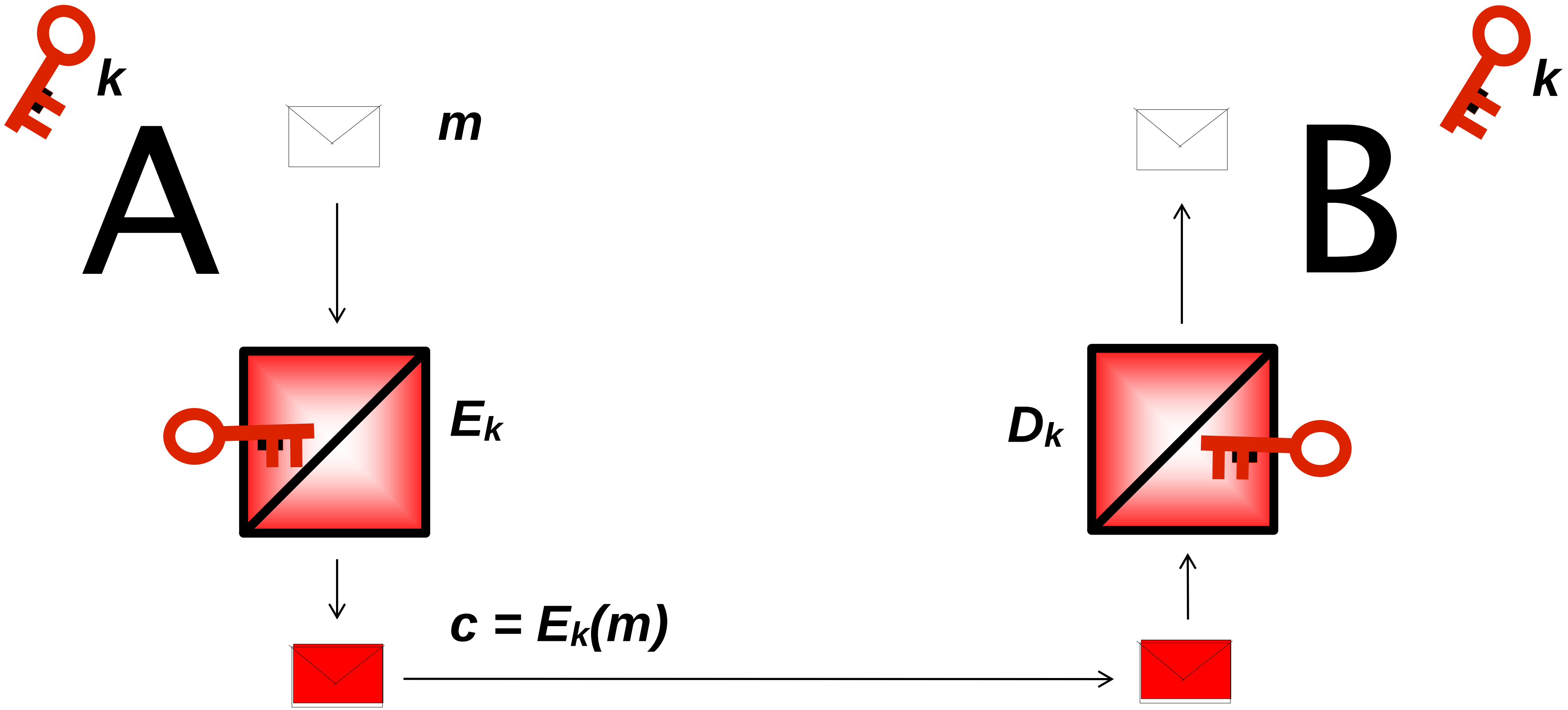
MessageDigest_Demo

```java
public class MessageDigest_Demo {

  public static void main(String[] args) throws Exception {

    FileInputStream fis
     = new FileInputStream("shattered-1.pdf");
    byte[] m = fis.readAllBytes();

    MessageDigest md = MessageDigest.getInstance("SHA-1");
    byte[] hashValue = md.digest(m);

//    System.out.println("Data:");
//    System.out.println(Dump.dump(m));

//    System.out.println("Hash value of shattered-1.pdf:");
//    System.out.println(Dump.dump(hashValue));
  }
}
```

Symmetric Ciphers

- JCA implementations: Cipher




k
A
m
E_k
$c = E_k(m)$
D_k
B
k

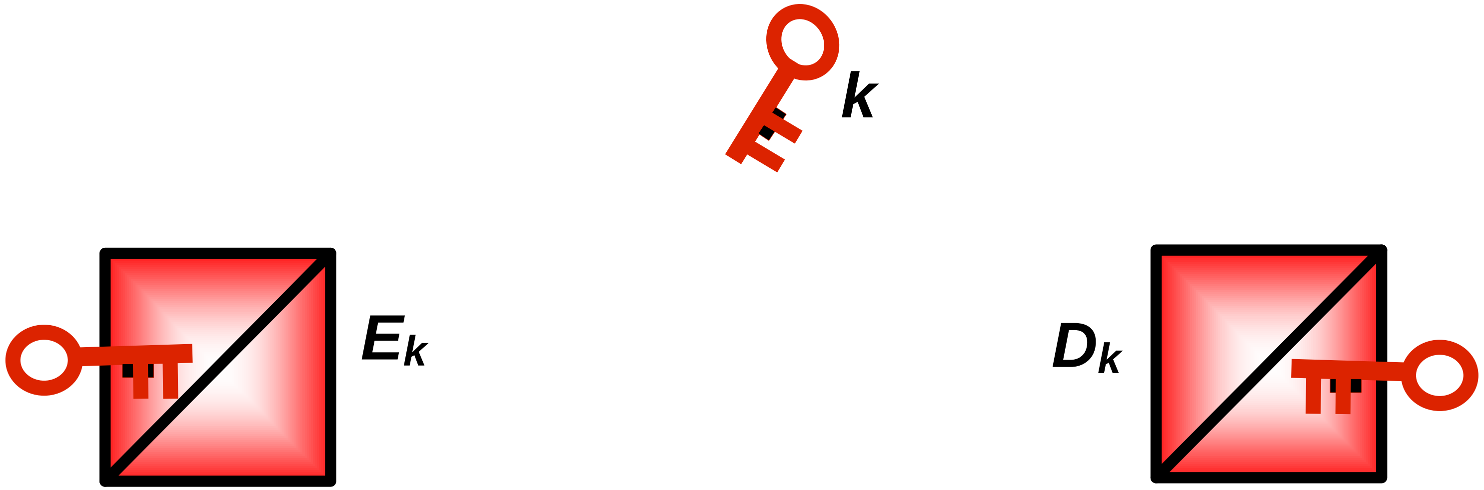
- **Key exchange:**
  - Alice and Bob must share a secret key, which has to be exchanged over a secure channel, before it can be used to initialize an encryption algorithm to encrypt messages.

- **Key storage:**
  - Keys have to be securely managed and stored.

- **Aim of constructions of cipher algorithms:**
  - No attack has a better performance than a *Brute Force attack*.
  - This means: The size of the key space $|K|$ (number of possible keys) is directly proportional to the security of the algorithm.

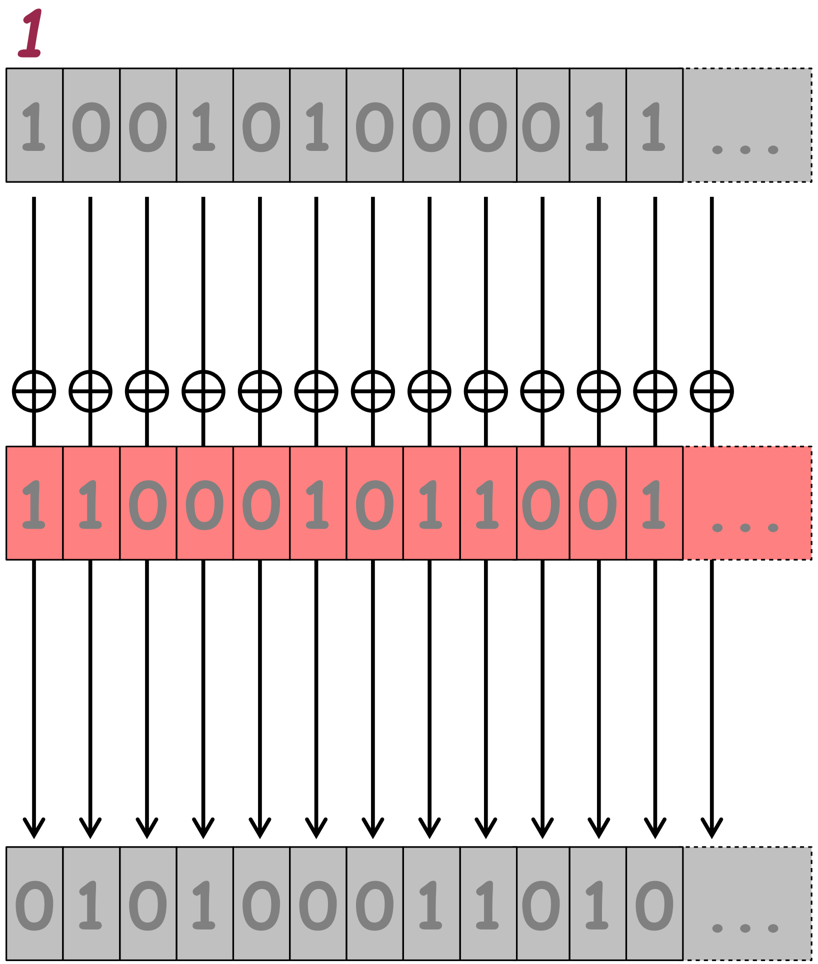
- **Stream ciphers**

- **Block ciphers**
- Modes of operation:
  - ECB (Electronic Codebook Modus)
  - CBC (Cipher Block Chaining Modus)
  - CFB (Cipher Feedback Modus)
  - OFB (Output Feedback Modus)
  - ...



k
E_k
D_k

One Time Pad (OTP)



<u>Plaintext</u> (Bitstream)

<u>One Time Pad</u> *k* (generated by a real random process)

<u>Ciphertext</u>

One Time Pad (OTP) - Pros

- A truely randomly generated one time pad is the only cipher that guarantees absolut (provable) security.
- The only information that can be deduced from eavesdropping is the length of the plaintext.

One Time Pad (OTP) - Cons

- **Key establishment**

  The one time pad has to be exchanged over some other secure channel prior to its use.

- **Key length**

  The one time pad (key) has to be as long as the plaintext.

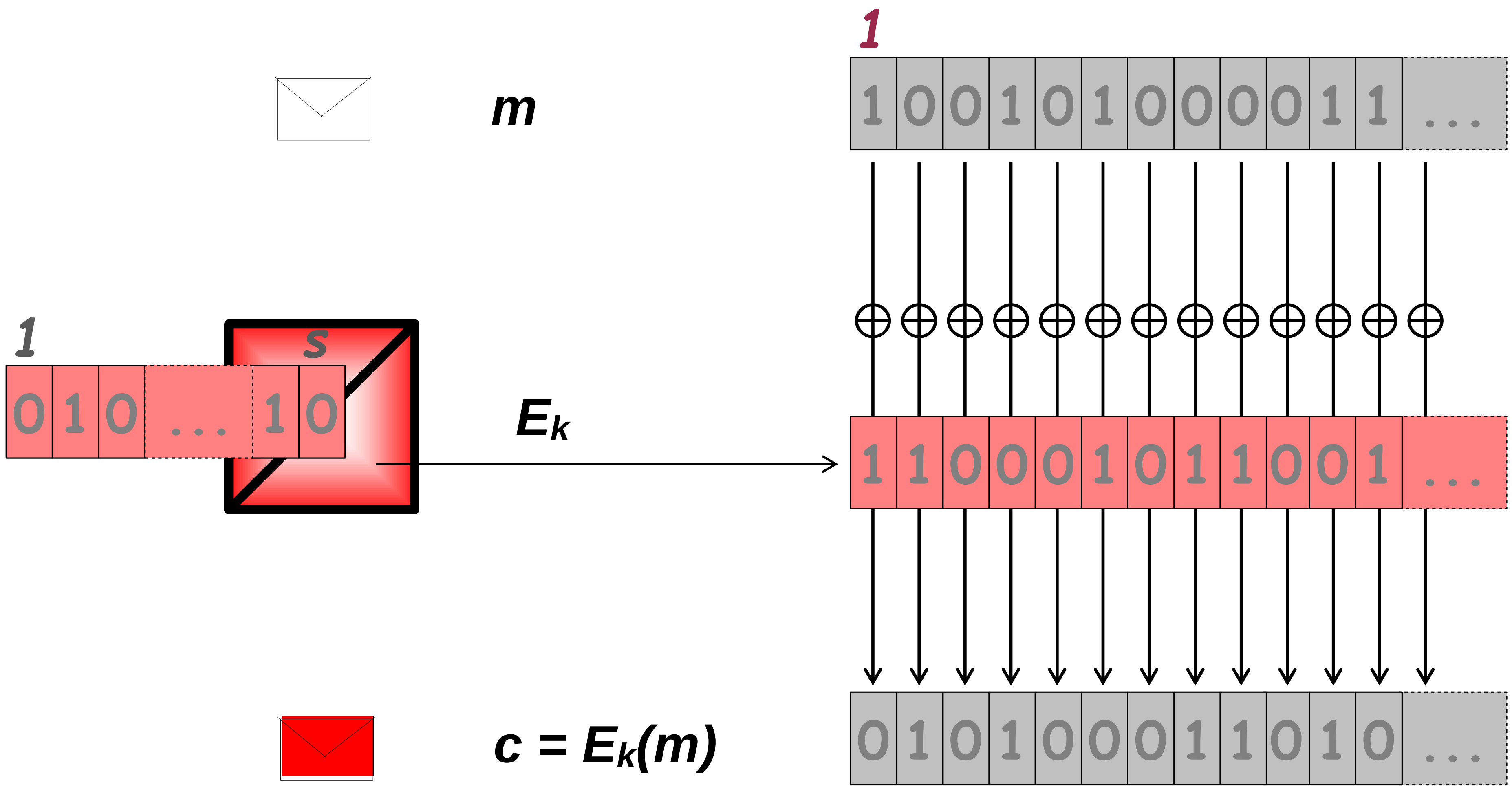
- **Reusability**

  Reusage of a one time pad is strictly prohibited, as it would allow an attack by statistical analysis.

- **Key generation**

  Costly, as a real physicaly random process has to be used.

Additive Synchronous Stream Ciphers



Additive Synchronous Stream Ciphers – Pros

- The keystream is independent of the plaintext. (Keystream can be precalculated.)
- Encryption is a simple (fast) XOR operation.
- Decryption = Encryption

Additive Synchronous Stream Ciphers – Cons

- **Key establishment**

  The key has to be exchanged over some other secure channel prior to its use.

- **Reusability**

  Reusage of the same key is strictly prohibited, as it compromises the encryption scheme.

