crypto attacks & defenses RELOADED

JP Aumasson, Philipp Jovanovic

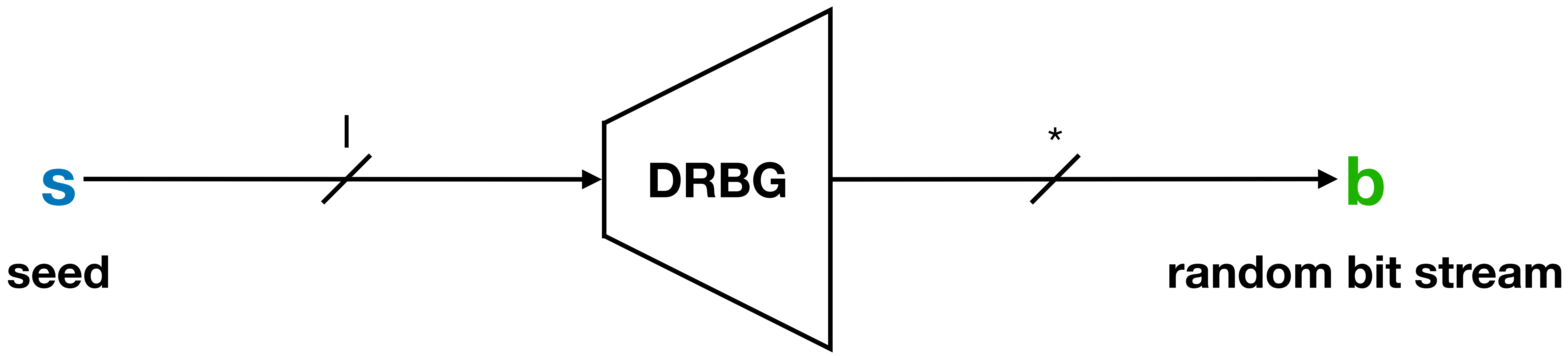
ringzerø

symmetric crypto

Goals of Symmetric Crypto

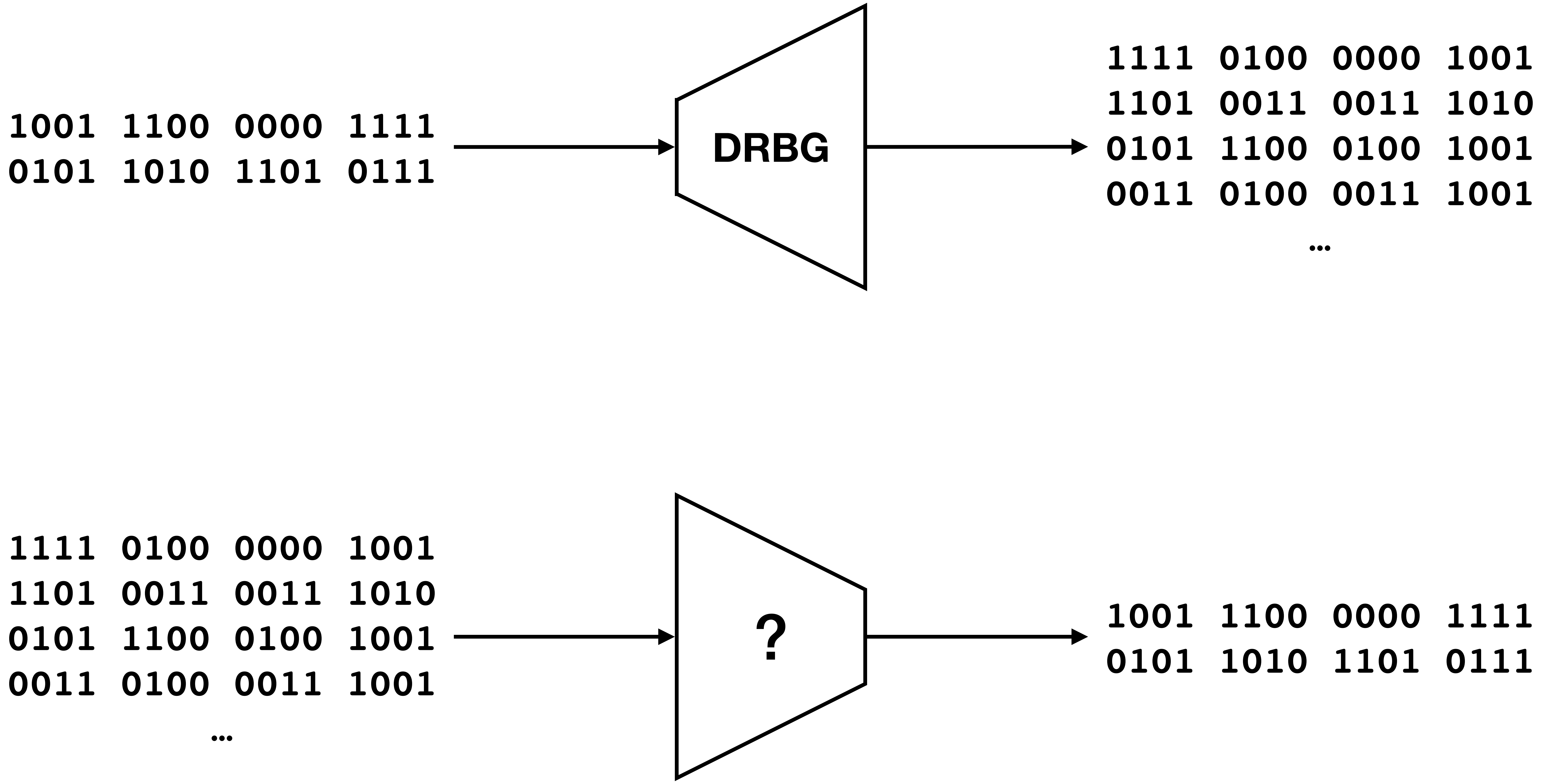
- Confidentiality (encryption)
- Integrity (hash functions)
- Authenticity (MACs)
- ~~Non-repudiation~~

Deterministic Random Bit Generators (DRBGs)

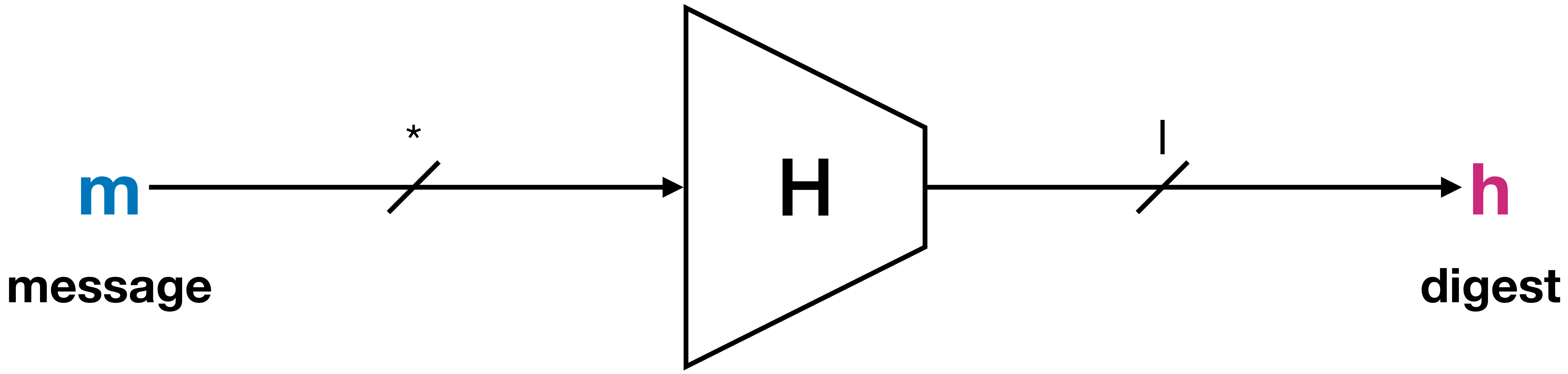


- Given a fixed-size seed **s** as input, deterministically generate an arbitrary long, uniformly random bit stream **b** = **DRBG**(**s**).
- Note: knowing some bits of **b** does not allow you to recover earlier bits (esp. not the seed **s**) or predict future ones.

What's the "Inverse" of a DRBG?



