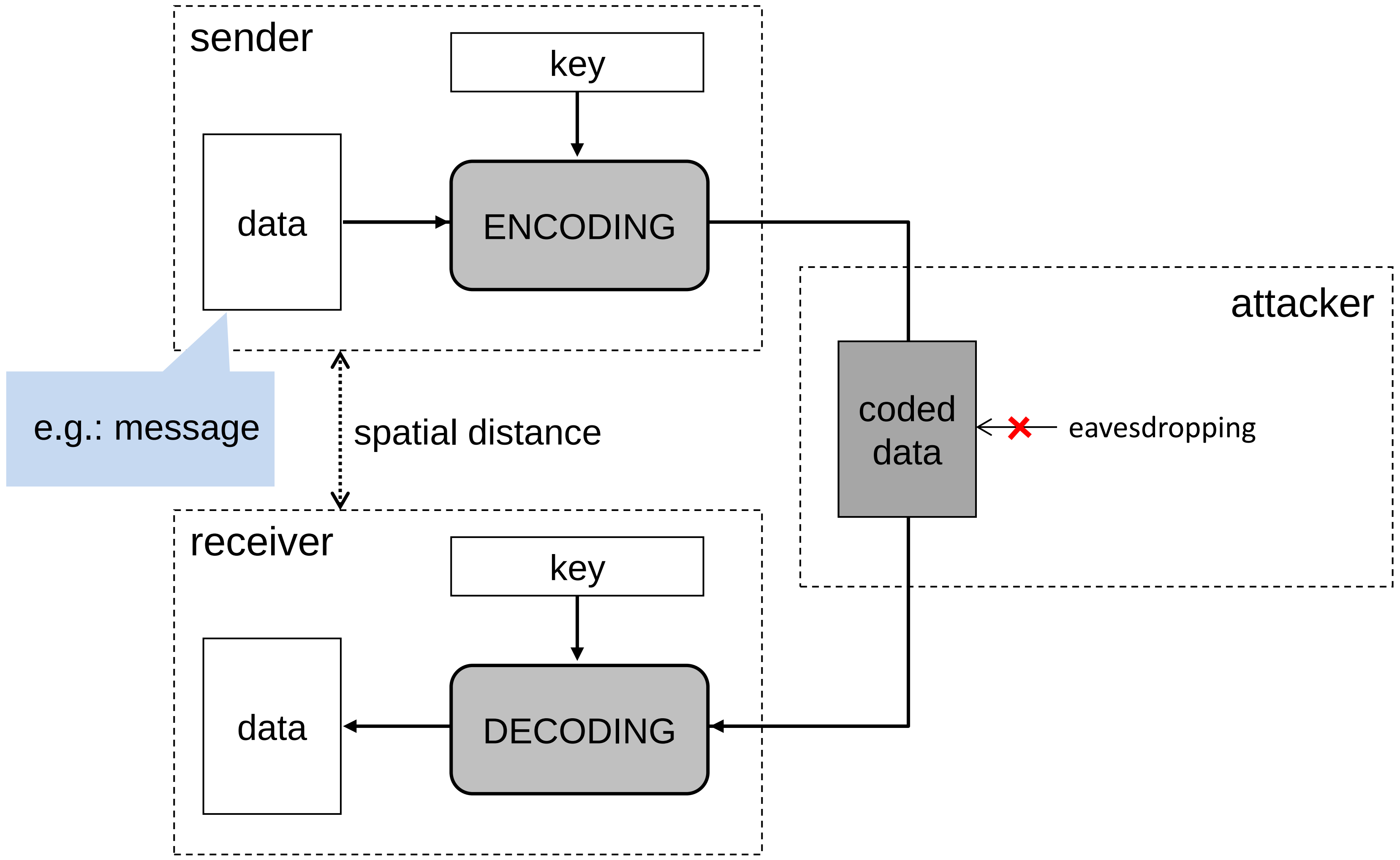
Summary on Crypto Primitives and Protocols

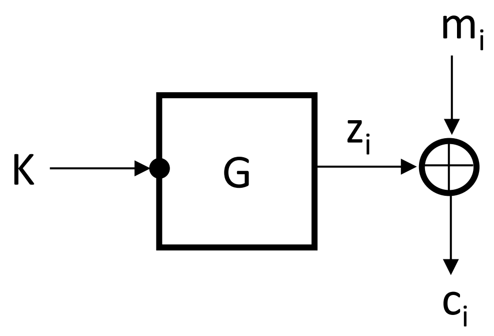
Levente Buttyán
CrySyS Lab, BME
www.crysys.hu