- **Integrity**

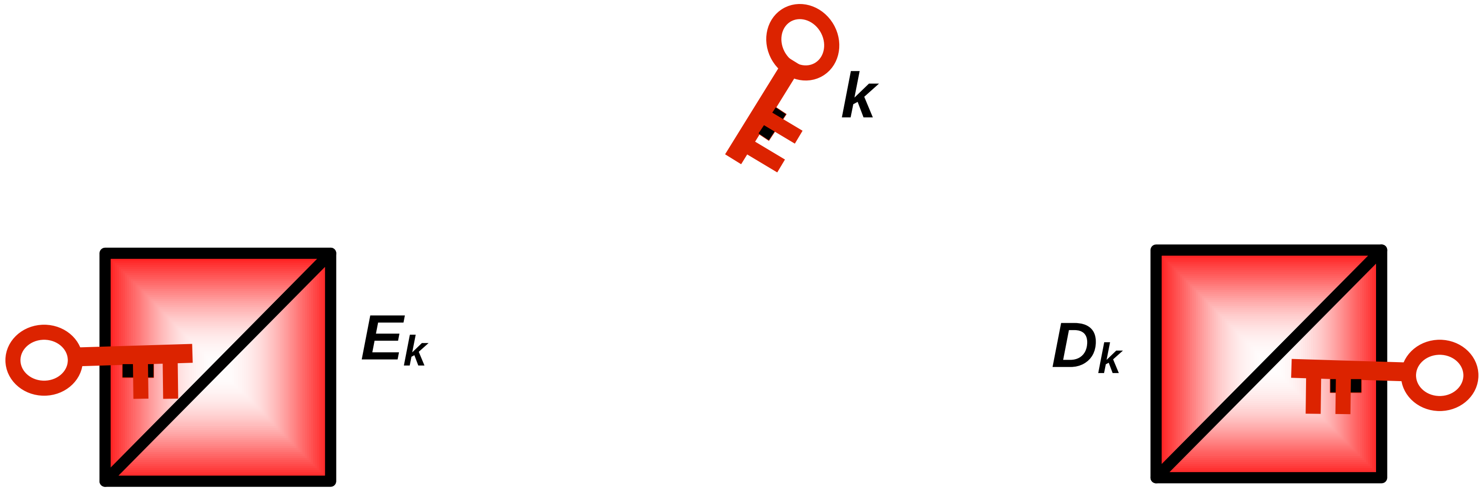
  Integrity is not protected: Single bits can be switched by an attacker.

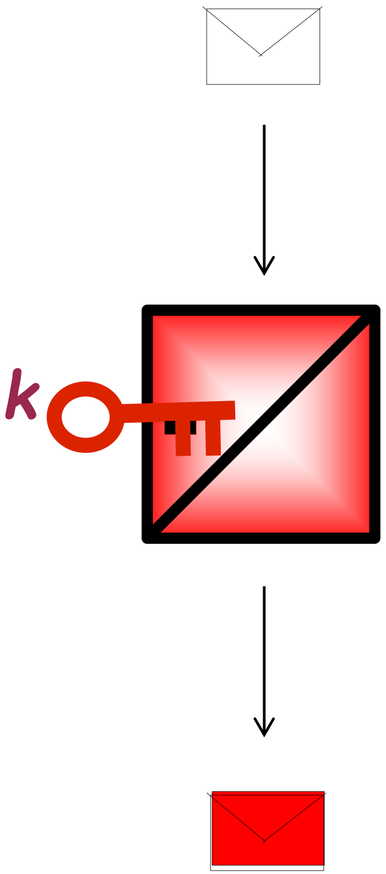
ChaCha20

- Additive Synchronous Stream Cipher specified in:
  - **RFC 8439 - ChaCha20 and Poly1305 for IETF Protocols**
- Currently the only alternative to the AES cipher defined for record layer protocol encryption in TLS 1.3 (used in combination with the Poly1305 authenticator). See RFC 8446, B.4. Cipher Suites).

```java
Cipher cipher = Cipher.getInstance("ChaCha20");

byte[] key = new byte[32];
for( byte i = 0; i < 32; ++i ) {
  key[i] = i;
}
SecretKeySpec keyChaCha20 = new SecretKeySpec(key, "ChaCha20");

byte[] nonce
 = new byte[]{0, 0, 0, 0, 0, 0, 0, 0x4a, 0, 0, 0, 0};
AlgorithmParameterSpec params
 = new ChaCha20ParameterSpec(nonce, 1);
cipher.init(Cipher.ENCRYPT_MODE, keyChaCha20, params);

byte[] m = ("Ladies and Gentlemen of the class of '99:"
 + " If I could offer you only one tip for the future,"
 + " sunscreen would be it.").getBytes();
byte[] c = cipher.doFinal(m);
```

Symmetric Ciphers – Block Ciphers



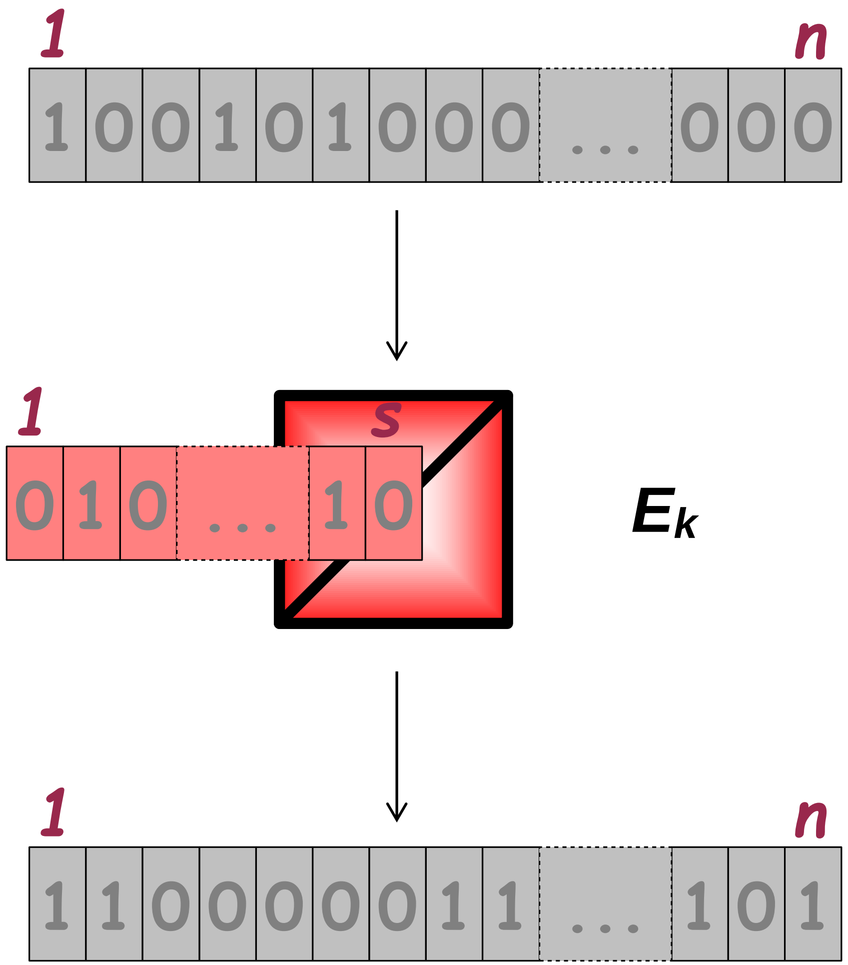


m — n bit plaintext (block length n)

E_k — $k \in K$ (key space K)

$c = E_k(m)$ — n bit cipher text

Block Ciphers



$m \in (F_2)^n$ **(block)**

$k \in K = (F_2)^s$

$E : K \times (F_2)^n \rightarrow (F_2)^n$

$E_k : (F_2)^n \rightarrow (F_2)^n$

$c \in (F_2)^n$ **(cipher block)**

- How many different block ciphers can be defined for the encryption of blocks of length *n* ?

- Why not use random permutations of the set of all 2^n blocks of length *n* for the construction of block ciphers?

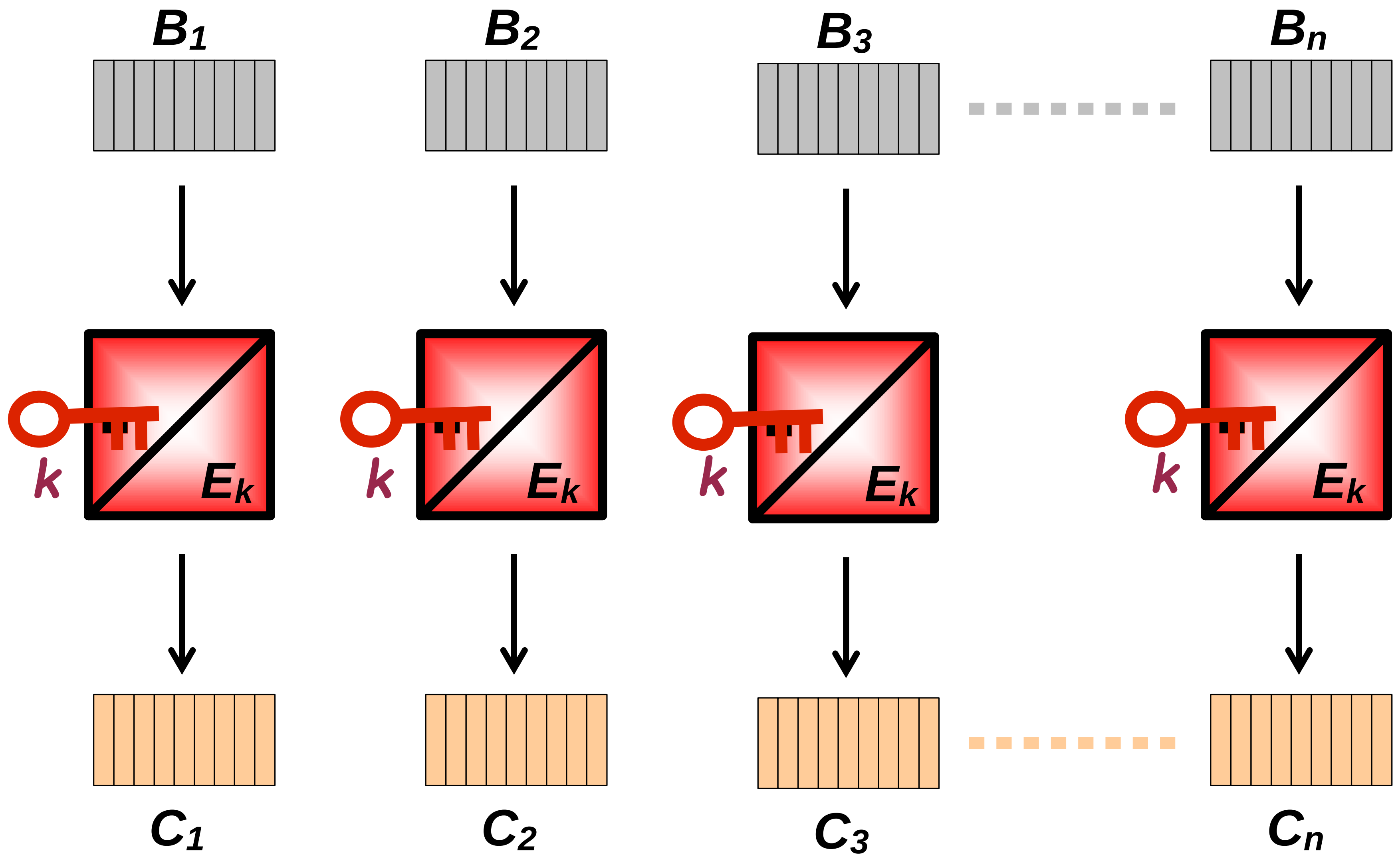
- A block cipher shall exhibit the same statistical features as a random permutation of all 2^n blocks of length n.
- Encryption and Decryption shall be efficiently implementable in SW and HW (runtime performance, memory requirements).
- Block ciphers are usually organized in rounds, where the following types of basic operations are repeatedly executed:
  - Permutations of the bits of a block.
  - Substitutions (S-Boxes) of values in subblocks.

- **AES (Advanced Encryption Standard)**
  - Specified key lengths: 128, 192, 256 bit
  - Block length: 128 bit
  - Winning algorithm (Rijndal algorithm) from an international contest (organised by NIST).
  - US Federal Standard FIPS PUB 197, published 2001.
  - Nice animation available in Cryptool1
    - https://www.cryptool.org/en/ct1
    - CryptTool -> Indiv. Procedures -> Visualisation of Algorithms -> AES -> Rijndale Animation -> Steuerung -> Abspielen

- The length of plaintext data must be a multiple of *n*.
  - Padding operations are needed.
- Simply encrypting data block by block (ECB modus) may allow dictionary attacks. To prevent such attacks, use:
  - CBC modus
  - Random IV values

ECB	Electronic Code Book
CBC	Cipher Block Chaining
IV	Initialization Vector

ECB mode (Electronic Code Book)



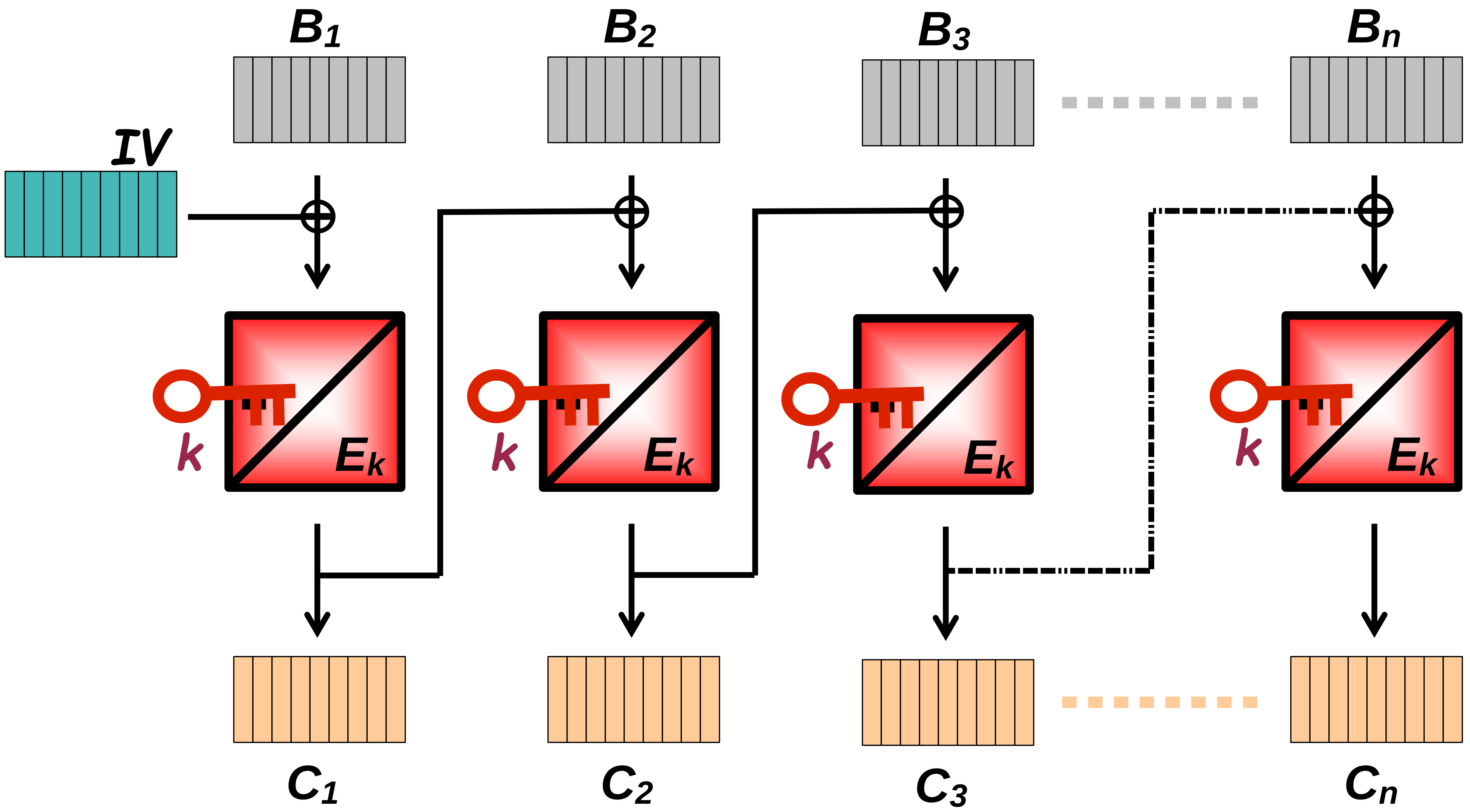
Cipher_ECB_Demo

```java
byte[] key = Dump.hexString2byteArray(
 "0102030405060708090A0B0C0D0E0F10");
SecretKey secretKey = new SecretKeySpec(key, "AES");

Cipher cipher
 = Cipher.getInstance("AES/ECB/NOPADDING");

cipher.init(Cipher.ENCRYPT_MODE, secretKey);

byte[] m = new byte[48];
```