Hash Functions



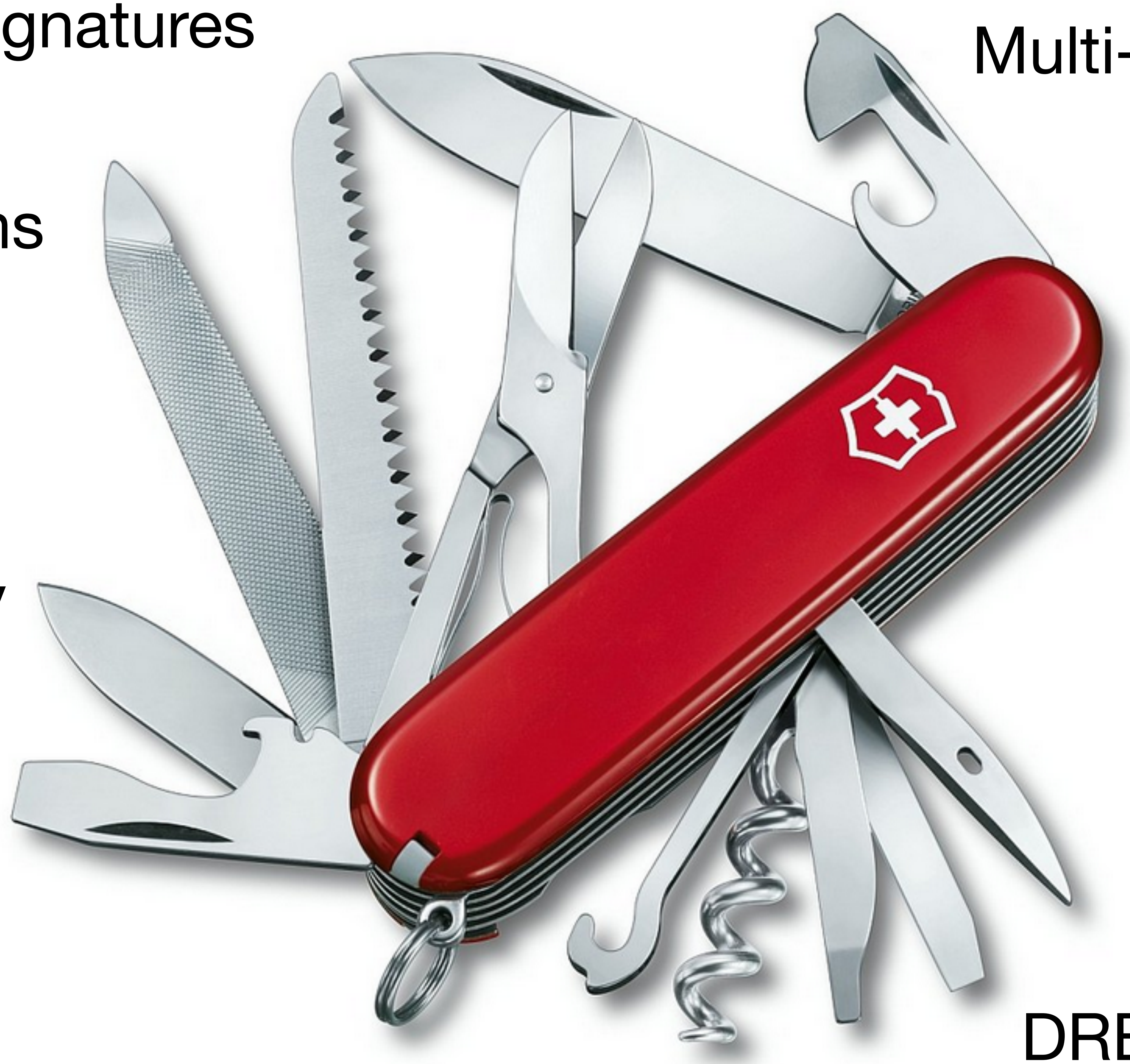
- Compress any message **m** into a short fixed-size digest **h** = **H**(**m**).
- **h** ensures the *integrity* of **m**
- Security: There should be no relations between **m** and **h** besides the one specified by **H** (i.e., no bias, no exploitable structure, etc.).
- In particular: *collisions* and *preimages* should be practically impossible to find

Hash Functions Everywhere

Public-key signatures

Multi-party protocols

Key derivation functions



File integrity

MACs

Zero-knowledge proofs

DRBGs & PRNGs

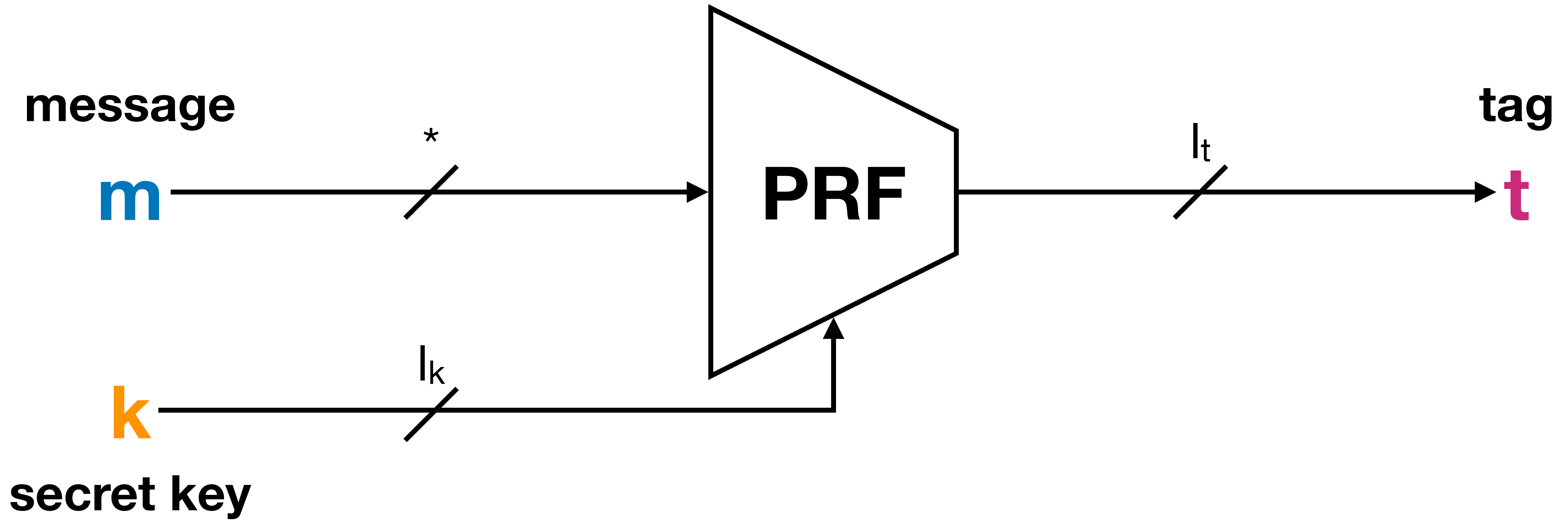
Watch Out

Do not confuse *cryptographic* with *regular* **hash functions**
(As used in data structures, or for non-cryptography checksums)

Hash Function Cheat Sheet

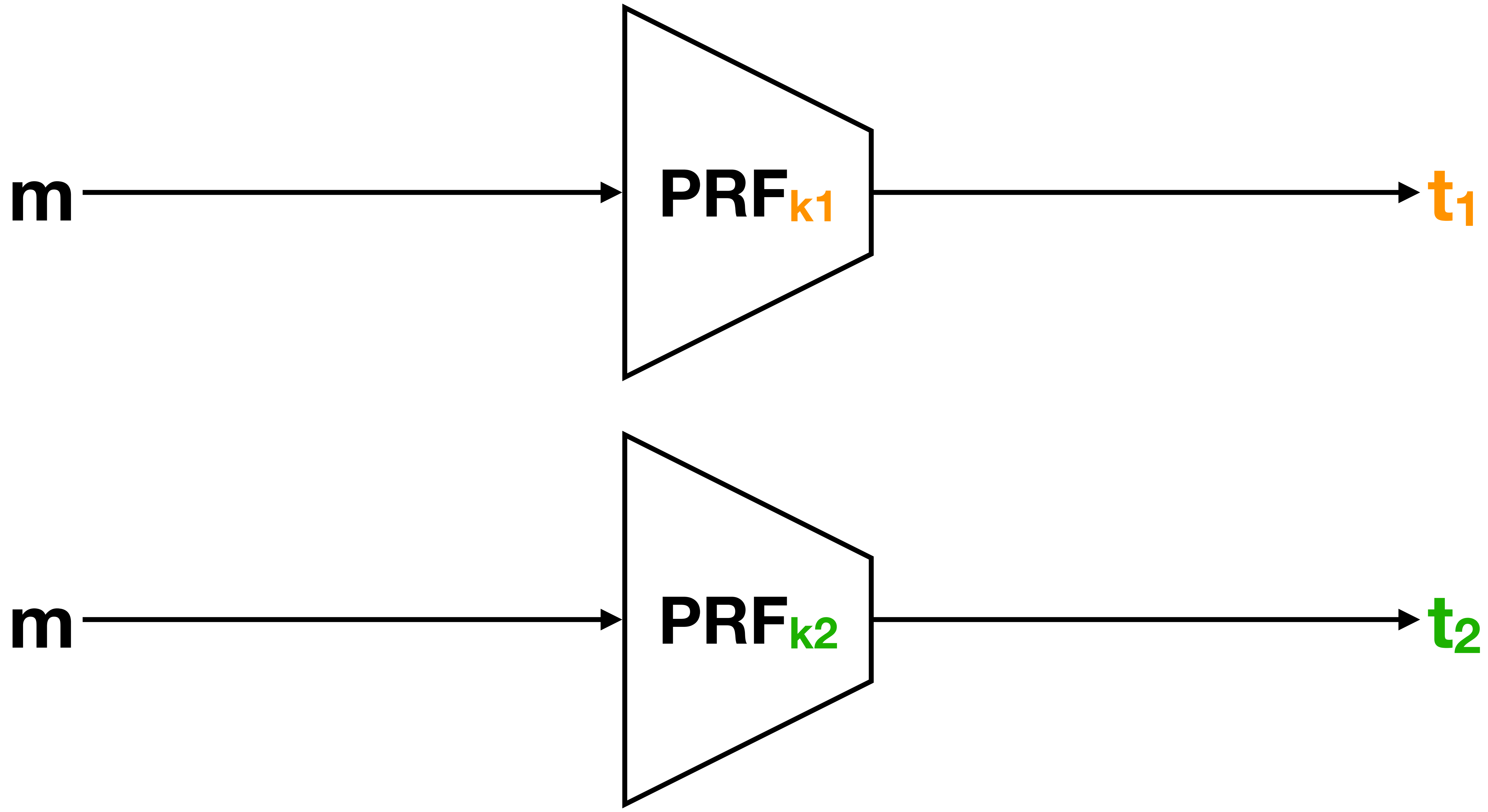
Bad	Good
MD4, MD5	BLAKE2, BLAKE3
SHA-1 (deprecated)	SHA2 (-224, -256, -384, -512)
Non-cryptographic hash functions	SHA-3
Cyclic redundancy check (CRC)	
Your own hash function	