Basic model of cryptography

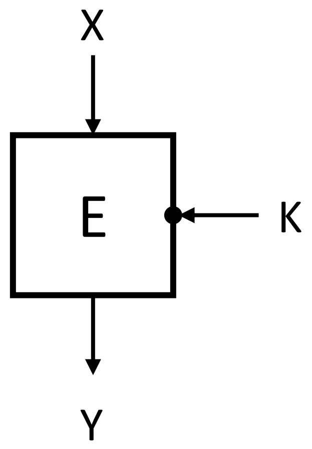


Symmetric key (conventional) encryption

- stream ciphers
  - XOR (+ or ⊕)
  - one-time pad (truely random key stream)
  - stream ciphers (pseudo random key stream)
    - » large size of the effective state space is important
    - » do not provide any integrity protection
    - » does not increase message length



- block ciphers
  - operate on larger blocks (typical size is 128 bits)
  - can be viewed as random permutations
  - product ciphers use simple operations in many rounds
  - AES (block size: 128; key size = 128, 192, 256)

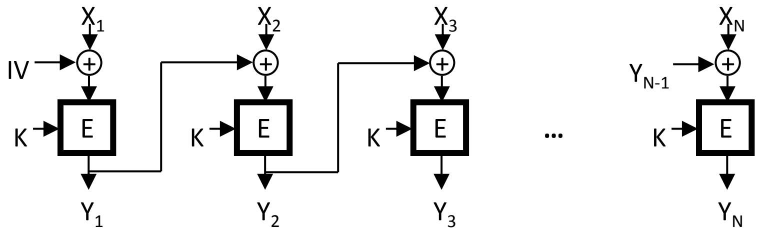


Attacks

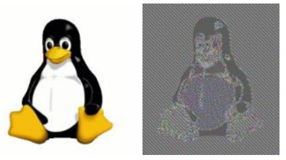
- Kerckhoff’s principle
  - it is assumed that the encryption algorithm is known to the attacker
- attack models
  - ciphertext-only attack
  - known-plaintext attack
  - (adaptive) chosen-plaintext attack
  - (adaptive) chosen-ciphertext attack
- exhaustive key search attack
  - average complexity is 2^{k-1}, if key length is k bits
- algebraic attacks
  - weaknesses in the algebraic structure of a cipher may lead to attacks that are *substantially* more efficient than the exhaustive key search attack

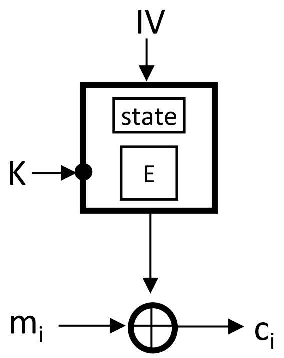
Block cipher modes

- Cipher Block Chaining (CBC)
  - IV: unpredictable, non-manipulable
  - padding
  - padding oracle attack



- Electronic Code Book (ECB)





- Cipher Feedback (CFB)
- Output Feedback (OFB)
- Counter (CTR)