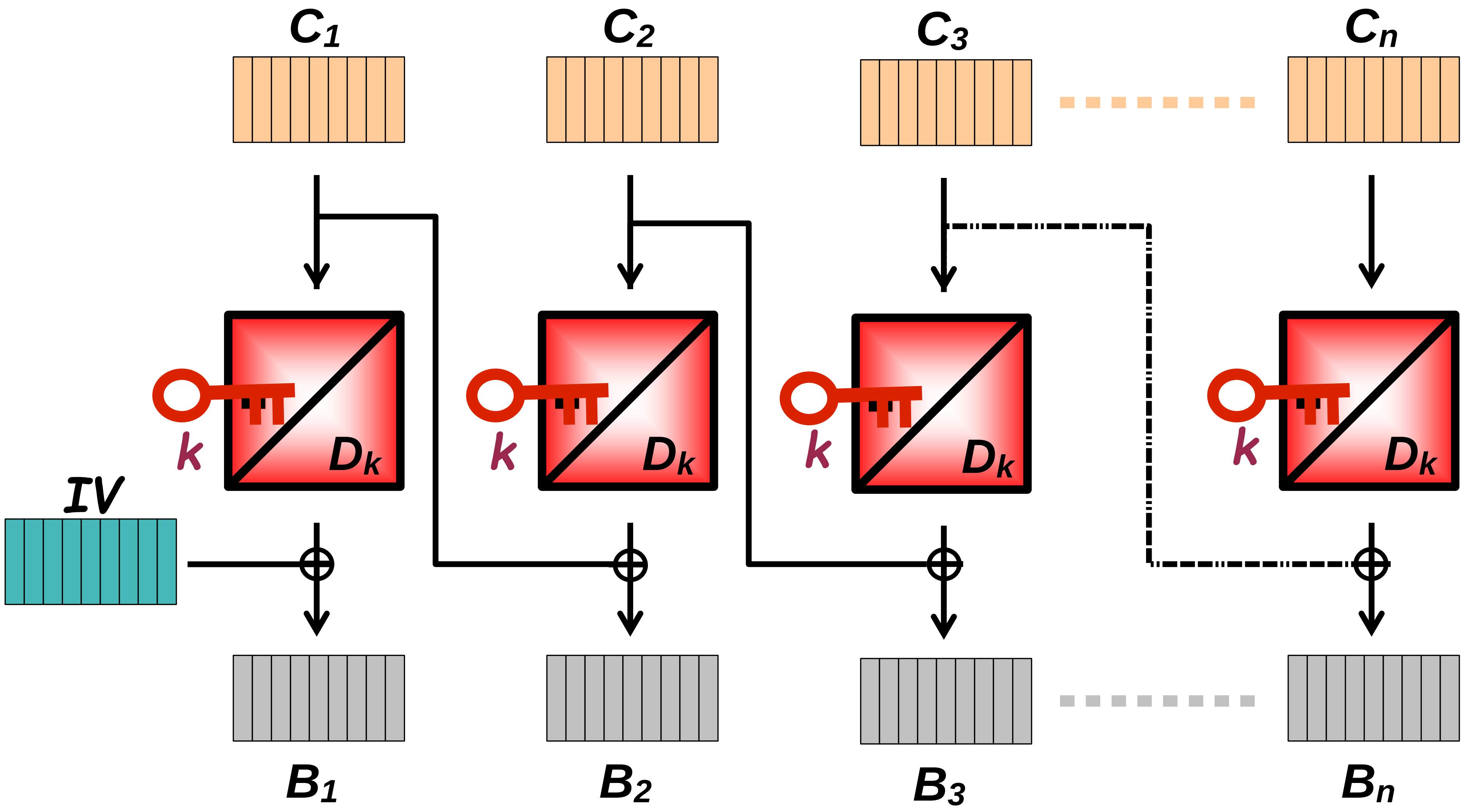
```java
byte[] c = cipher.doFinal(m);

System.out.println("Ciphertext:");
System.out.println(Dump.dump(c));
```

```java
cipher.init(Cipher.DECRYPT_MODE, secretKey);
byte[] m2 = cipher.doFinal(c);
System.out.println("Decrypted Ciphertext:");
System.out.println(Dump.dump(m2));
```

CBC mode (Cipher Block Chaining) - Encryption



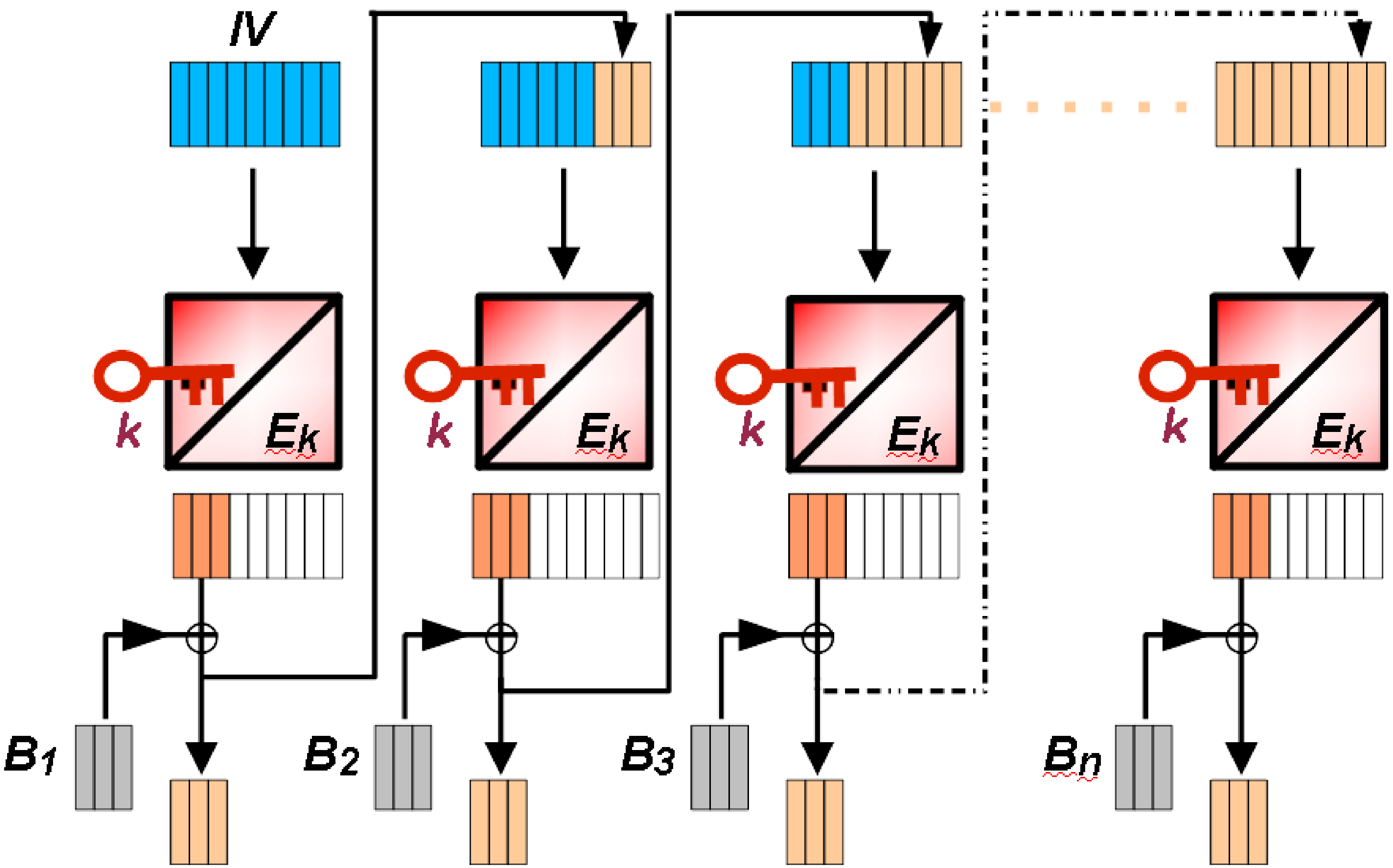
CBC mode (Cipher Block Chaining) - Decryption



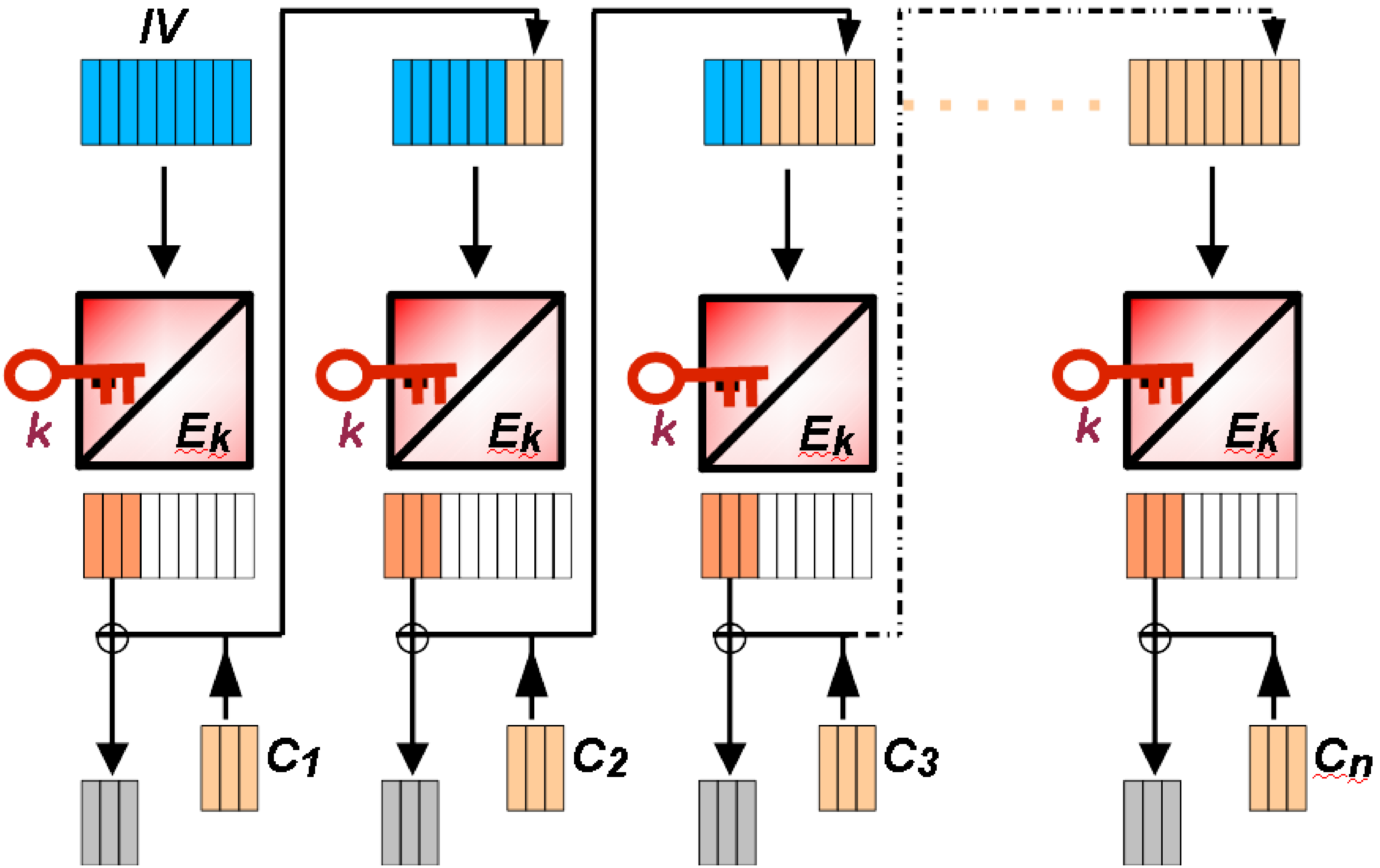
Cipher_CBC_Demo

```java
byte[] key = Dump.hexString2byteArray(
 "0102030405060708090A0B0C0D0E0F10");
SecretKey secretKey = new SecretKeySpec(key, "AES");

Cipher cipher
 = Cipher.getInstance("AES/CBC/PKCS5PADDING");

byte[] ivBytes = new byte[16];
new SecureRandom().nextBytes(ivBytes);
IvParameterSpec iv = new IvParameterSpec(ivBytes);
cipher.init(Cipher.ENCRYPT_MODE, secretKey, iv);

byte[] m = new byte[48];
```