Hashing With a Key: PRFs



- Keyed hash functions are called **pseudo-random functions (PRFs)**
- Tag **t** = **PRF**(**k**, **m**) can protect the *authenticity* of **m** (proves knowledge of **k**)
- A secure PRF is a secure **MAC** (Message Authentication Code)
- No non-repudiation, unlike in public-key signatures.

PRFs as Families of Hash Functions



Each new key create a completely “new" hash function instance, useful when different hashes are needed, even if the key does not have to be secret

HMAC: Hash-Based MAC

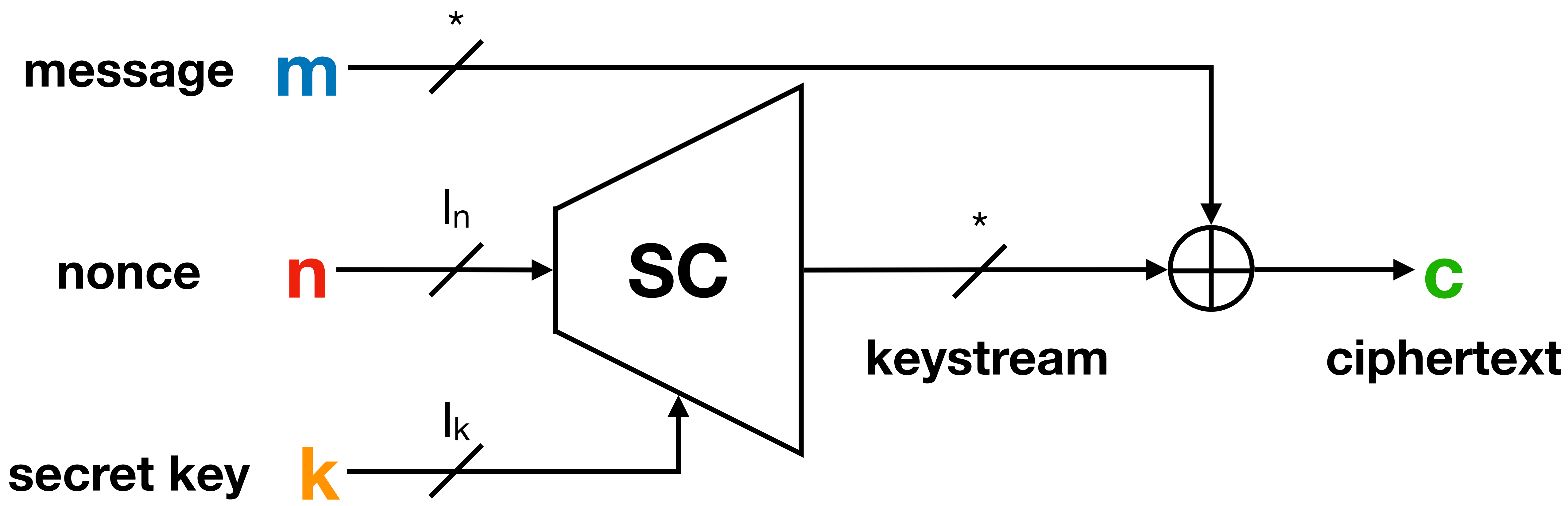
HMAC(Key, message) = **Hash**(Key1 || **Hash**(Key2 || message)

- HMAC is a MAC, but *not all MACs are HMACs* :-)
- where Key1 = Key XOR constant1, Key2 = Key XOR constant2
- “HMAC-SHA-512” = HMAC where Hash = SHA-512
- More efficient: keyed modes of SHA-3, BLAKE2, BLAKE3

PRF / MAC Cheat Sheet

Bad	Good
HMAC-MD4, HMAC-MD5	Keyed BLAKE2
H(`message` || `key`) with SHA1/2	HMAC-SHA2
H(`key` || `message`) with SHA1/2	Keyed SHA3
Your own MAC	Poly1305-AES
	SipHash

Stream Ciphers

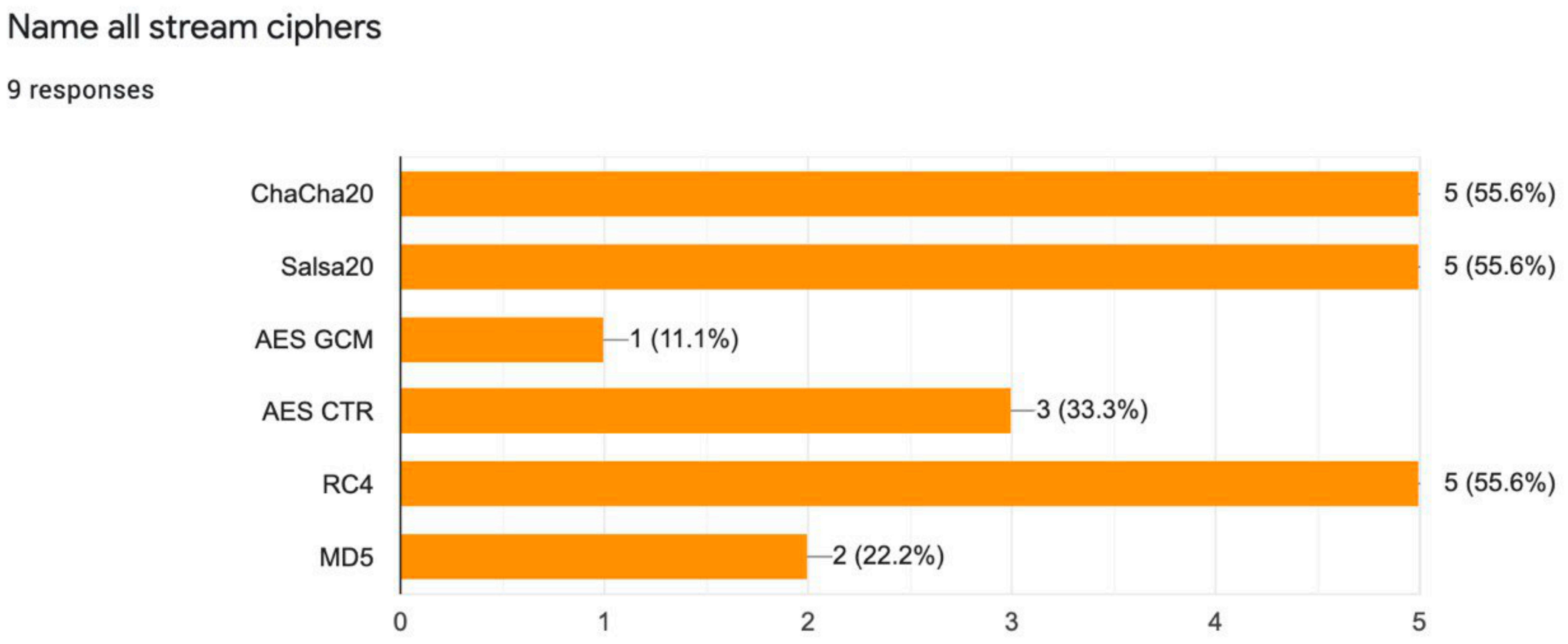


- Can be seen as a DRBG with seed = key || nonce
- Encryption and decryption are the same: XOR with the keystream
- **c** = SC(**k**,**n**,**m**) protects the *confidentiality* of **m**
- Keystreams are *unpredictable* (even if some of its bits are known)

Stream Ciphers Simulate One-Time Pad

1111 0100 0000 1001 … 1101 0011 0011 1010		Keystream
$\oplus$		
0101 1100 0100 1001 … 0011 0100 0011 1001		Plaintext
=		
1010 1000 0100 0000 … 1110 0111 0000 0011		Ciphertext