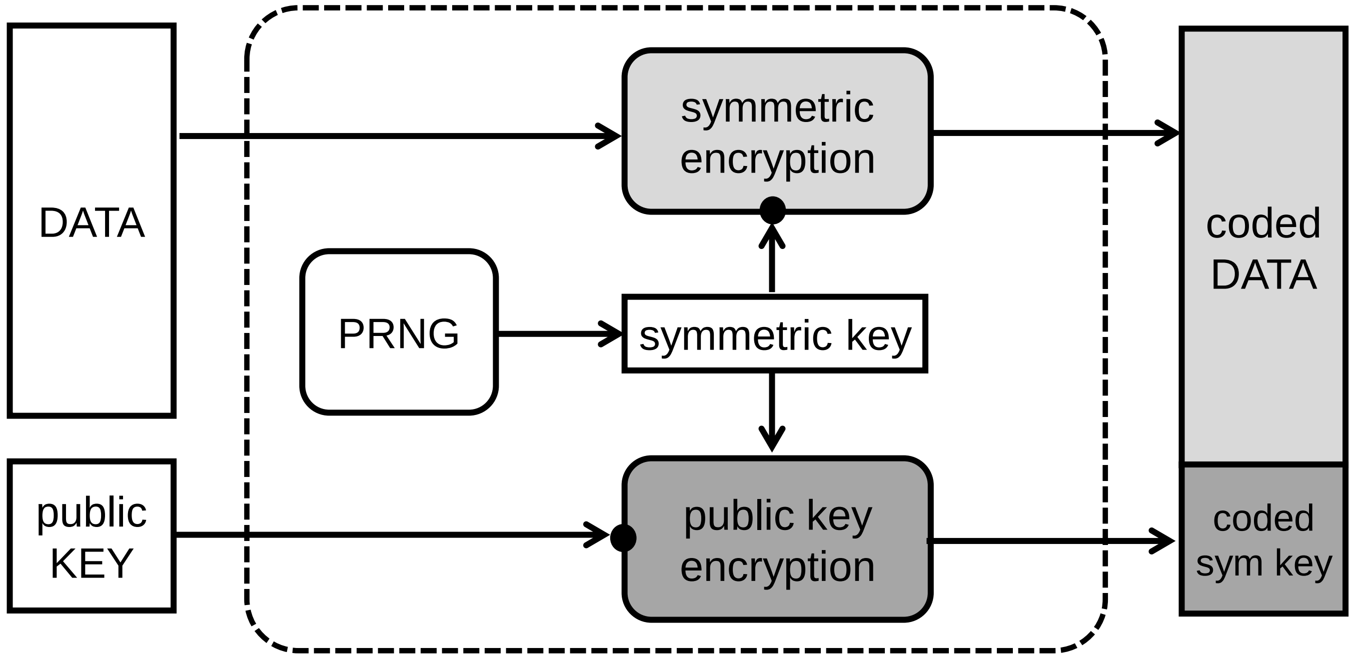
- special modes:
  - CBC-CTS (ciphertext stealing)
- authenticated encryption modes:
  - CCM, GCM, OCB

Asymmetric (public) key encryption

- encryption and decryption are performed with different keys
- in fact, the key has two parts:
  - one part is used for encryption; this can even be **public**
  - the other part is used for decryption; this must be kept **private**
  - computing the private part from the public part is hard
- only the public key needs to be transmitted to the recipient, and this does not need a secure channel
- there is no need to have shared secret between sender and recipient → this makes key management easier
- example: RSA

Hybrid encryption

- public key crypto is slower than symmetric key crypto and require longer (e.g. 2048 bits) keys for similar security
- the speed problem can be solved with hybrid encryption:



Security of public key crypto schemes

- security is usually related to the difficulty of some problems that are widely believed to be hard to solve

  <u>examples:</u>
  - factoring
  - computing discrete logarithm

- sometimes it can even be rigorously proven that breaking the encryption scheme would mean that there exist an efficient solution to the related hard problem (reduction proof)
  - although widely used practical schemes have no complete proofs