CFB mode (Cipher Feedback) - Encryption

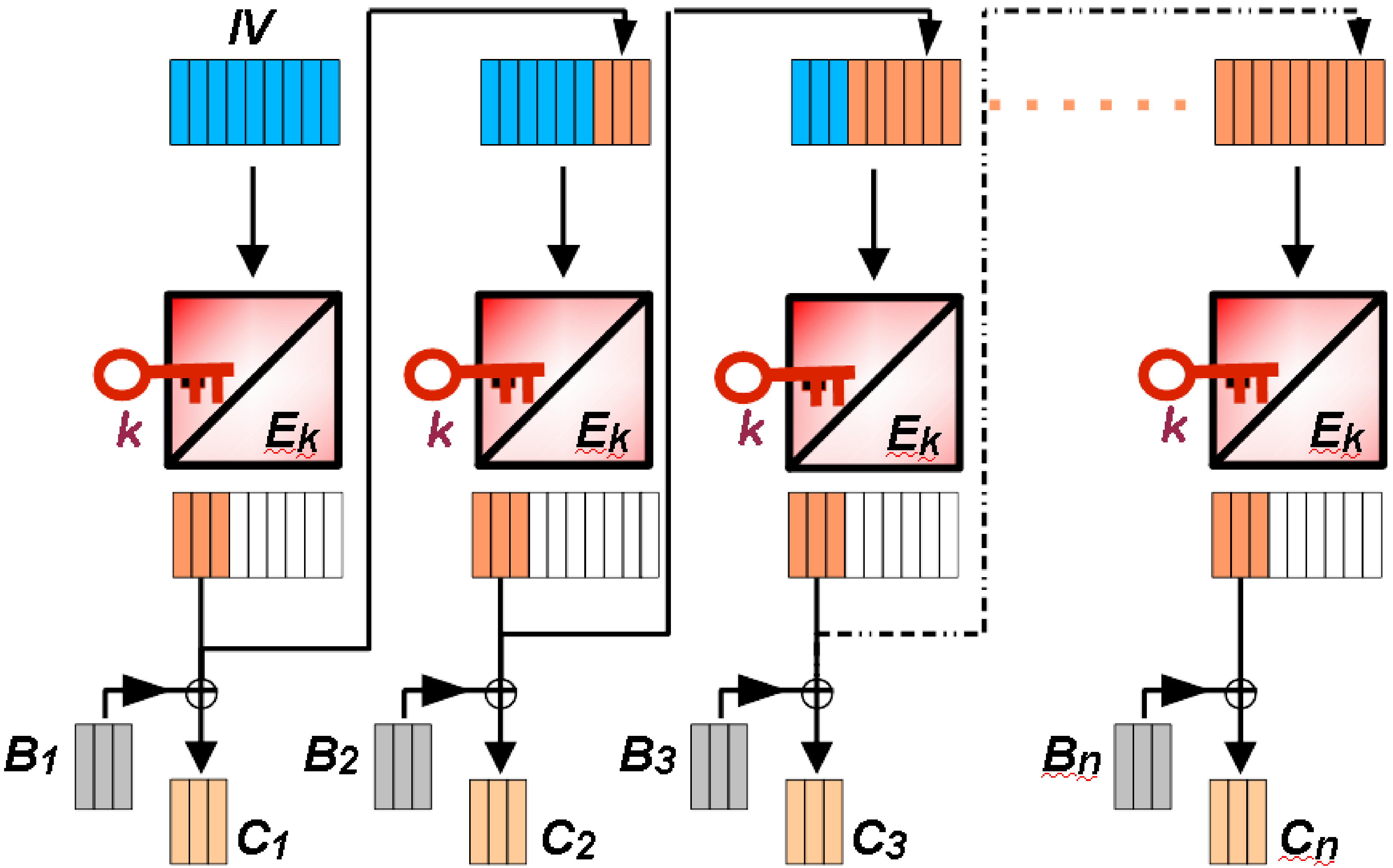


CFB mode (Cipher Feedback) - Decryption

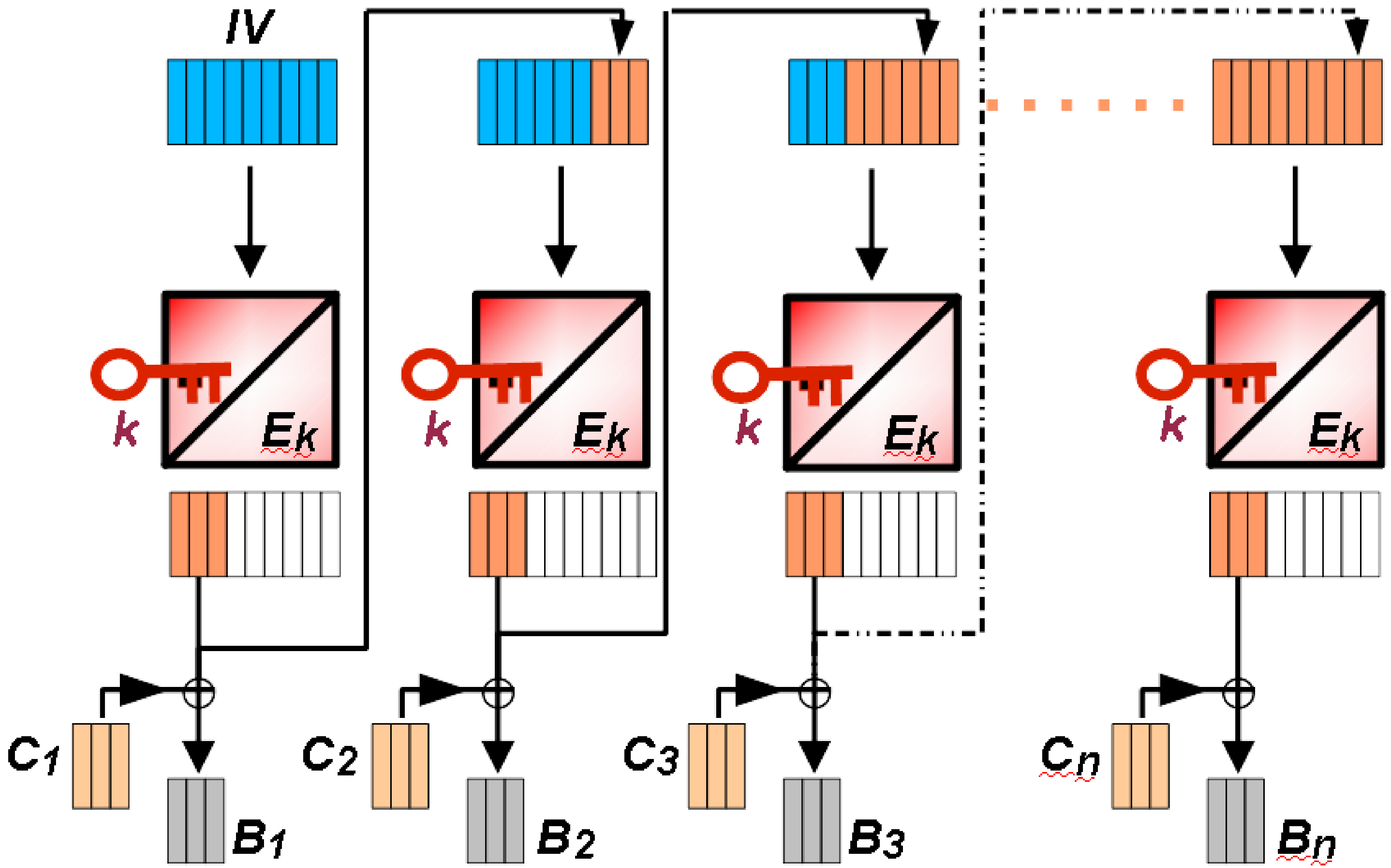


```java
byte[] key = Dump.hexString2byteArray(
 "0102030405060708090A0B0C0D0E0F10");
SecretKey secretKey = new SecretKeySpec(key, "AES");

Cipher cipher
 = Cipher.getInstance("AES/CFB24/NOPADDING");

byte[] ivBytes = new byte[16];
(new Random()).nextBytes(ivBytes);
IvParameterSpec iv = new IvParameterSpec(ivBytes);
cipher.init(Cipher.ENCRYPT_MODE, secretKey, iv);

byte[] m = "Test".getBytes();

System.out.println("Plaintext:");
System.out.println(Dump.dump(m));
System.out.println();

byte[] c = cipher.doFinal(m);
```

OFB mode (Output Feedback) - Encryption



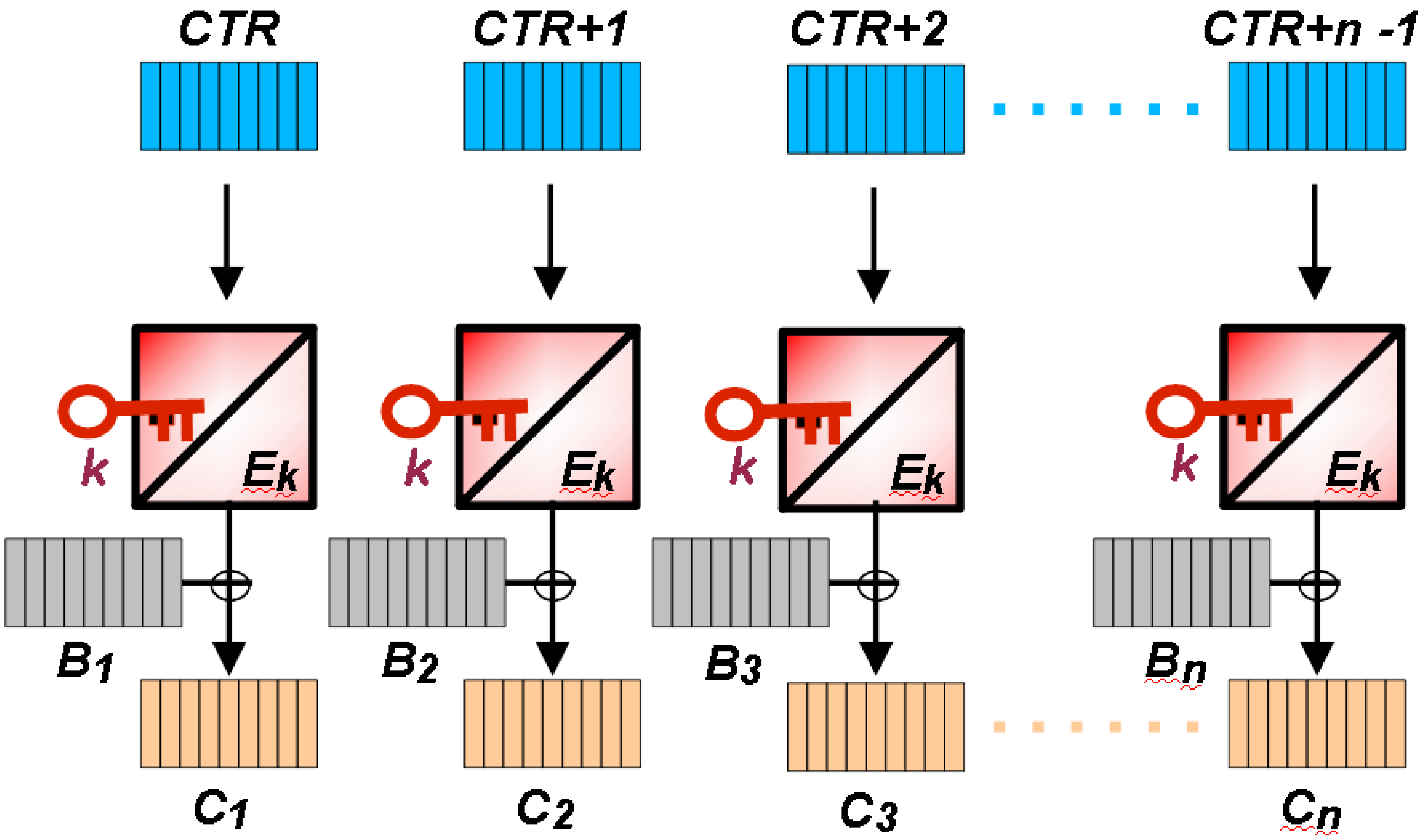
OFB mode (Output Feedback) - Decryption



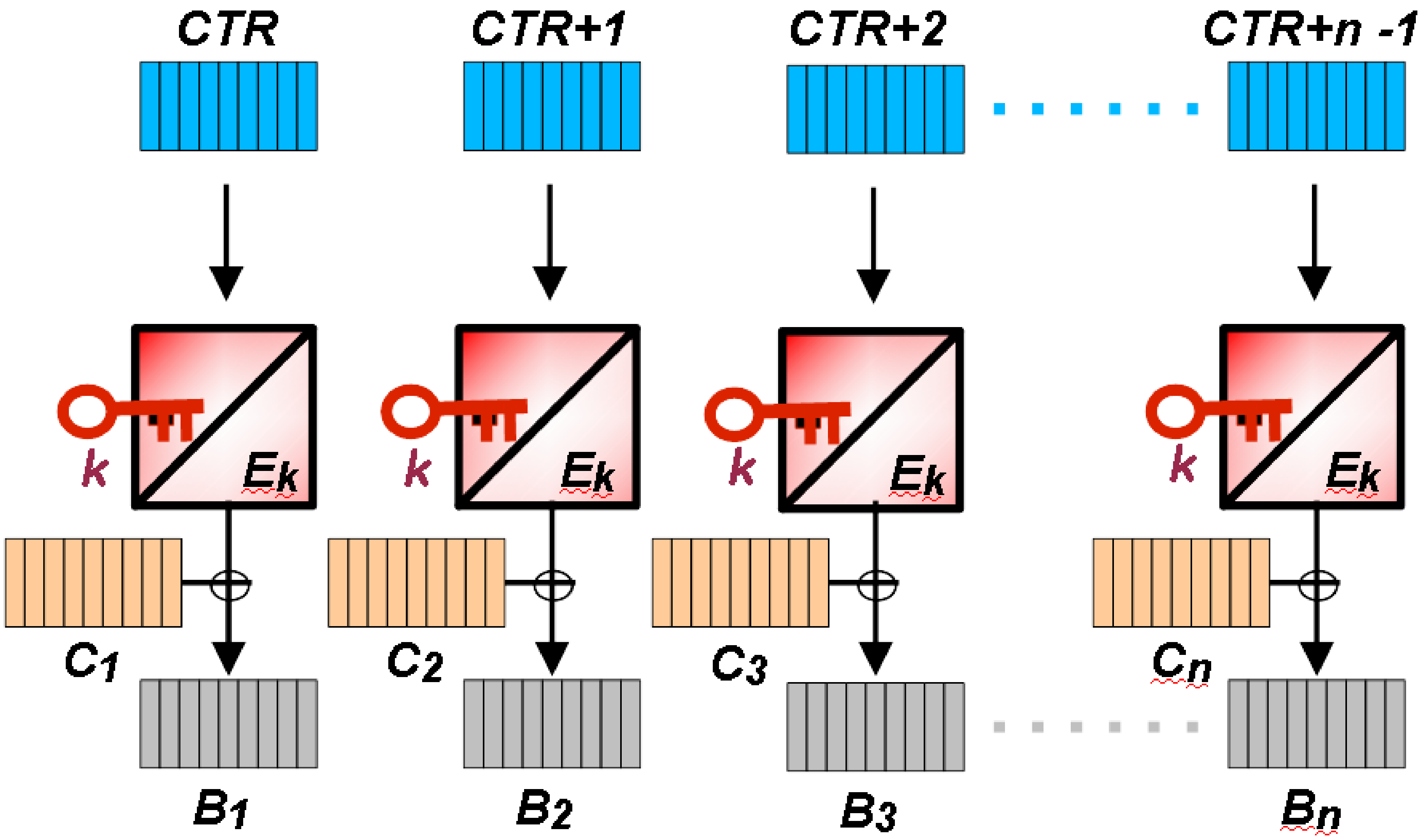
Cipher_OFB_Demo


```java
        byte[] key = Dump.hexString2byteArray(
          "0102030405060708090A0B0C0D0E0F10");
        SecretKey secretKey = new SecretKeySpec(key, "AES");

        Cipher cipher
//         = Cipher.getInstance("AES/OFB24/NOPADDING");
          = Cipher.getInstance("AES/OFB/NOPADDING");

        byte[] ivBytes = new byte[16];
        (new SecureRandom()).nextBytes(ivBytes);
        IvParameterSpec iv = new IvParameterSpec(ivBytes);
        cipher.init(Cipher.ENCRYPT_MODE, secretKey, iv);

        byte[] m = "Test".getBytes();

        System.out.println("Plaintext:");
        System.out.println(Dump.dump(m));
        System.out.println();

        byte[] c = cipher.doFinal(m);
```

CTR mode (Counter) - Encryption



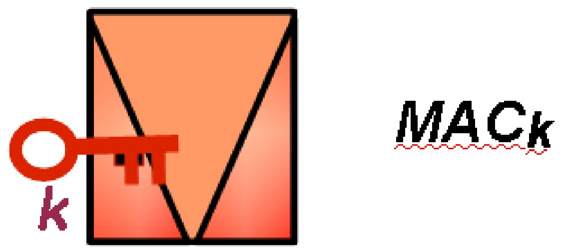
CTR mode (Counter) - Decryption



Cipher_CTR_Demo

```java
byte[] key = Dump.hexString2byteArray(
 "0102030405060708090A0B0C0D0E0F10");
SecretKey secretKey = new SecretKeySpec(key, "AES");

Cipher cipher
 = Cipher.getInstance("AES/CTR/NOPADDING");

byte[] ctr = Dump.hexString2byteArray(
 "FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE");
IvParameterSpec iv = new IvParameterSpec(ctr);
cipher.init(Cipher.ENCRYPT_MODE, secretKey, iv);

byte[] m = new byte[35];

byte[] c = cipher.doFinal(m);
```

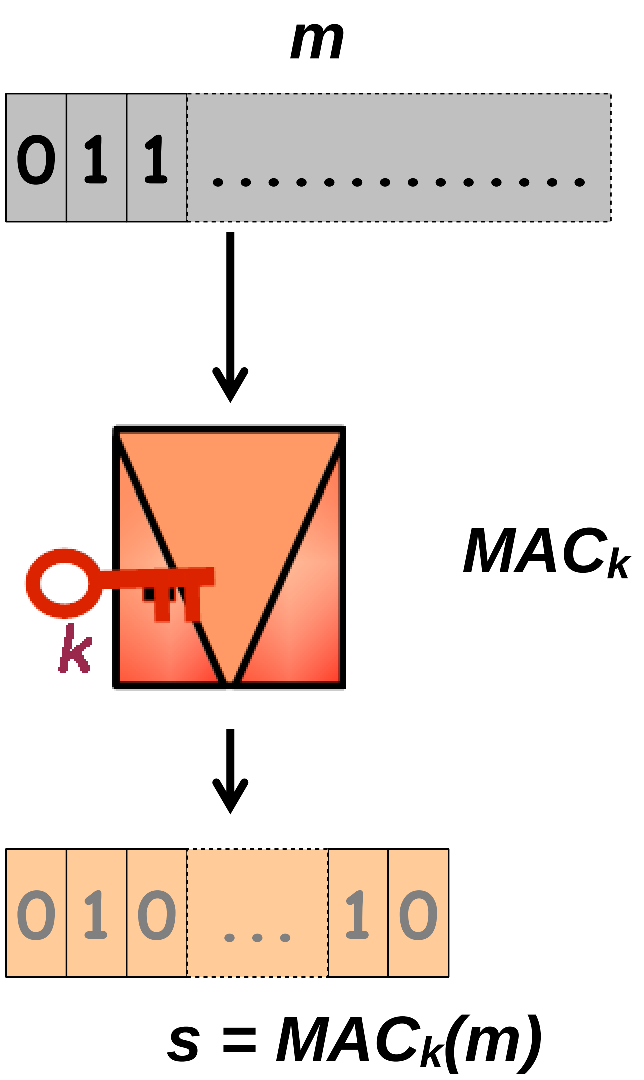
MAC‘s (Message Authentication Codes)



A Message Authentication is a hash function that depends on a key k.