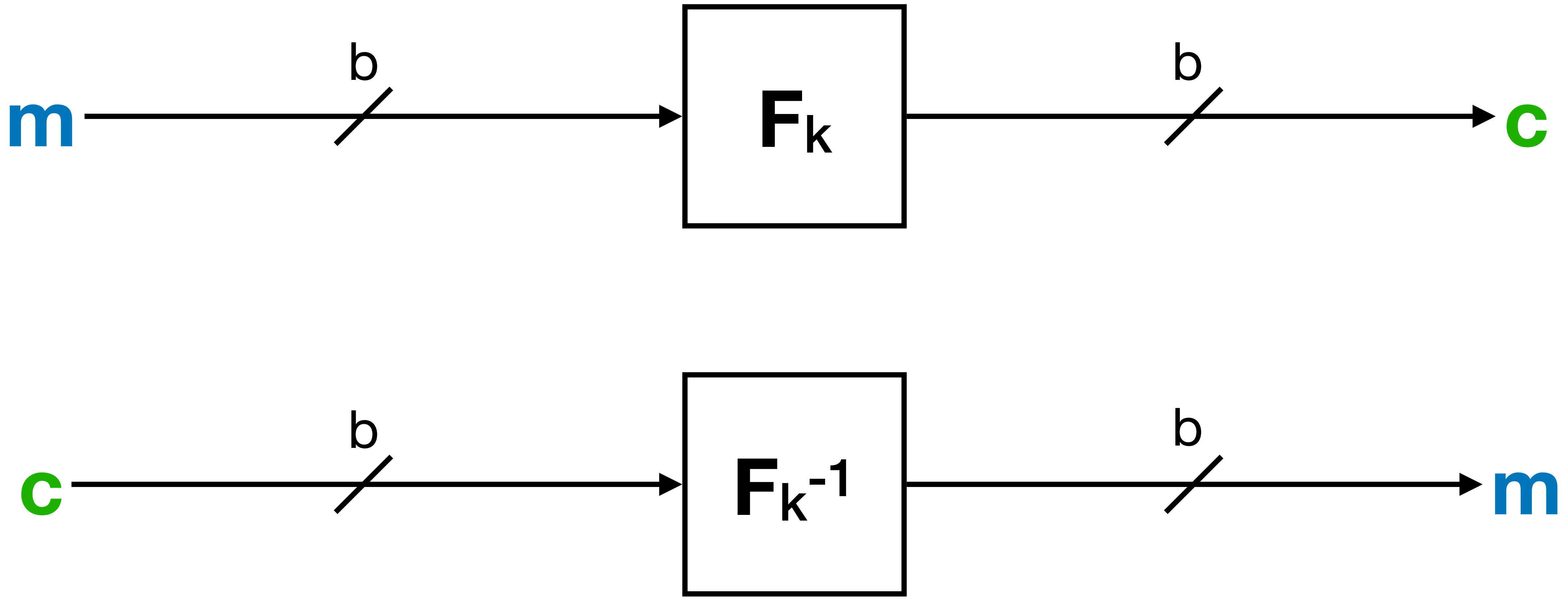
Stream Ciphers



Stream Cipher Cheat Sheet

Bad (do not use)	Good (do use)
RC4	AES-CTR
ISAAC	Salsa20
LFSRs	ChaCha20
MD5 :)	
Your own stream cipher	

Pseudorandom Permutations (PRPs)



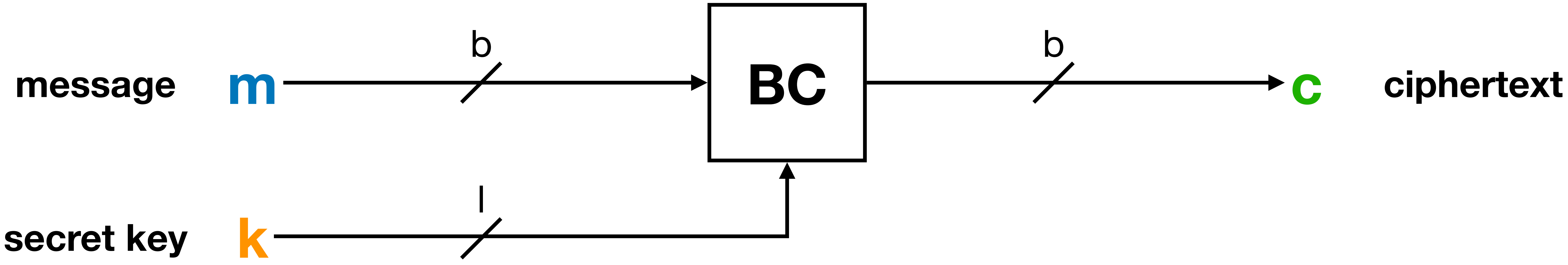
Input space = Output space

such that the function can be inverted:

$c = F_k(m)$, $m = F_k^{-1}(c)$

What's the other word for this?

Block Ciphers



- Given a fixed-size message **m**, and a secret key **k**, compute a uniformly random ciphertext **c** = **BC(m,k)** of the same length as **m.**
- **c** protects the *confidentiality* of **m**.
- BC_k is invertible if you know **k**, i.e., **m** = $\mathbf{BC}^{-1}$(**BC**(**m**,**k**), **k**).
- Given any set of pairs (**m**,**c**) it should be hard to recover **k**.

Block Cipher Cheat Sheet

Bad	Good
DES	**AES**
64-bit block algorithms (3DES, Blowfish, XTEA, etc.)	Many other fine block ciphers, but why not just use AES?
Your own block cipher	

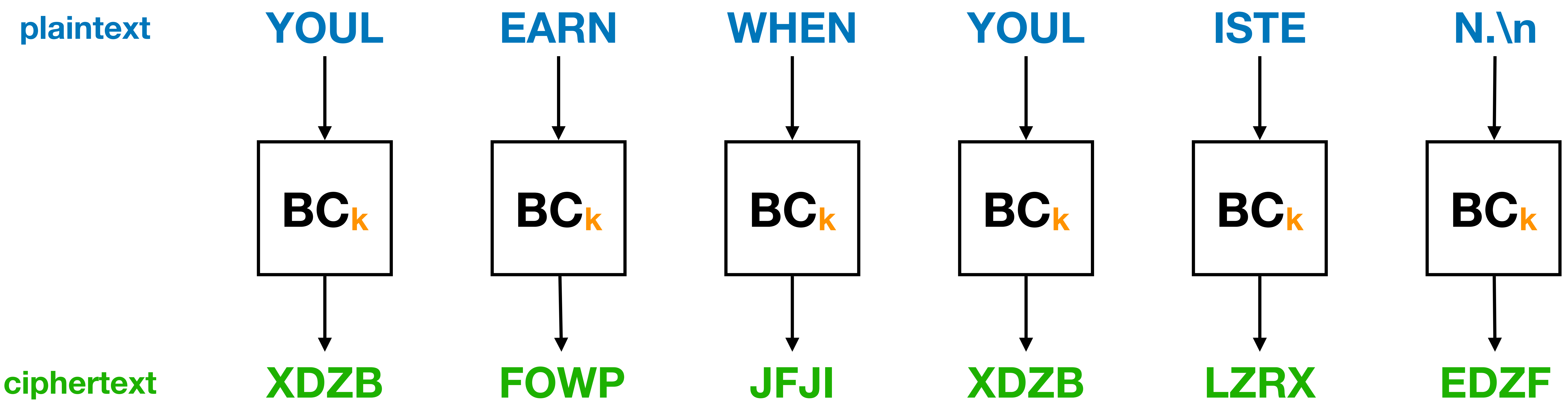
Block Cipher Modes

Or how to encrypt **more than one block**

What modes do you know?

Electronic Codebook (ECB)

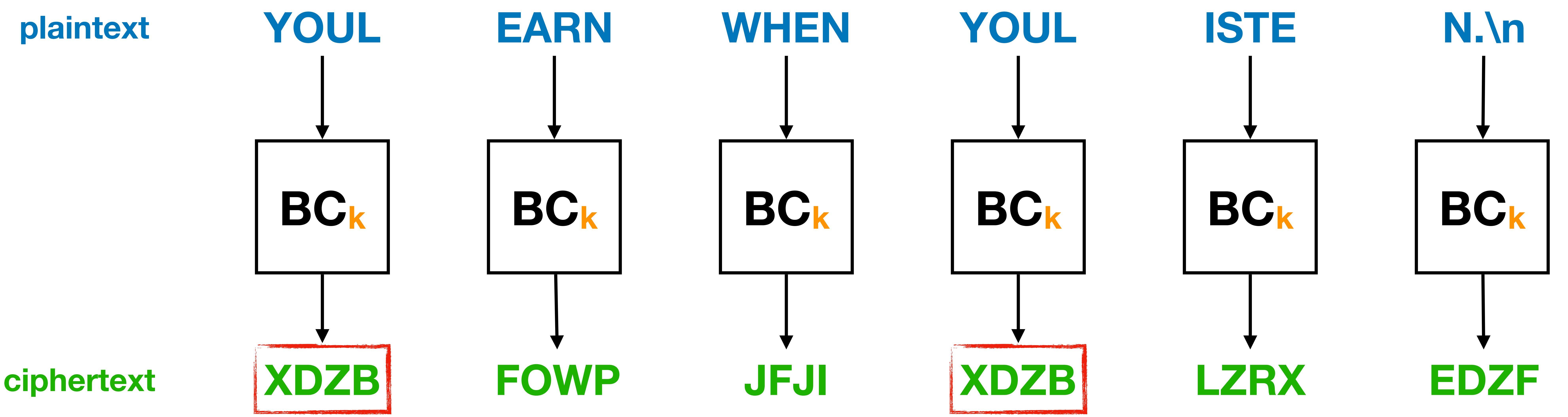
Split plaintext into equal-size blocks, then encrypt block-by-block