- practical considerations:
  - semantic security
  - non-malleability

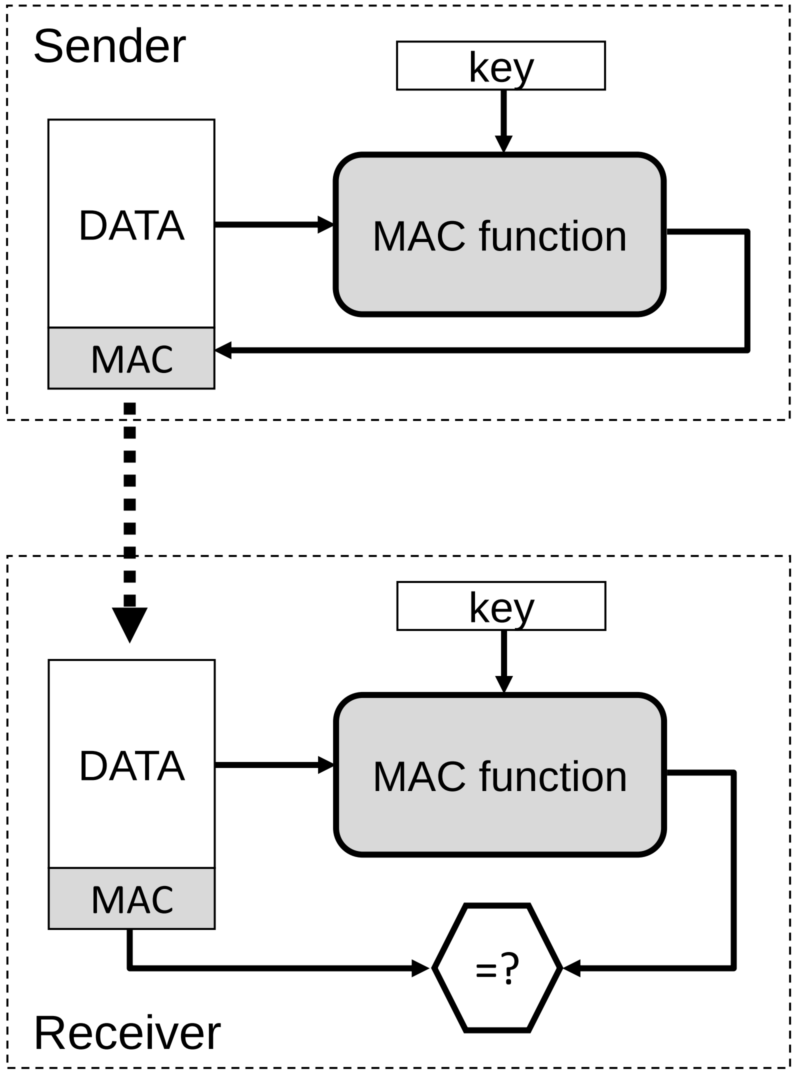
Other primitives

- cryptographic hash functions
  - map arbitrary long inputs into a fixed length output (digest, hash value)
  - three important properties:
    1. collision resistance
    2. weak collision resistance (2nd preimage resistance)