Again the set of all binary sequences of finite length $m = (m_1, m_2, m_3, \ldots)$ is mapped to the set of binary sequences of some fixed length n:

$$Mac_k(m) = (h_1, h_2, \ldots, h_n) \in (F_2)^n$$



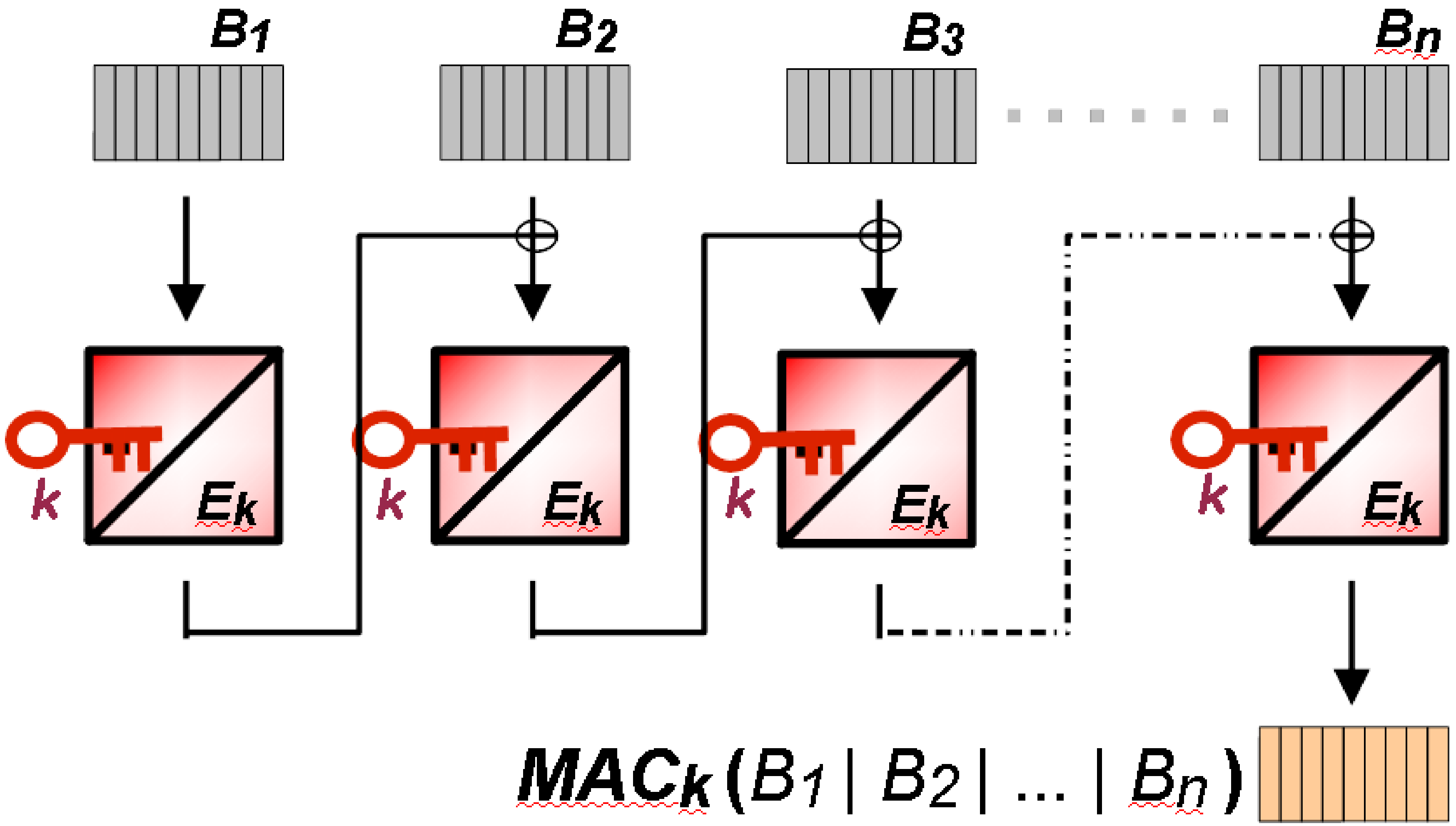
Applications:

Protection of message authenticity within a (closed) group of users with a common secret key.

Constructions of MAC‘s:

- CBC MAC based on a symmetric block cipher
- HMAC (Hash MAC) based on a hash function

CBC MAC



RFC 2104 - Keyed-Hashing for Message Authentication

- Hash function H, using a compression function which compresses b bytes from the input per round.
- (Example: b = 64 for SHA-1)
- Hash output length h (Example: h = 20 für SHA-1)
- HMAC(m) = H(k XOR opad | H(k XOR ipad | m))

  ipad = 0x36 | 0x36 | … | 0x36 (b Bytes)

  opad = 0x5c | 0x5c | … | 0x5c (b Bytes)

RFC 5869 - HMAC-based Extract-and-Expand Key Derivation Function (HKDF)

- HKDF-Extract(salt, IKM) -> PRK
- HKDF-Expand(PRK, info, L) -> OKM
- Used in TLS 1.3 for key derivations, see RFC 8446, 7.1.

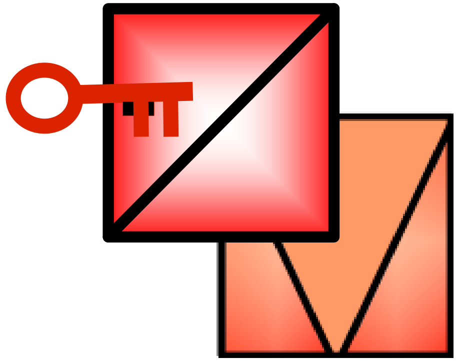
```
7.1.  Key Schedule

   The key derivation process makes use of the HKDF-Extract and
   HKDF-Expand functions as defined for HKDF [RFC5869], as well as the
   functions defined below:

       HKDF-Expand-Label(Secret, Label, Context, Length) =
            HKDF-Expand(Secret, HkdfLabel, Length)

       Where HkdfLabel is specified as:

       struct {
           uint16 length = Length;
           opaque label<7..255> = "tls13 " + Label;
           opaque context<0..255> = Context;
       } HkdfLabel;

       Derive-Secret(Secret, Label, Messages) =
            HKDF-Expand-Label(Secret, Label,
                              Transcript-Hash(Messages), Hash.length)
```

TLS 1.3 – Key Schedule

```
             0
             |
             v
   PSK ->  HKDF-Extract = Early Secret
             |
             +-----> Derive-Secret(., "ext binder" | "res binder", "")
             |                     = binder_key
             |
             +-----> Derive-Secret(., "c e traffic", ClientHello)
             |                     = client_early_traffic_secret
             |
             +-----> Derive-Secret(., "e exp master", ClientHello)
             |                     = early_exporter_master_secret
             v
       Derive-Secret(., "derived", "")
             |
             v
   (EC)DHE -> HKDF-Extract = Handshake Secret
             |
             +-----> Derive-Secret(., "c hs traffic",
             |                     ClientHello...ServerHello)
             |                     = client_handshake_traffic_secret
             |
             +-----> Derive-Secret(., "s hs traffic",
             |                     ClientHello...ServerHello)
             |                     = server_handshake_traffic_secret
             v
       Derive-Secret(., "derived", "")
             |
             v
   0 -> HKDF-Extract = Master Secret
             |
             +-----> Derive-Secret(., "c ap traffic",
             |                     ClientHello...server Finished)
             |                     = client_application_traffic_secret_0
             |
             +-----> Derive-Secret(., "s ap traffic",
             |                     ClientHello...server Finished)
             |                     = server_application_traffic_secret_0
             |
             +-----> Derive-Secret(., "exp master",
             |                     ClientHello...server Finished)
             |                     = exporter_master_secret
             |
             +-----> Derive-Secret(., "res master",
                                   ClientHello...client Finished)
                                   = resumption_master_secret
```

AEAD Algorithms

- **Authenticated Encryption with Associated Data**



AEAD

- **Algorithm for combined encryption/decryption and MAC calculation/verification**
- **Encryption and MAC calculation:**
  - Input: Plaintext P, Additional Data A, Key k, Nonce IV
  - Output: Ciphertext $C = E_{k,IV}(P)$, MAC $T = MAC_{k,IV}(A,C)$
- ***AEAD algorithm used with TLS:***
  - Galois/Counter Mode (GCM) of operation of the AES algorithm (NIST Special Publication 800-38D)
  - ChaCha20 and Poly1305 (RFC 8439)

GCM – Encryption and MAC (Tag) calculation

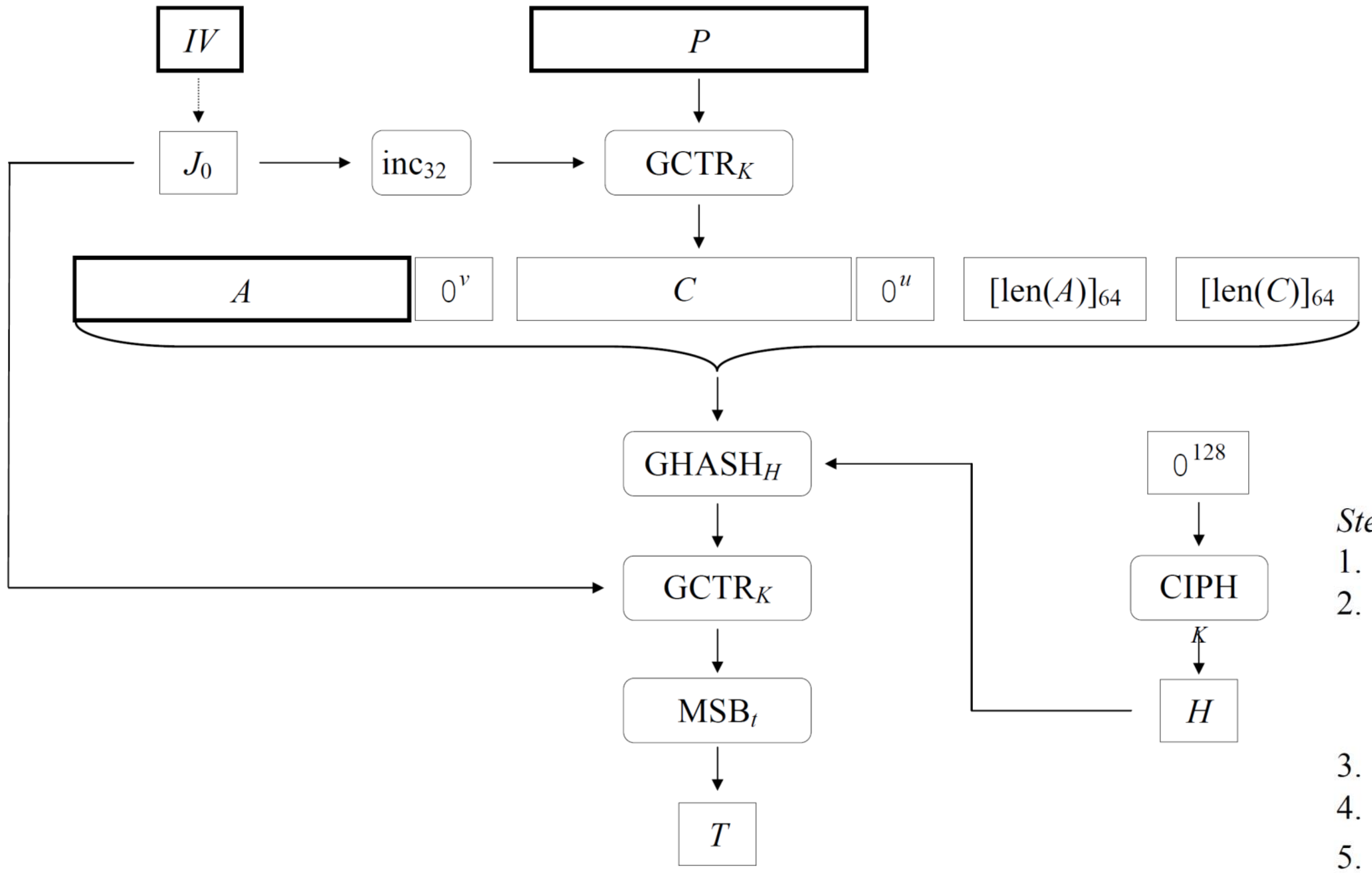


Figure 3: $GCM\text{-}AE_K(IV, P, A) = (C, T)$.

[NIST Special Publication 800-38D]

Steps:

1. Let $H = CIPH_K(0^{128})$.
2. Define a block, J_0, as follows:
   If len(IV)=96, then let $J_0 = IV \,||\, 0^{31} \,||1$.
   If len(IV) $\neq$ 96, then let $s = 128\lceil len(IV)/128\rceil - len(IV)$, and let
   $J_0 = GHASH_H(IV || 0^{s+64} || [len(IV)]_{64})$.
3. Let $C = GCTR_K(inc_{32}(J_0), P)$.
4. Let $u = 128 \cdot \lceil len(C)/128 \rceil - len(C)$ and let $v = 128 \cdot \lceil len(A)/128 \rceil - len(A)$.
5. Define a block, S, as follows:
   $$S = GHASH_H(A \,||\, 0^v \,||\, C \,||\, 0^u \,||\, [len(A)]_{64} \,||\, [len(C)]_{64}).$$
6. Let $T = MSB_t(GCTR_K(J_0, S))$.
7. Return (C, T).