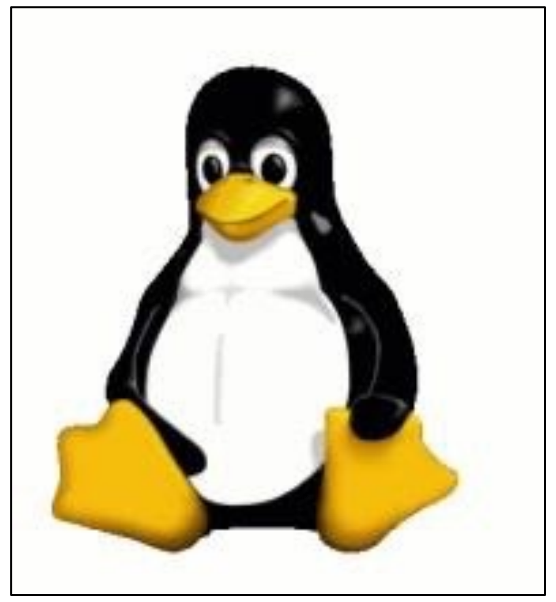


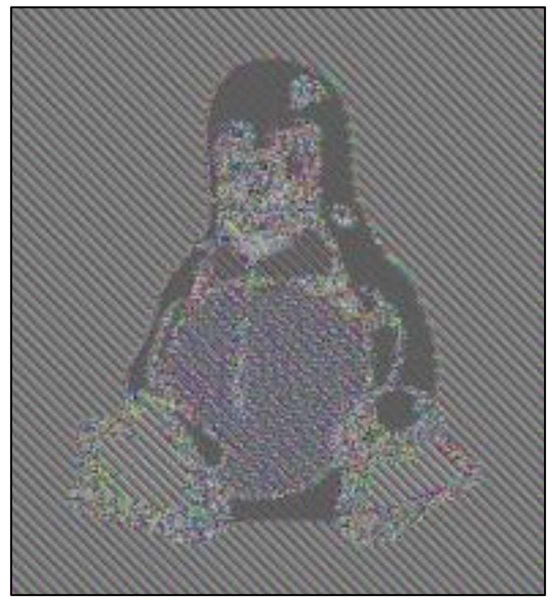
What’s the problem?

Electronic Codebook (ECB)

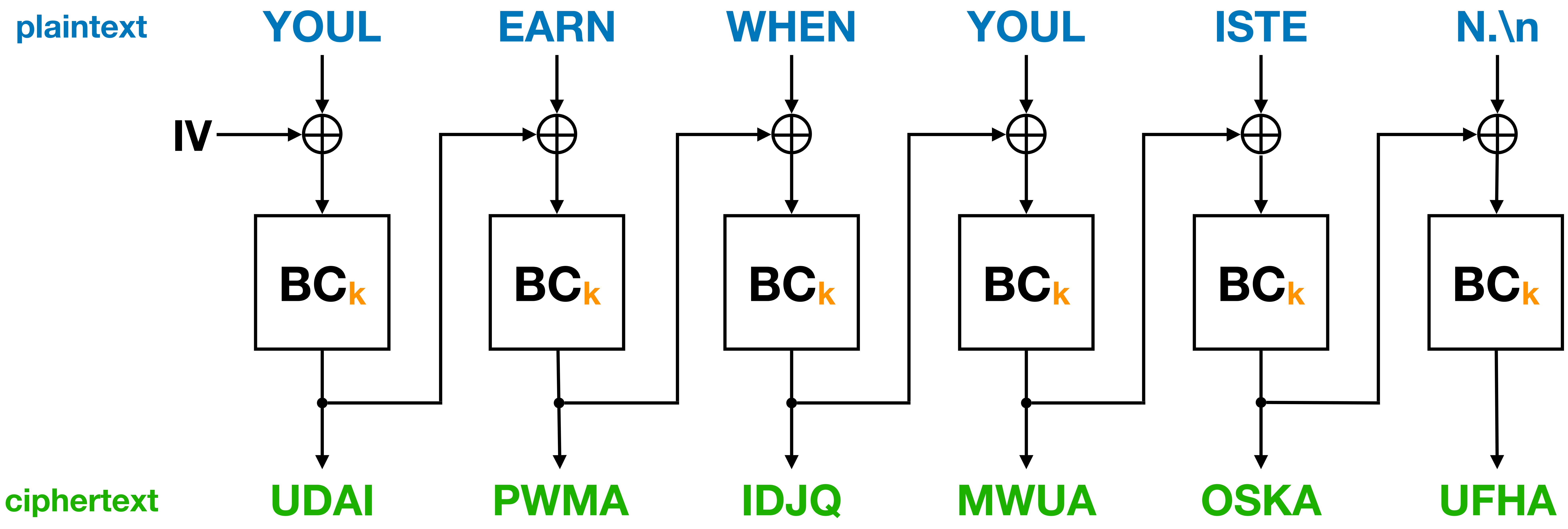
Problem: Identical plaintext blocks become identical ciphertext blocks







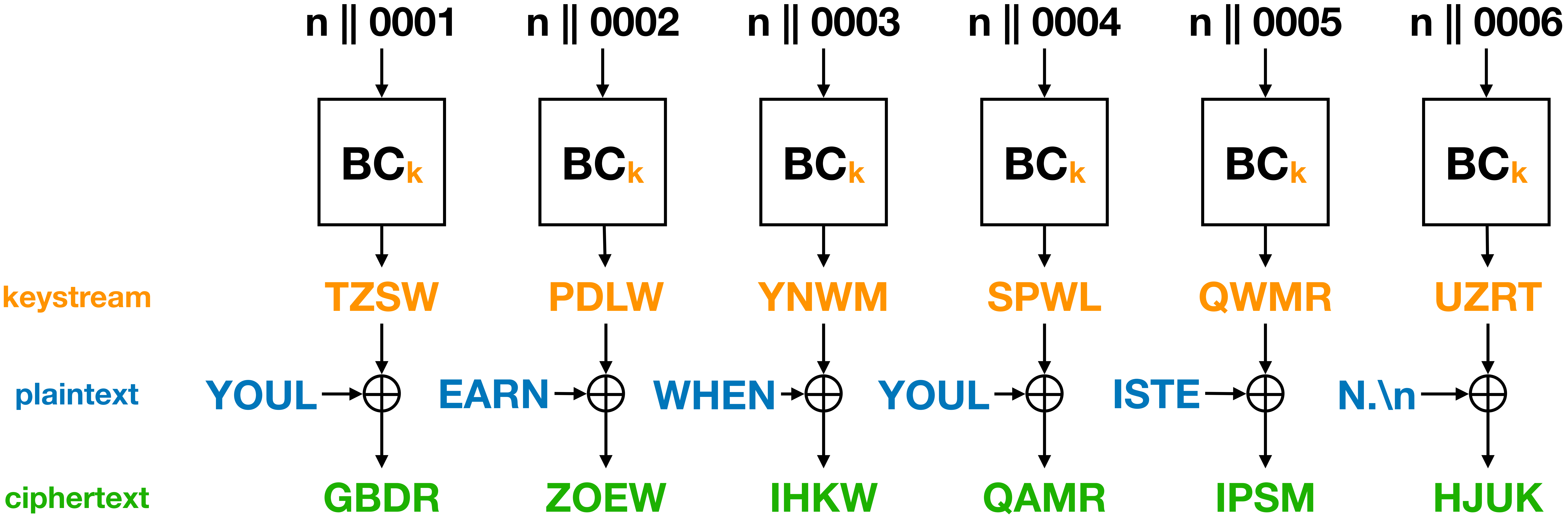
Cipher Block Chaining (CBC)



Secure if IV's are random (spoiler: unless padding oracle attacks are possible)

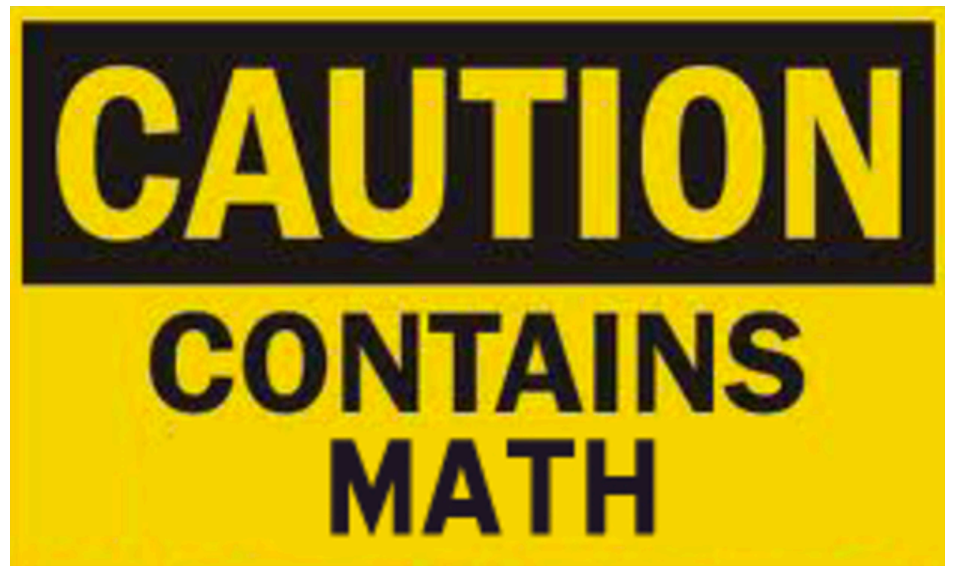
Counter (CTR)

Encrypt nonce || counter to produce a keystream, then use it as a stream cipher!