    3. one-wayness (preimage resistance)
  - birthday paradox → complexity of brute force collision search is $2^{n/2}$

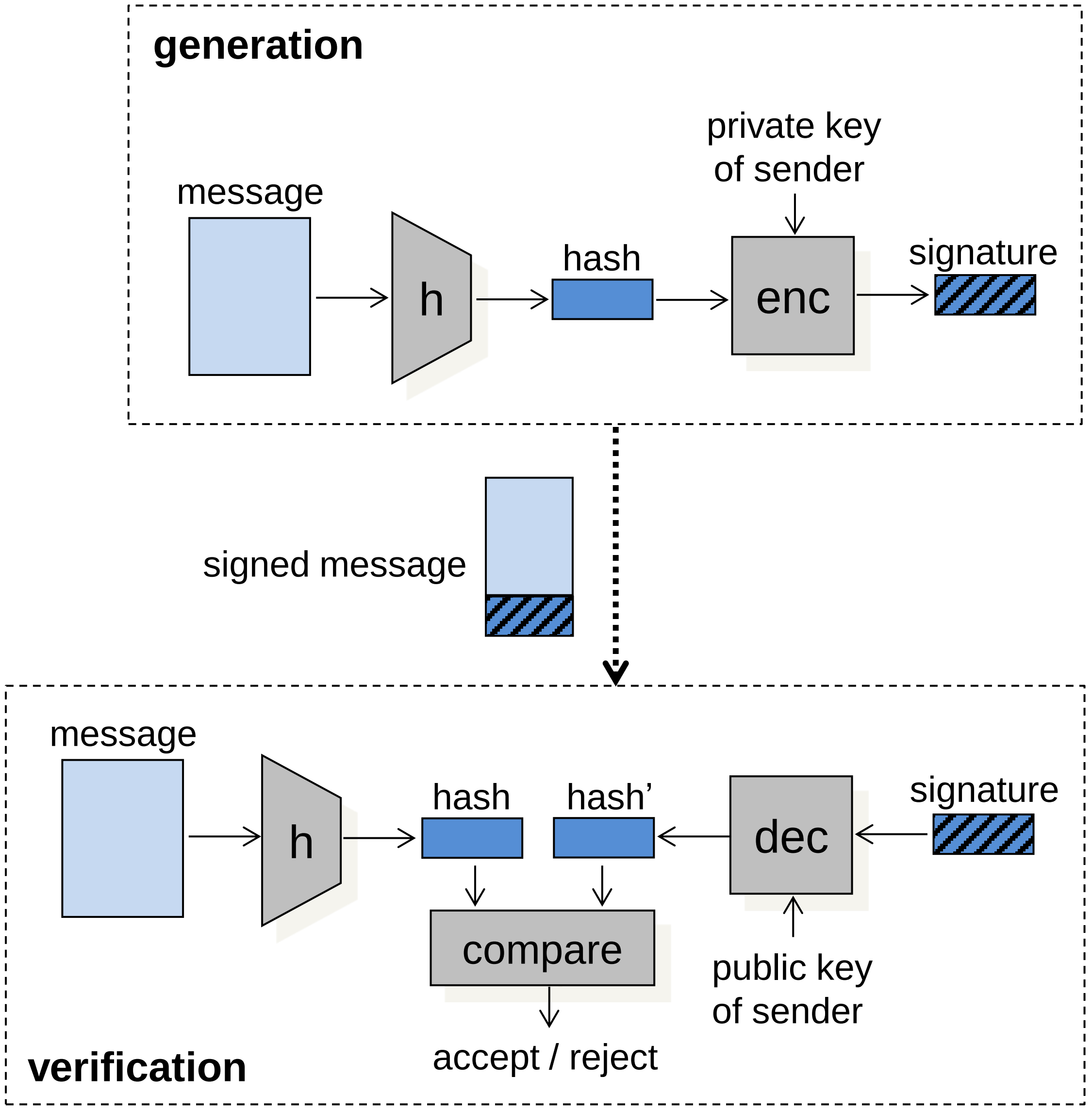
- MAC functions
  - similar to hash functions, but have an additional input (a symmetric key)
  - used for message integrity protection and message origin authentication