GCM – Encryption in CTR mode

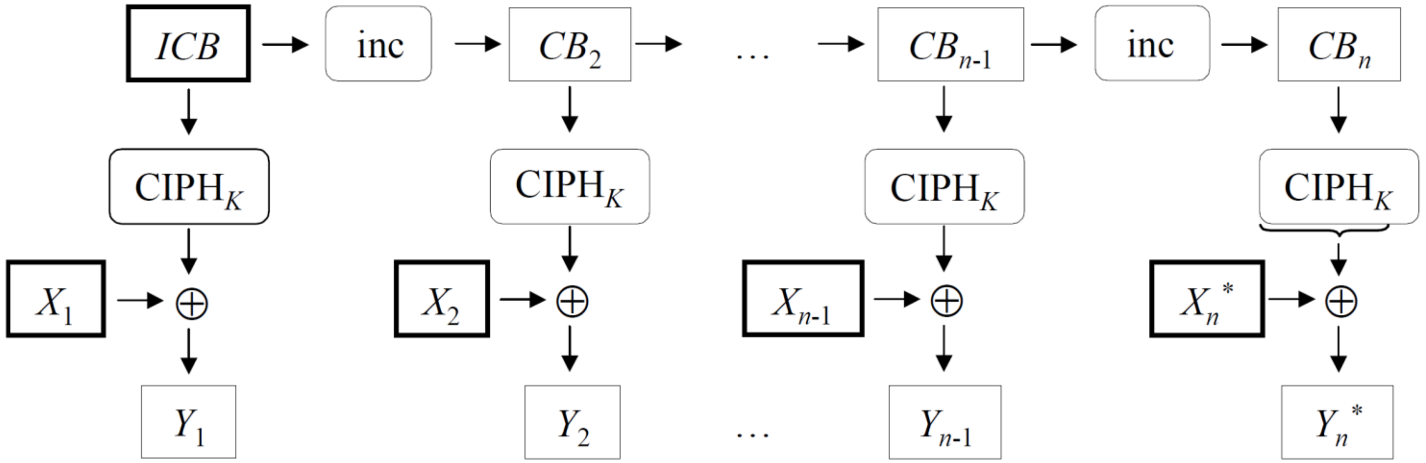


Figure 2: $\mathrm{GCTR}_K\,(ICB, X_1 \,||\, X_2 \,||\, ... \,||\, X_n^*) = Y_1 \,||\, Y_2 \,||\, ... \,||\, Y_n^*$.

[NIST Special Publication 800-38D]

GCM – GHASH function

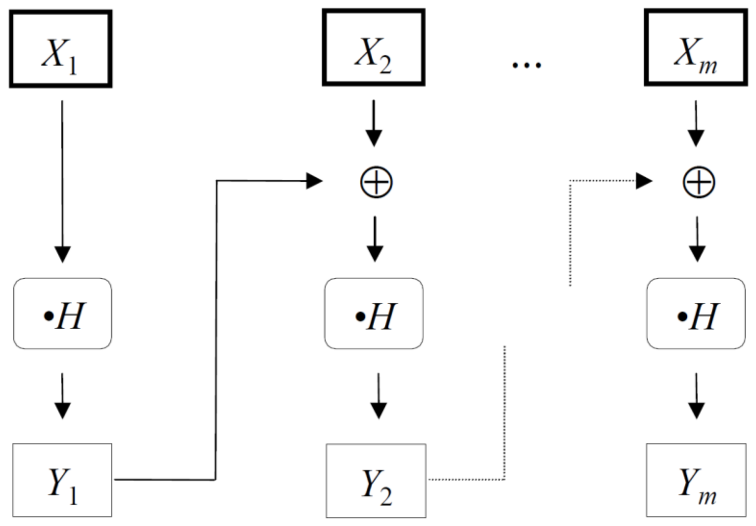


Figure 1: $\text{GHASH}_H(X_1 \| X_2 \| \ldots \| X_m) = Y_m$.

[NIST Special Publication 800-38D]

Let R be the bit string $11100001 \| 0^{120}$.

The $\bullet$ operation on (pairs of) the 2^{128} possible blocks corresponds to the multiplication operation for the binary Galois (finite) field of 2^{128} elements. The fixed block, R, determines a representation of this field as the modular multiplication of binary polynomials of degree less than 128.

Algorithm 1: $X \bullet Y$

Input:
blocks X, Y.

Output:
block $X \bullet Y$.

Steps:

1. Let $x_0x_1 \ldots x_{127}$ denote the sequence of bits in X.
2. Let $Z_0 = 0^{128}$ and $V_0 = Y$.
3. For $i = 0$ to 127, calculate blocks Z_{i+1} and V_{i+1} as follows:

$$Z_{i+1} = \begin{cases} Z_i & \text{if } x_i = 0; \\ Z_i \oplus V_i & \text{if } x_i = 1. \end{cases}$$

$$V_{i+1} = \begin{cases} V_i >> 1 & \text{if } \text{LSB}_1(V_i) = 0; \\ (V_i >> 1) \oplus R & \text{if } \text{LSB}_1(V_i) = 1. \end{cases}$$

4. Return Z_{128}.

GCM – Decryption and MAC (Tag) verification

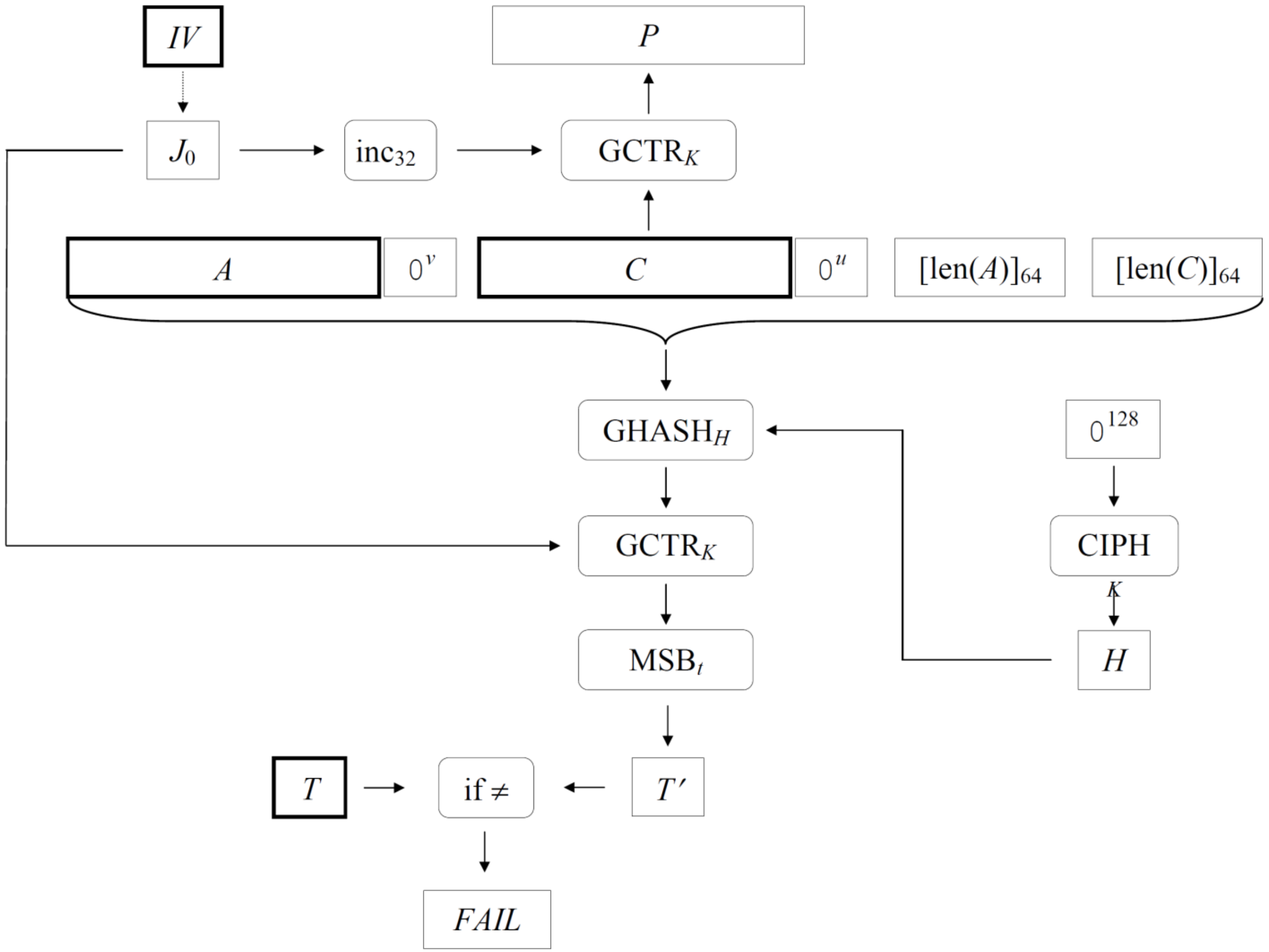


Figure 4: $GCM\text{-}AD_K(IV, C, A, T) = P$ or $FAIL$.

[NIST Special Publication 800-38D]

Asymmetric Cryptography

- **Public Key Cryptography**

- **Problem**

  Usage of symmetric ciphers require the exchange of secret keys over some secure channel.

- **Basic Idea**

  Usage of a mathematical operation, whose inversion is not computational feasible without the knowledge of a key value (trap door function).

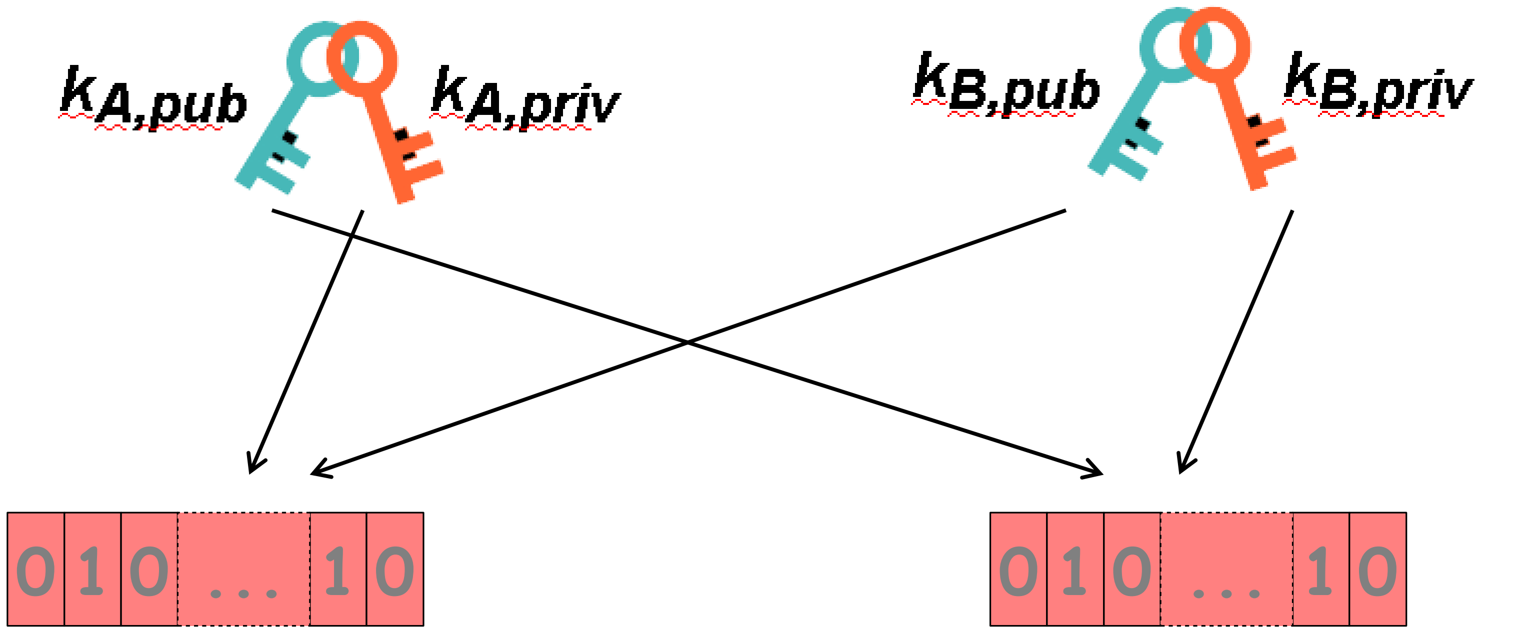
  - Factorization of integers
  - Calculation of discrete logarithms in $\mathbb{Z}_p$
  - Calculation of discrete logarithms in groups defined by elliptic curves over finite fields

First published solutions:

- W. Diffie, M.E. Hellman, *New Directions in Cryptography*, 1976
- R.C. Merkle, *Secure Communication over Insecure Channels*, 1978
- R.L. Rivest, A. Shamir, L.M. Adleman, *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*, 1978

- **Algorithms from public key cryptography:**
  - Key derivation algorithms / schemes
  - Asymmetric Ciphers (encryption without a shared secret key)
  - Digital Signatures

Key Derivation



$k = F(k_{A,priv}, k_{B,pub})$

$k = F(k_{B,priv}, k_{A,pub})$

Diffie-Hellman (DH) key derivation

- Key derivation scheme proposed by **W. Diffie** and **M.E. Hellman** in ***New Directions in Cryptography*** (1976).
- Based on the mathematical (computational) problem of finding **discrete logarithms**. (Multiplicative order of an element in **<*g*>** for some fixed $g \in \mathbb{Z}_n$.)
- **ECDH** (Elliptic Curve Diffie-Hellman): Based on the problem of determining the order of a point of an elliptic curve defined over a finite field.
  - Applying elliptic curves in cryptography was suggested by **N. Koblitz** and **V. S. Miller** in 1985.
  - Widely used since ~2005.

Let $b \in \mathbb{N}$, $n \in \mathbb{N}$.