The **nonce must be unique** for each new message, insecure otherwise

Why CTR Needs Unique Nonces



Say you encrypt P_1 into C_1, and P_2 into C_2 with the same nonce

Same nonce = same keystream S, therefore

$P_1 = C_1 \oplus S$, that is: $S = C_1 \oplus P_1$

$P_2 = C_2 \oplus S$, thus $P_2 = C_2 \oplus C_1 \oplus P_1$

An attacker who knows P_1, C_1, and C_2 can thus recover P_2

An attacker who only knows and P_1, C_1 can recover the value $C_2 \oplus P_2$

Block Cipher Fail: Tarsnap (2011)

The following patch introduced a critical security vulnerability.

Do you see the problem?

```
        /* Encrypt the data. */
-       aes_ctr(&encr_aes->key, encr_aes->nonce++, buf, len,
-           filebuf + CRYPTO_FILE_HLEN);
+       if ((stream =
+           crypto_aesctr_init(&encr_aes->key, encr_aes->nonce)) == NULL)
+               goto err0;
+       crypto_aesctr_stream(stream, buf, filebuf + CRYPTO_FILE_HLEN, len);
+       crypto_aesctr_free(stream);
```

Block Cipher Fail: Tarsnap (2011)

What happened: Nonce reuse in AES-CTR ("++" forgotten)

```
        /* Encrypt the data. */
-       aes_ctr(&encr_aes->key, encr_aes->nonce++, buf, len,
-           filebuf + CRYPTO_FILE_HLEN);
+       if ((stream =
+           crypto_aesctr_init(&encr_aes->key, encr_aes->nonce)) == NULL)
+               goto err0;
+       crypto_aesctr_stream(stream, buf, filebuf + CRYPTO_FILE_HLEN, len);
+       crypto_aesctr_free(stream);
```

Block Cipher Fail: Tarsnap (2011)

The bug

Tarsnap archives data by first converting it into a series of "chunks" of average size 64 kB; next compressing and encrypting each chunk; and finally uploading those chunks. The encryption is performed using a per-session AES-256 key in CTR mode.

In versions 1.0.22 through 1.0.27 of Tarsnap, the CTR nonce value is not incremented after each chunk is encrypted. (The CTR counter is correctly incremented after each 16 bytes of data was processed, but this counter is reset to zero for each new chunk.)

How the bug happened

Up to version 1.0.21 of Tarsnap, AES-CTR was used in two places: First, to encrypt each chunk of data; and second, in the Tarsnap client-server protocol. In version 1.0.22 of Tarsnap, I introduced passphrase-protected key files, which used AES-CTR encryption (with a key computed using scrypt).

In order to simplify the Tarsnap code — and in the hopes of reducing the potential for bugs — I took this opportunity to "refactor" the AES-CTR code into a new file (lib/crypto/crypto_aesctr.c in the Tarsnap source code) and modified the existing places where AES-CTR was used to take advantage of these routines.

Counter (CTR)

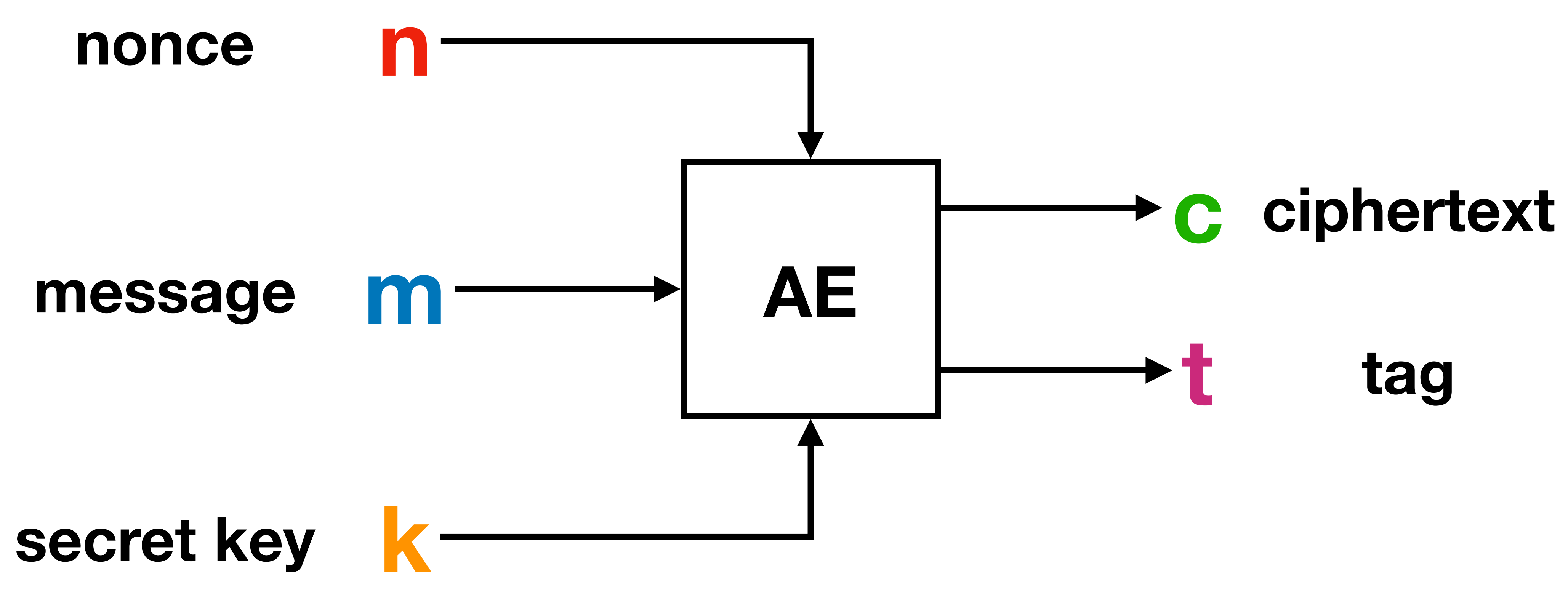
Advantages:

- **Fast** in software (thanks to pipelined processing, keystream precomputation)
- **Ciphertext length = plaintext length** ("ciphertext stealing" is a trick for CBC)
- Needs **only block cipher encryption**

Disadvantages:

- **Nonce reuse** can expose full plaintext independent of the used block cipher (more dangerous than IV reuse in CBC)

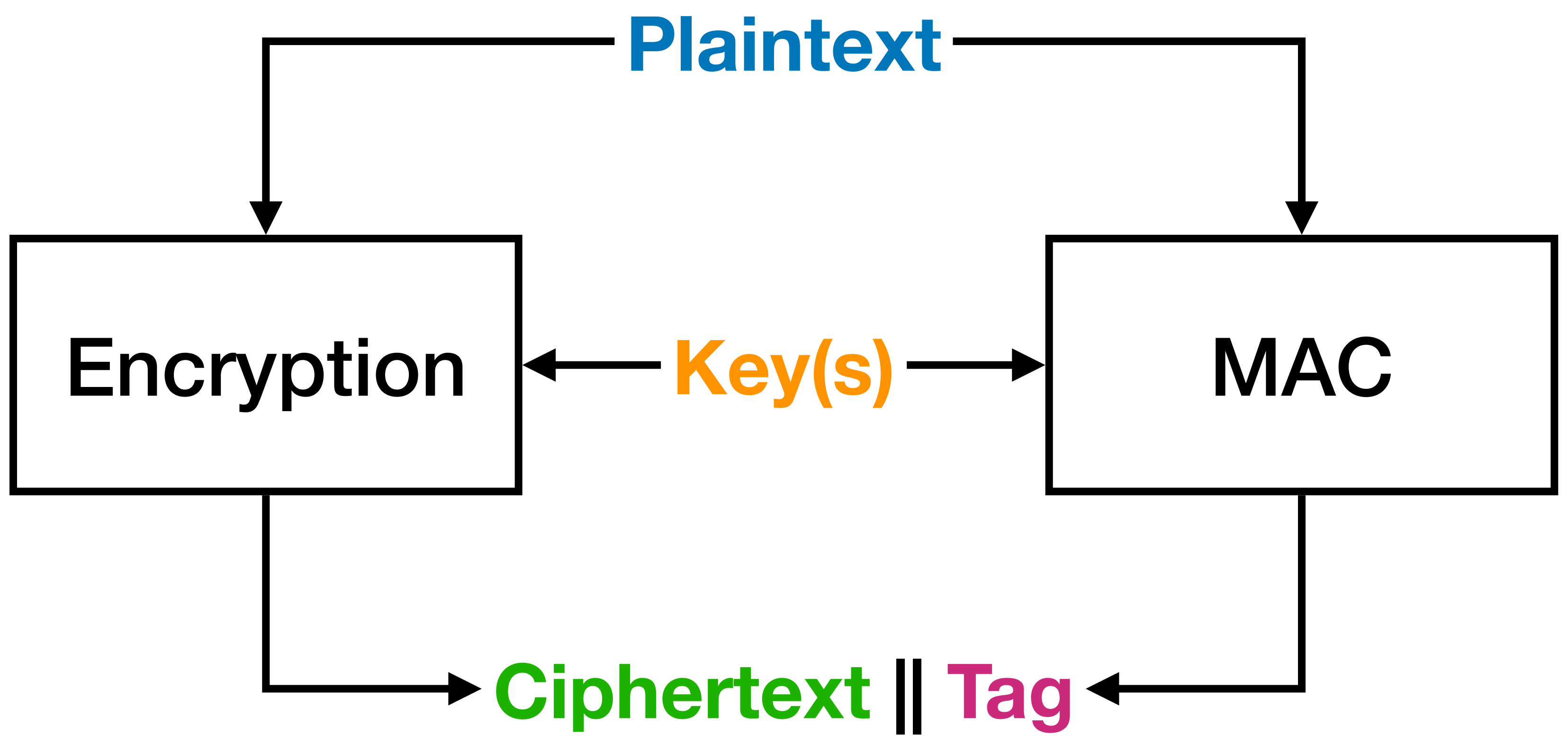
Authenticated Encryption (AE)



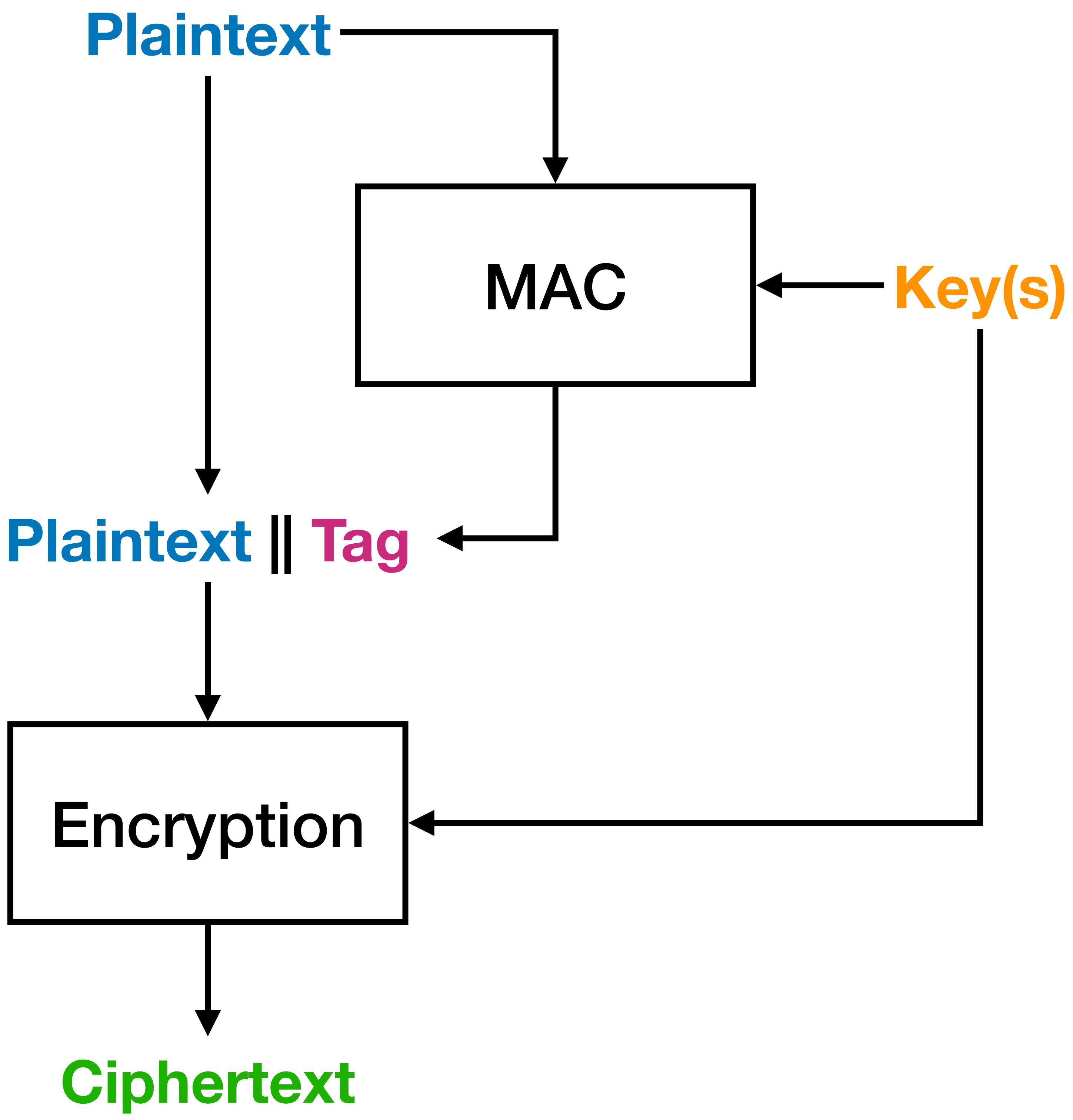
- *Confidentiality* of **m** ensured by ciphertext **c**.
- *Authenticity* of **m** ensured by tag **t**.
- AEAD = AE with *associated data* (authenticated but not encrypted)

AE with a Cipher and a MAC

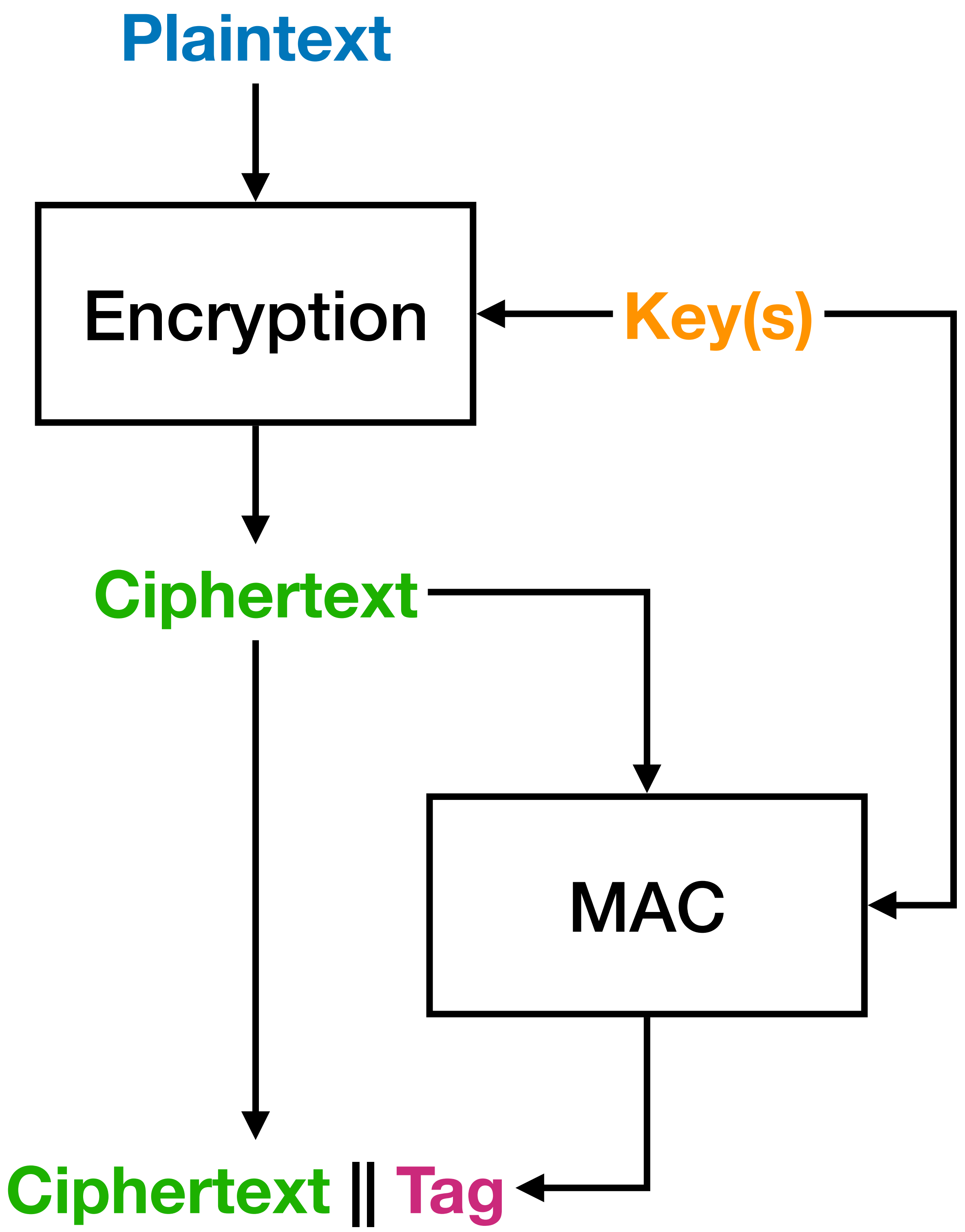
Encrypt-and-MAC
(as in SSH originally)