DL-Problem: Determine for a given

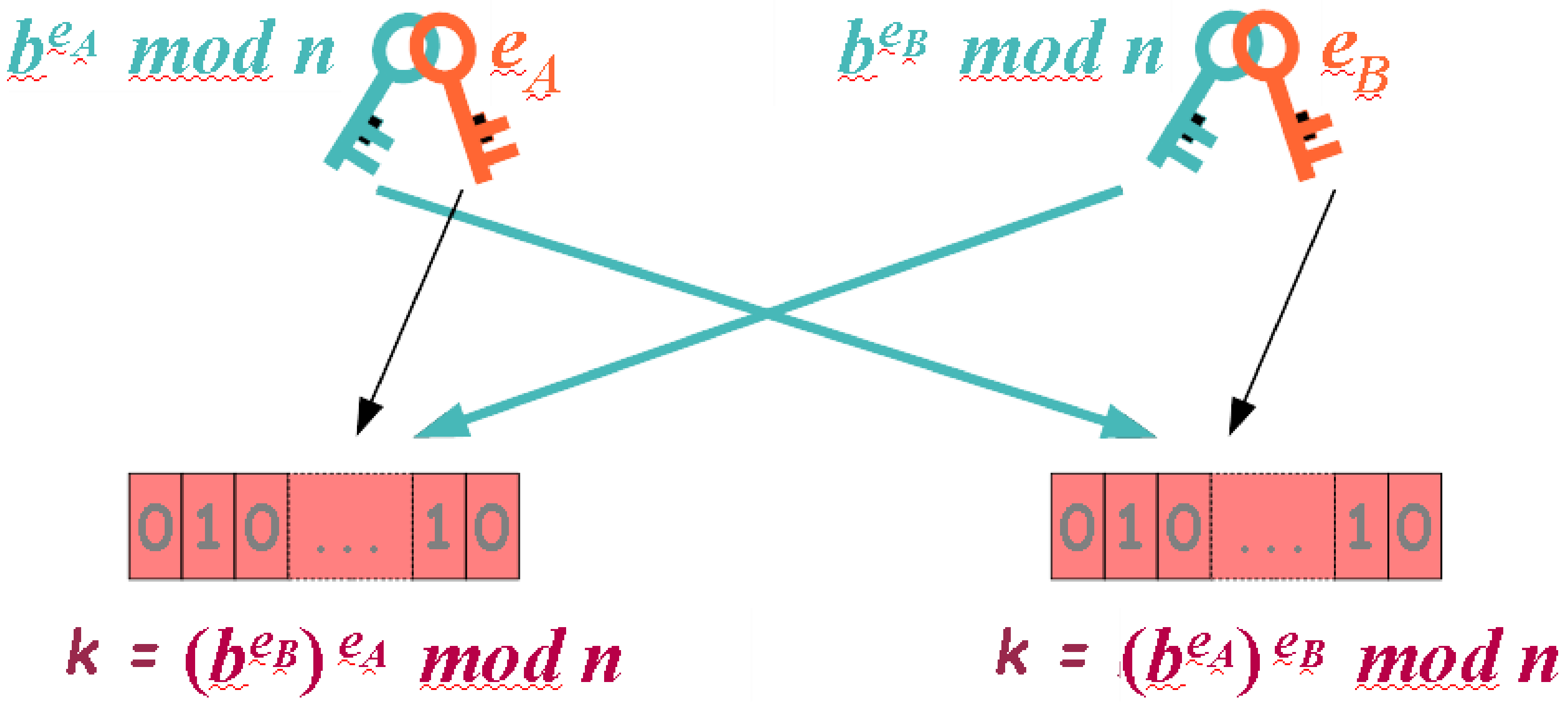
power $c = b^e \ mod \ n$

the exponent e.

Diffie-Hellman (DH) key derivation

$n \in \mathbb{N}$ *prime number*

$b \in \{2, 3, \ldots, (n-2)\}$

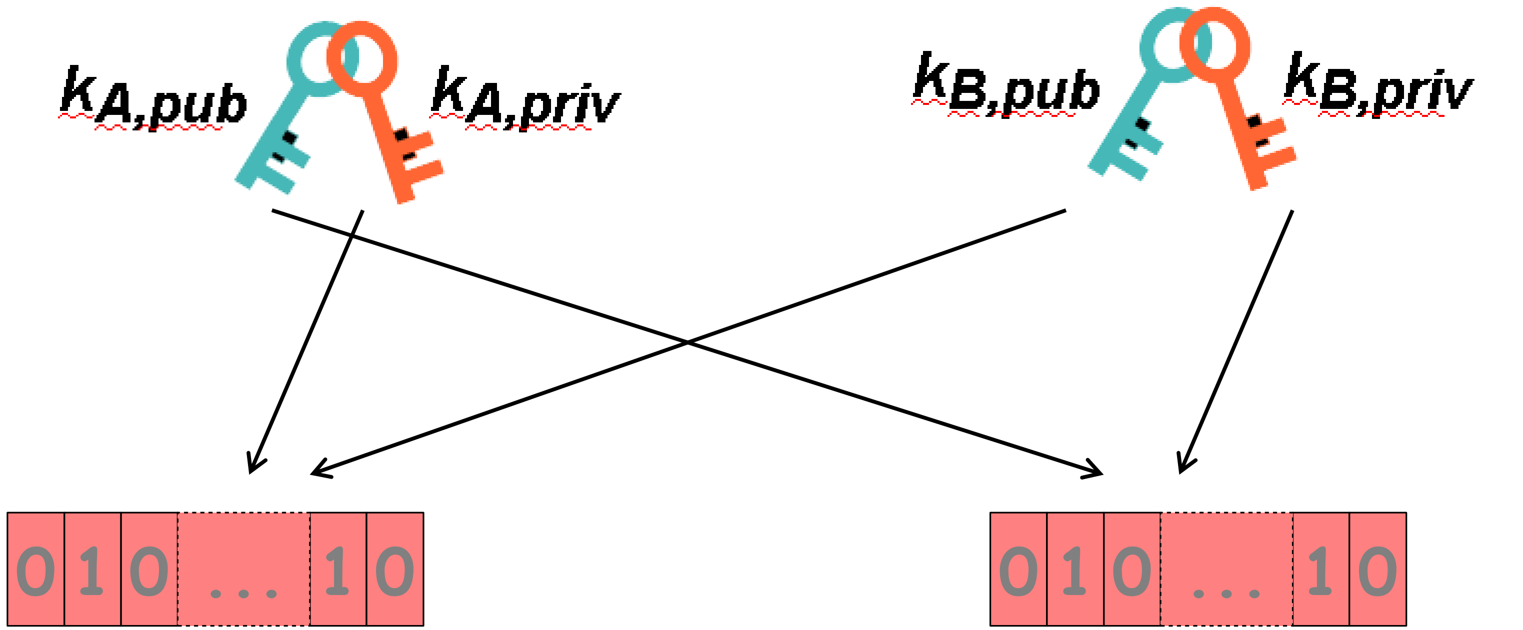


$$n \in \mathbb{N} \quad \textit{prime number}$$
$$b \in \{2, 3, \ldots, (n-2)\}$$

- ***n* -1** should have a big prime factor ***q***, such that ***q*** divides the order of ***b***.
- The order of ***b*** should be large.

- ECC (Elliptic Curve Cryptography) is based on the group structure on the sets of points of an Elliptic Curve defined over $\mathbf{F}_p$ or $\mathbf{F}_{2^n}$.

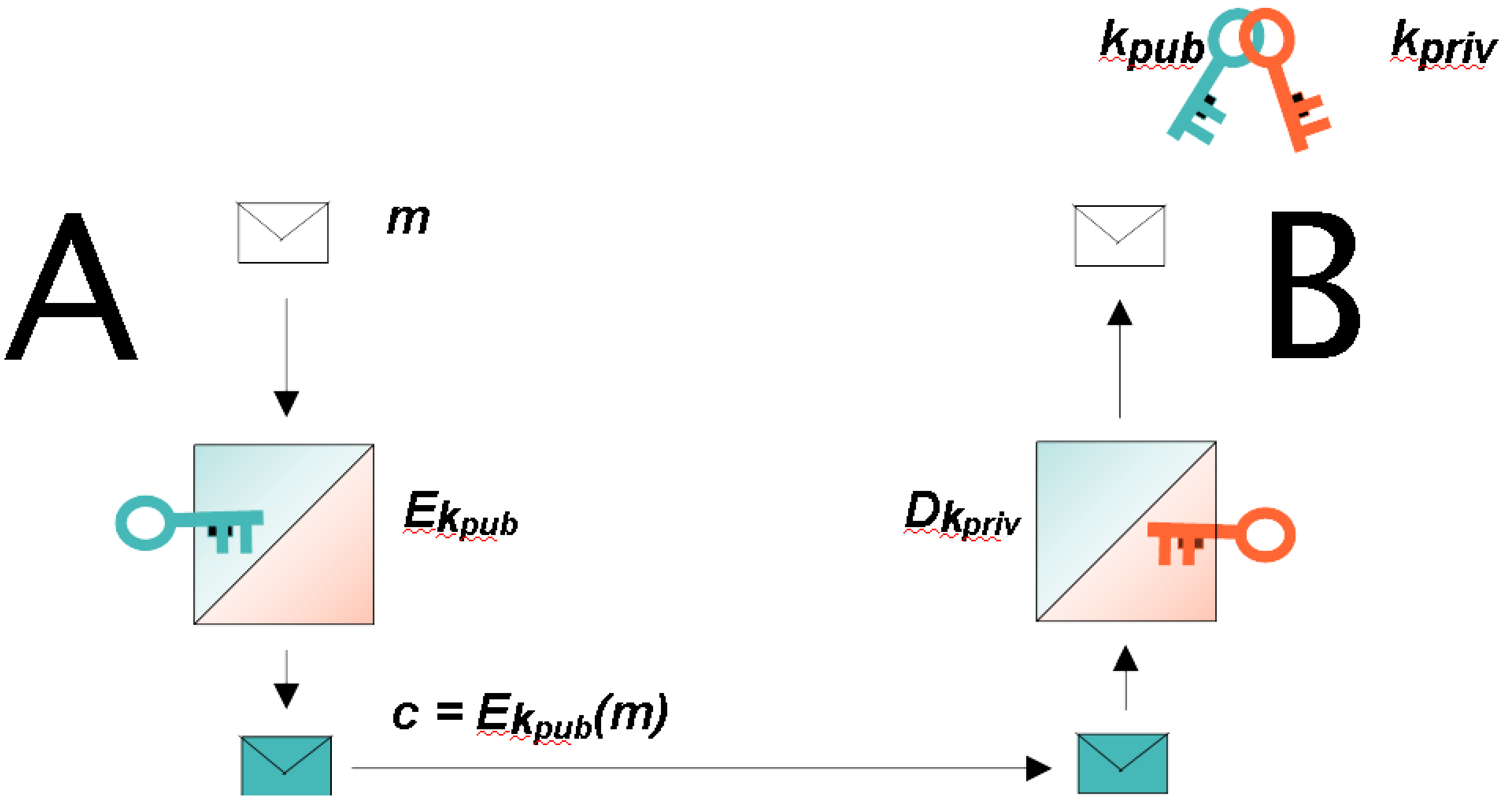
Elliptic Curve Diffie-Hellman (EC-DH) key derivation



$k = F(\, k_{A,priv},\, k_{B,pub}\,)$

$k = F(\, k_{B,priv},\, k_{A,pub}\,)$

Asymmetric Ciphers



- Public key encryption scheme proposed by **R.L. Rivest, A. Shamir, L.M. Adleman** in ***A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*** (1978)
- Depends on the mathematical (computational) problem of **factorizing integers**.

If p is a prime number and z any number coprime to p,
i.e. gcd(p,z) = 1, then

$$z^{p-1} \equiv 1 \pmod{p}$$

Proof:

- Put: $t = (1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-1)) \bmod p$
- Multiplication with z defines a bijective mapping:

  $m: \mathbb{Z}_p \to \mathbb{Z}_p,\ m(x) = (x \cdot z) \bmod p$
- It follows that:

  $t \equiv 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-1) \equiv (1 \cdot z) \cdot (2 \cdot z) \cdot (3 \cdot z) \cdot \ldots \cdot ((p-1) \cdot z) \equiv t \cdot z^{p-1} \pmod{p}$
- Division in $\mathbb{Z}_p$ by t ≠ 0 gives the claimed identity.

A consequence of Fermat's Lemma

Let p,q be prime numbers ($p \neq q$) and $r \in \mathbb{Z}$ with:

$$r \equiv 1 \ (\textbf{mod} \ \mathrm{lcm}(p\text{-}1,\ q\text{-}1))$$

Then:

$$z^r \equiv z \ \ (\textbf{mod} \ (p \cdot q)) \quad \text{for all } z \in \mathbb{Z}$$

Proof:

- If $z \equiv 0$ (**mod** p), then $z^r \equiv 0$ (**mod** p).
- If p does not divide z and $r = 1 + n(p\text{-}1)$, then:

  $z^r = z \cdot (z^{(p\text{-}1)})^n \equiv z$ (**mod** p)
- Similarly:

  $z^r \equiv z$ (**mod** q)

- Chose two random primes p and q ($> 2^{1000}$)
- Put $n = pq$, $v = lcm\,(p - 1, q - 1)$
- Define a public exponent e with:

$$\gcd(e, v) = 1$$

- Determine the private exponent d with:

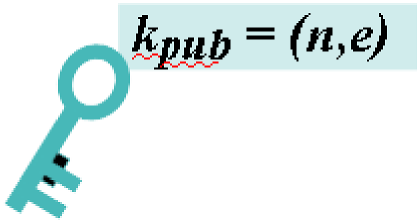
$$ed \equiv 1 \ (\mathbf{mod}\ v)$$

- Key pair (k_{pub}, k_{priv}):

$$k_{pub} = (n,e) \qquad k_{priv} = (n,d)$$

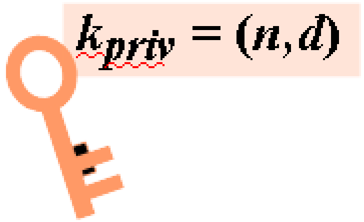
RSA encryption and decryption

- **Encryption** of a message m (< n):



$$c = E(m) = m^e \bmod n$$

- **Decryption** of c:



$$D(c) = c^d \bmod n$$

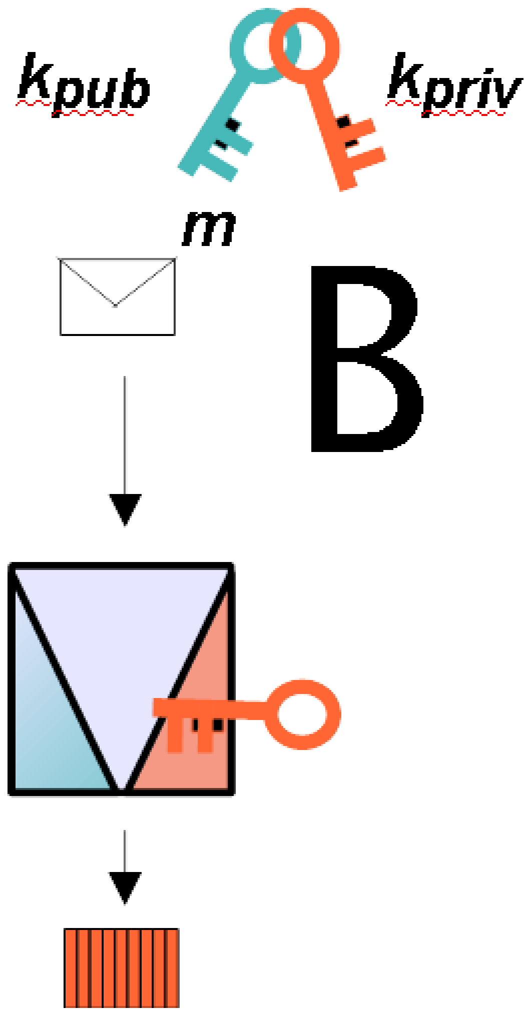
- $D(c) = m$ follows from $e \cdot d \equiv 1 \ (mod\ v)$ and Fermat‘s lemma:

$$D(c) = c^d \bmod n = (m^e)^d \bmod n = m^{ed} \bmod n = m \bmod n = m$$

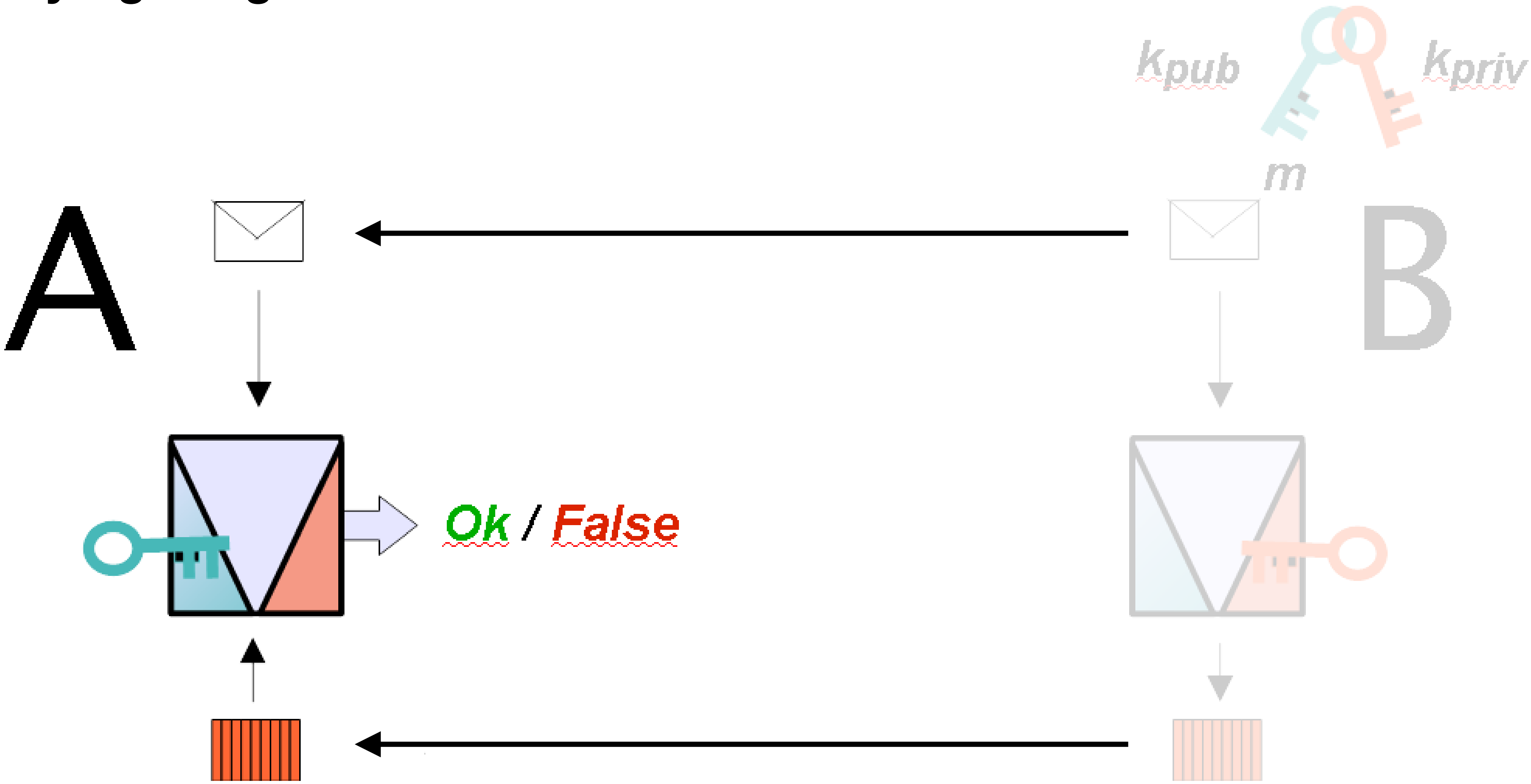
Digital Signatures

- [FIPS PUB 186-4: Digital Signature Standard (DSS)]
  - Chapter 4: The Digital Signature Algorithm (DSA)
  - Chapter 5: The RSA Digital Signature Algorithm
  - Chapter 6: The Elliptic Curve Digital Signature Algorithm (ECDSA)

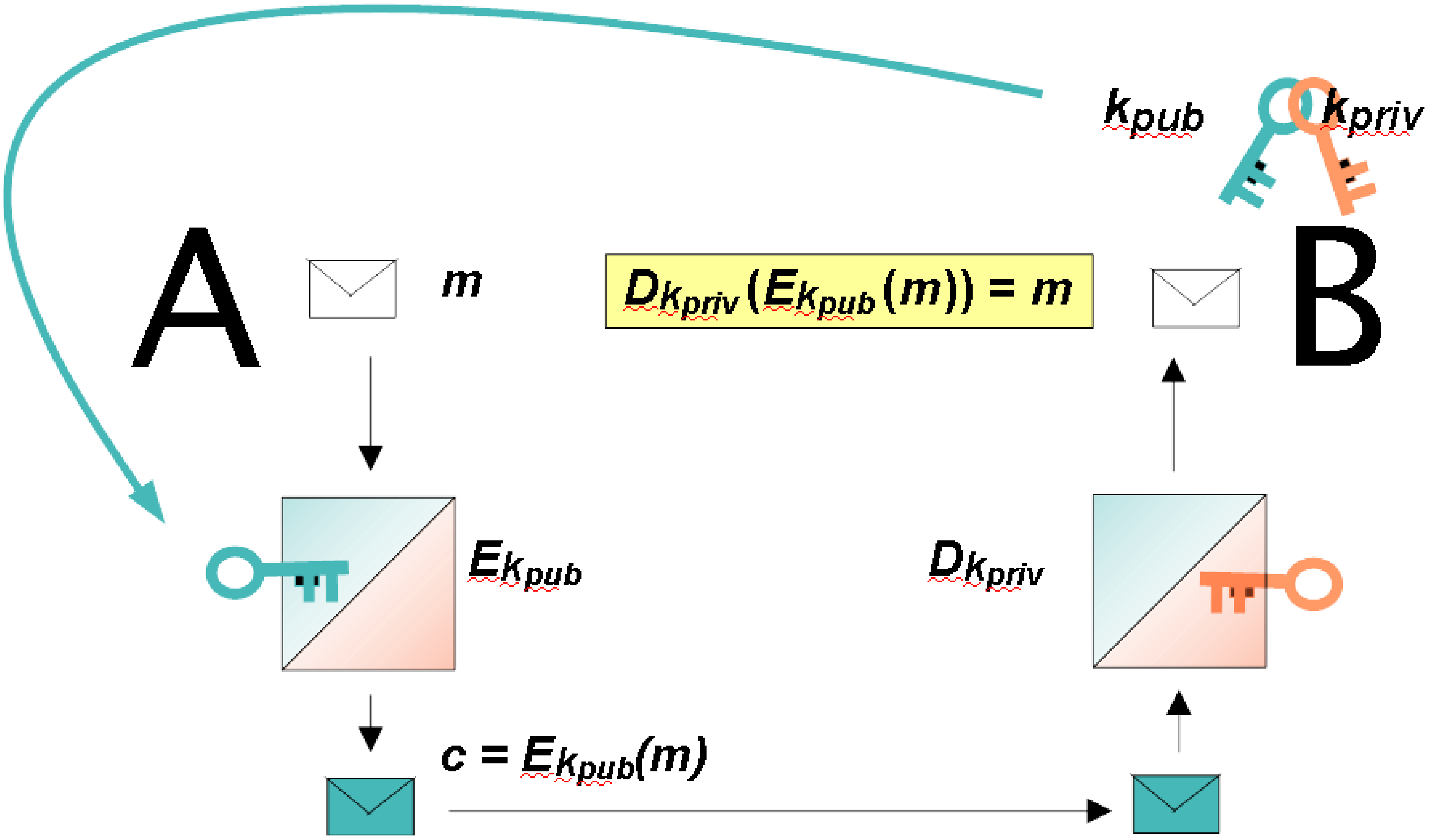
- **Signing a message**

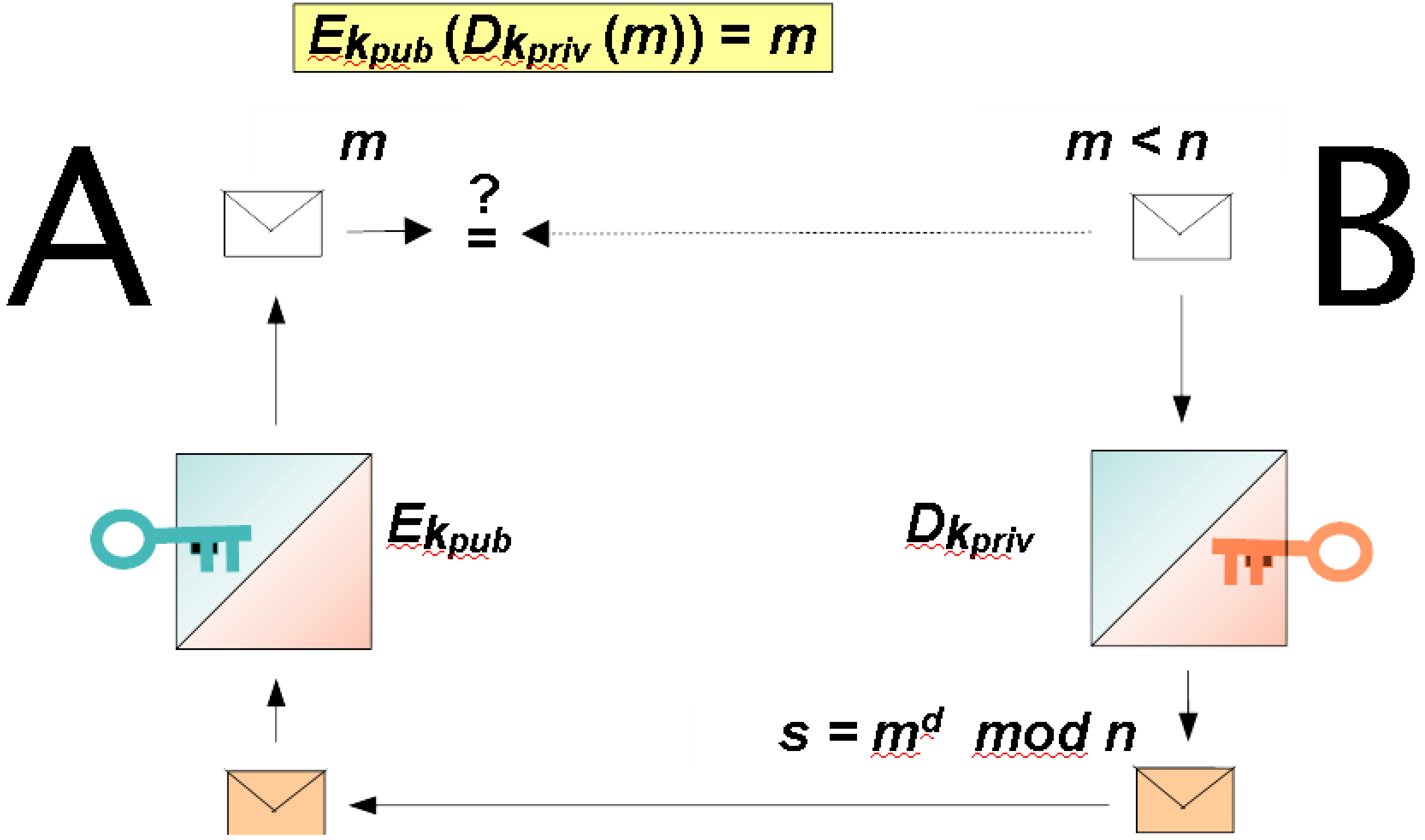


- **Verifying a signature**



RSA encryption and decryption





$E_{k_{pub}}(D_{k_{priv}}(m)) = m$
A
m
?
=
m < n
B
$E_{k_{pub}}$
$D_{k_{priv}}$
$s = m^d \mod n$

- **DSA Domain Parameters**
  - ***p*** : prime number of bit length ***L***
  - ***q*** : a prime divisor of ***p*-1** of bit length ***N***
  - ***g*** : element of **GF(*p*)*** with **o(*g*) = *q***

  - Selection of Parameter Sizes and **Hash** Functions for DSA:
    - $L = 1024$, $N = 160$
    - $L = 2048$, $N = 224$
    - $L = 2048$, $N = 256$
    - $L = 3072$, $N = 256$

- **DSA Domain Parameters**
  - $\boldsymbol{p}$: prime number of bit length $\boldsymbol{L}$
  - $\boldsymbol{q}$: a prime divisor of $\boldsymbol{p\text{-}1}$ of bit length $\boldsymbol{N}$
  - $\boldsymbol{g}$: element of $\mathbf{GF}(\boldsymbol{p})^*$ with $\mathbf{o}(\boldsymbol{g}) = \boldsymbol{q}$

- **DSA Key Pairs**
  - $\boldsymbol{x}$: private key with $\mathbf{0} < \boldsymbol{x} < \boldsymbol{q}$
  - $\boldsymbol{y}$: public key $\boldsymbol{y} = \boldsymbol{g}^{\boldsymbol{x}} \bmod \boldsymbol{p}$

Digital Signature Algorithm (DSA) – FIPS PUB 186-4, Chapter 4

- Domain Parameters: p, q, g
- Key Pair: x, y
- **Signature Generation for message *M***
  - k : per message newly generated secret random number, $0 < k < q$
  - $r := (g^k \bmod p) \bmod q$
  - z : Hash(M) (leftmost N bits)
  - $s := (k^{-1}(z + xr)) \bmod q$

  - $\mathrm{Sig}_x(M) := (r, s)$

- Domain Parameters: p, q, g
- Key Pair: x, y
- Signature for M: $\mathrm{Sig}_x(M)=(r, s)$, $r = (g^k \bmod p) \bmod q$, $s = (k^{-1}(z + xr)) \bmod q$
- **Signature Verification (given M, $\mathrm{Sig}_x(M) = (r, s)$, y)**
  - $w := s^{-1} \bmod q$
  - z : Hash(M) (leftmost N bits)
  - $u_1 := (zw) \bmod q$
  - $u_2 := (rw) \bmod q$
  - $v := ((g^{u_1} y^{u_2}) \bmod p) \bmod q$
  - $\mathrm{Sig}_x(M)$ ok iff $v = r$

Elliptic Curve Digital Signature Algorithm (ECDSA)

- **FIPS PUB 186-4, Ch. 6**
  - relates strongly to ANS X9.62, Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Standard (ECDSA)
  - FIPS PUB 186-4, Appendix D: Recommended Elliptic Curves for Federal Government Use
- Certicom Research: Standards for Efficient Cryptography
  - SEC 1: Elliptic Curve Cryptography
  - SEC 2: Recommended Elliptic Curve Domain Parameters

- **ECDSA Domain Parameters**
  - ***E*** : elliptic curve over F = GF(p) or F = GF(2^m)
  - ***q*** : a large prime divisor of **|*E*| = *qh*** (with cofactor ***h***)
  - ***G*** : point of **E** with **o(G) = *q***

- **ECDSA Domain Parameters**
  - ***E*** : elliptic curve over F = GF(p) or F = GF(2^m)
  - ***q*** : a large prime divisor of **|*E*| = *qh*** (with cofactor ***h***)
  - ***G*** : point of **E** with **o(G) = q**

- **ECDSA Key Pair**
  - ***x*** : private key with **0 < *x* < *q***
  - ***Y*** : public key ***Y* = *x*·*G***

- Domain Parameters: ***E*, *q*, *G***
- Key Pair: ***x*, *Y***
- **Signature Generation for message *M***
  - ***k*** : per message newly generated secret random number, $\mathbf{0 < k < q}$
  - $R := k \cdot G = (R_x, R_y)$, $r := R_x \bmod q$
  - z : **Hash(*M*)** (leftmost *N* bits)
  - $s := (k^{-1}(z + xr)) \bmod q$
  - $\mathrm{Sig}_x(M) := (r, s)$

Elliptic Curve Digital Signature Algorithm (ECDSA)

- Domain Parameters: E, q, G
- Key Pair: x, Y
- Signature for M: $\mathrm{Sig}_x(M)=(r, s)$, $r = (k \cdot G)_x \bmod q$, $s = (k^{-1}(z + xr)) \bmod q$
- **Signature Verification (given M, $\mathrm{Sig}_x(M) = (r, s)$, Y)**
  - $w := s^{-1} \bmod q$
  - z : Hash(M) (leftmost N bits)
  - $u_1 := (zw) \bmod q$
  - $u_2 := (rw) \bmod q$
  - $V := u_1 \cdot G + u_2 \cdot Y$, $v := V_x \bmod q$
  - $\mathrm{Sig}_x(M)$ ok iff $v = r$