MAC-Then-Encrypt
(as in < TLS 1.3)



Encrypt-Then-MAC
(as in IPSec)

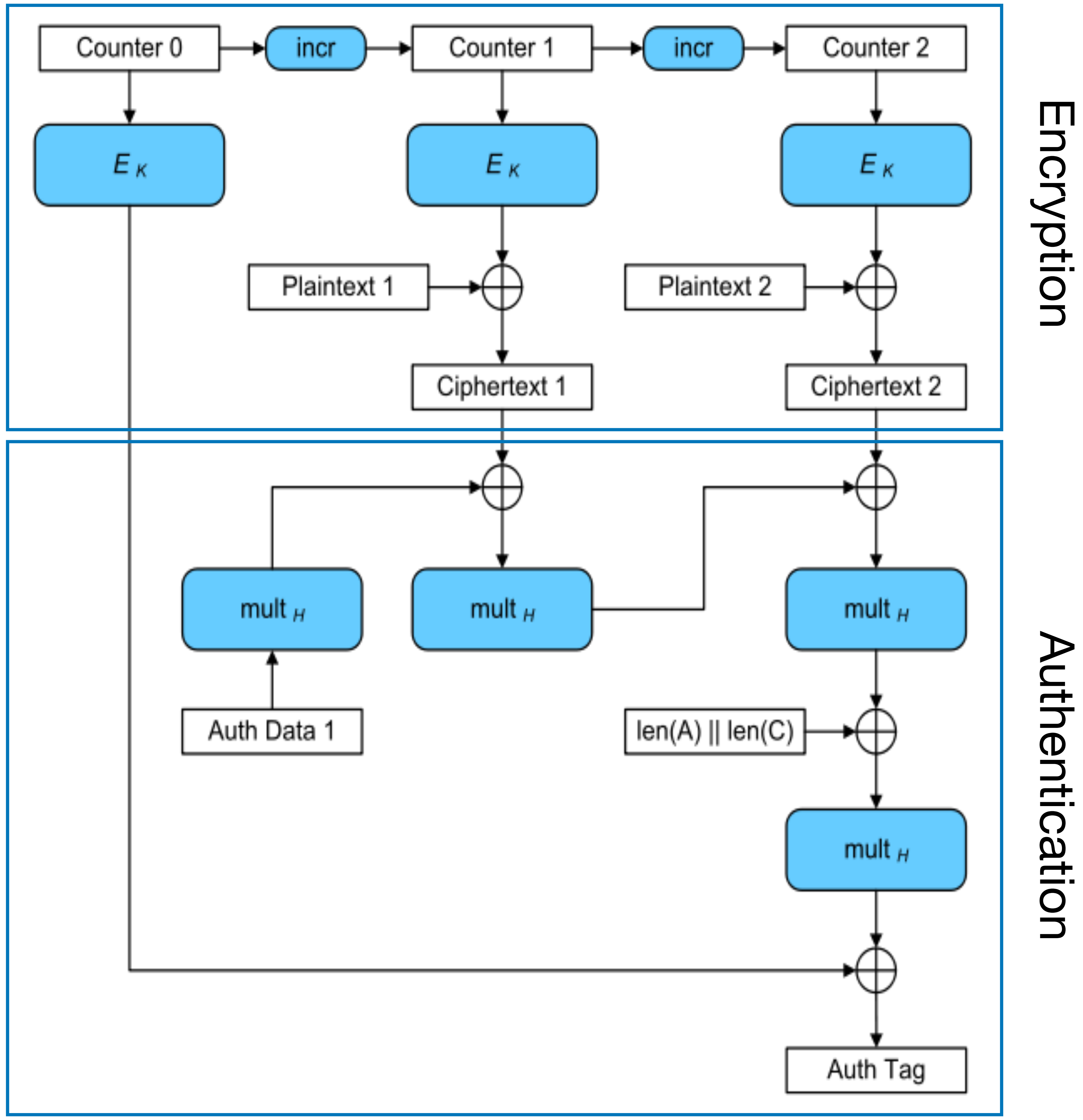


Exercise:
What are the pros and cons
of these approaches?

The AEAD Standard: AES-GCM

AES in Galois Counter Mode

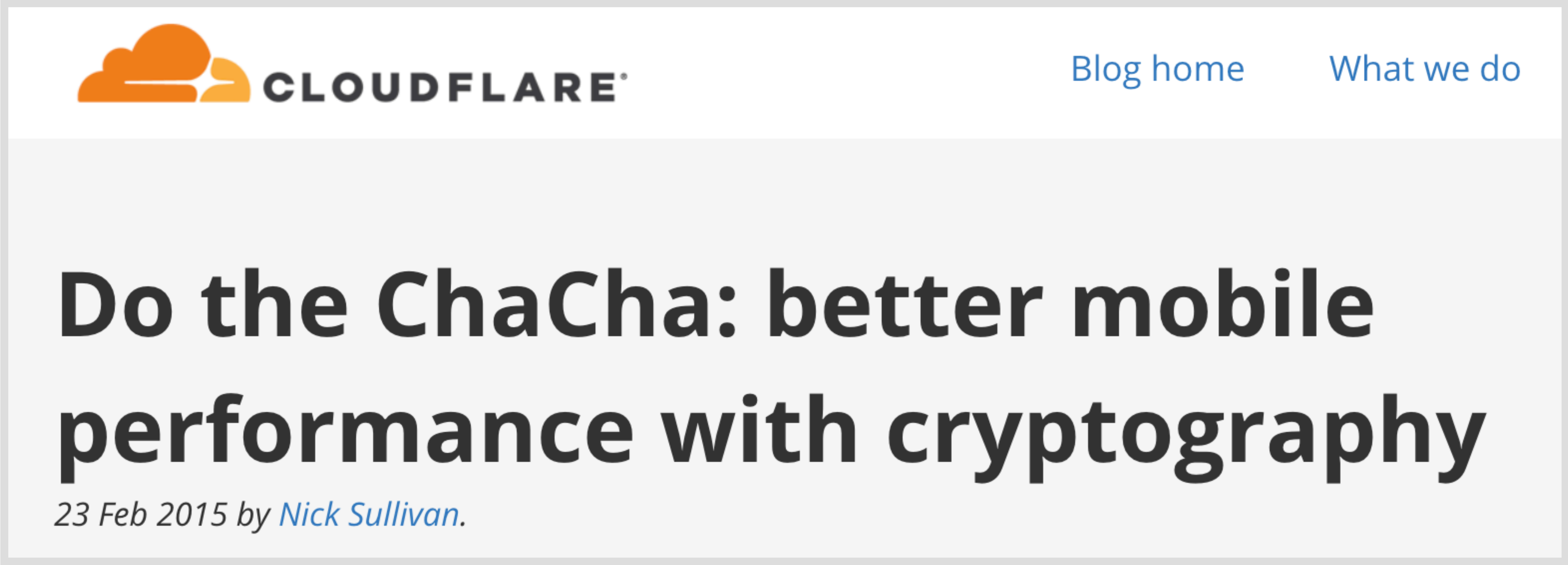
- Standardized in **NIST SP 800-38D**
- Basically: **CTR & 128-bit accumulator** (to compute auth. tag)
- Supported in IPSec, SSH, TLS{1.2,1.3}
- 128-bit carryless multiplication
  - No carry enables parallelizable bit ops.
  - PCLMULQDQ instruction
  - ~2.4 GiB/sec @ 3GHz on recent Intel



Less of a Standard: ChaCha-Poly

djb's **ChaCha** is a cousin of the Salsa20 stream cipher
djb's **Poly1305** is a one-time MAC based on a universal hash

- RFC 7905, 7539 , TLS_**CHACHA20_POLY1305**_SHA256 in TLS 1.3
- Only "ARX" operations and simple arithmetic, no CPU-specific feature required to make it fast (unlike AES-based modes), SIMD-friendly
- https://cr.yp.to/chacha.html



CAESAR Ciphers

Competition for Authenticated Encryption: Security, Applicability, and Robustness

2012-2019 research project

Academically vetted AEAD algorithms, not (yet?) standardised

AEGIS in the Linux kernel, supported for dm-crypt disk encryption

Performance benchmarks on https://bench.cr.yp.to/

Final portfolio

The final CAESAR portfolio is organized into three use cases:

- 1: Lightweight applications (resource constrained environments)
- 2: High-performance applications
- 3: Defense in depth

Final portfolio for use case 1 (first choice followed by second choice):

candidate	designers
Ascon, first choice for use case 1: home v1 v1.1 **v1.2**	Christoph Dobraunig, Maria Eichlseder, Floria
ACORN, second choice for use case 1: v1 v2 **v3**	Hongjun Wu

Final portfolio for use case 2 (alphabetical order, without a preference):

candidate	designers
AEGIS-128 for use case 2: v1 **v1.1**	Hongjun Wu, Bart Preneel
OCB for use case 2: v1 **v1.1**	Ted Krovetz, Phillip Rogaway

https://competitions.cr.yp.to/caesar.html

crypto attacks & defenses RELOADED

JP Aumasson, Philipp Jovanovic

ringzerø

symmetric crypto ✓