- digital signature schemes
  - similar to MAC functions, but use asymmetric keys
  - besides integrity protection and origin authentication, they also ensure non-repudiation

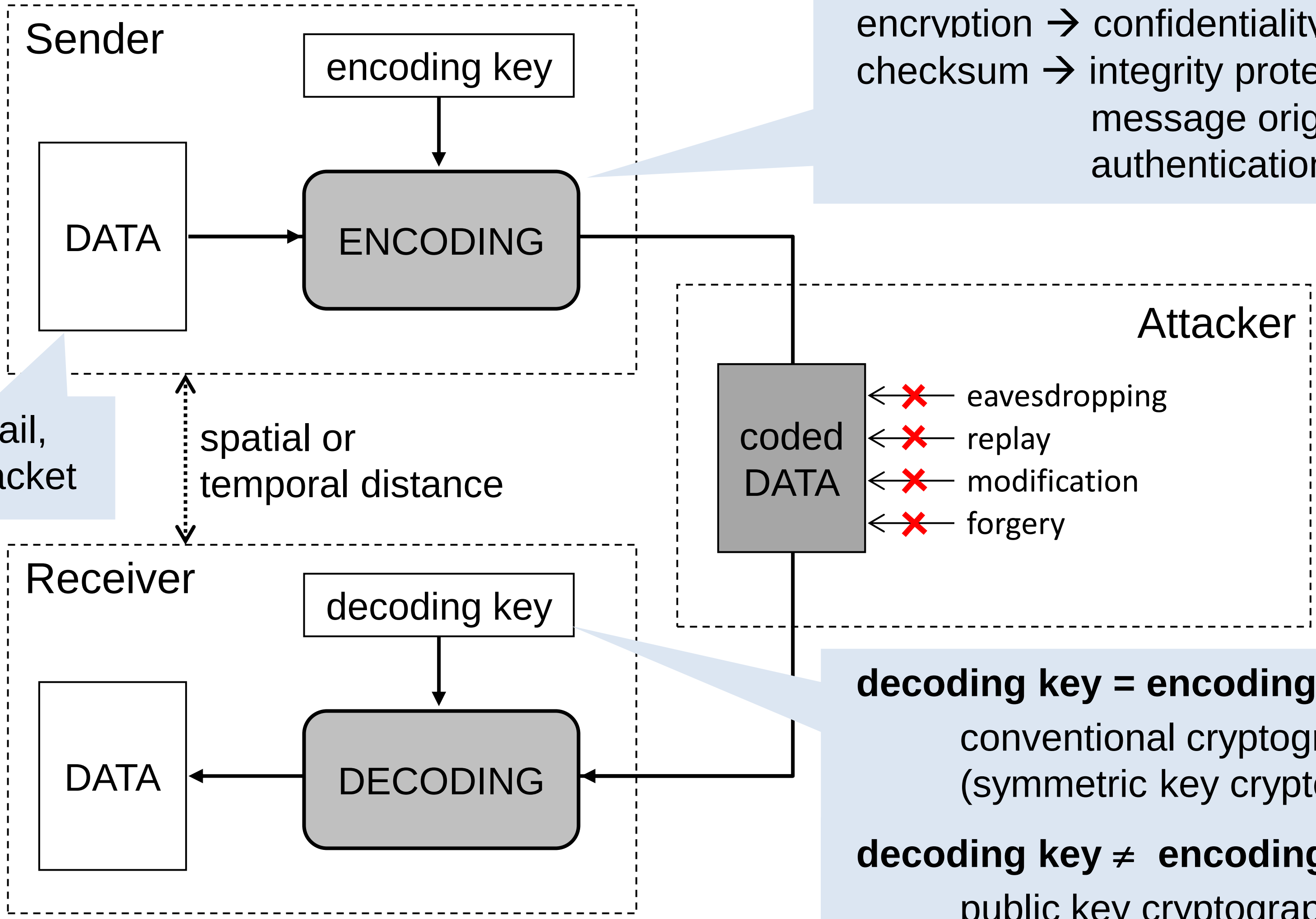
Using a MAC function



Hash-and-sign approach



General model of cryptographic coding



encryption → confidentiality
checksum → integrity protection, message origin authentication

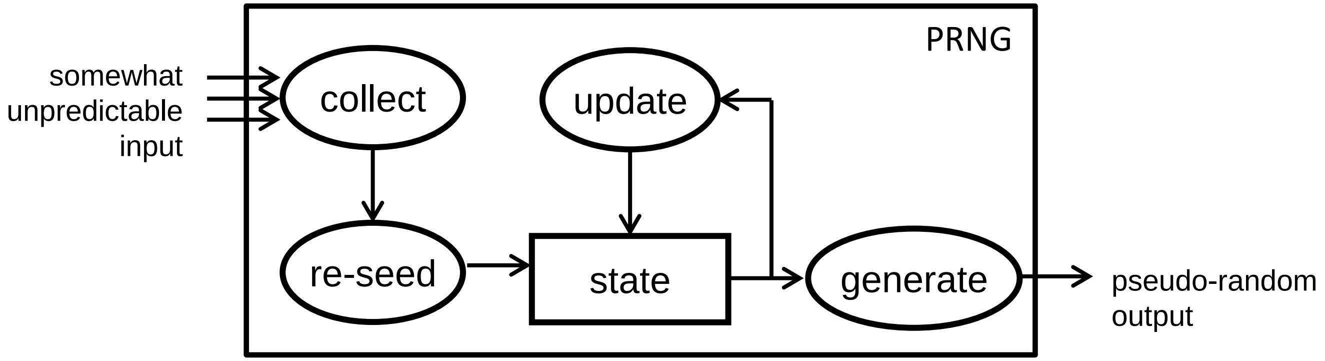
e.g.: e-mail, file, IP packet

decoding key = encoding key
conventional cryptography
(symmetric key crypto)

decoding key ≠ encoding key
public key cryptography
(asymmetric key crypto)

Support functions

- random number generation
  - a (cryptographic) **random number** is a number that cannot be predicted with better probability than random coin flips (even if all previous outputs have been observed)
  - a **pseudo-random number generator (PRNG)** processes somewhat unpredictable inputs and generates pseudo-random outputs (look very similar to real random numbers)



- design of PRNGs
  - where do you get real random input?
  - when and how do you re-generate the internal state of your PRNG?
  - how do you generate the output?
  - attacker models (only some outputs can be observed ... state compromise extension)
  - example: Fortuna

Support functions

- key exchange protocols
  - allow two remote parties to setup a shared key when needed
  - attacker model:
    - » attacker controls the communication channel (Man-in-the-Middle)
    - » cannot break crypto primitives
    - » eavesdropping, injection, replay, reflection, interleaving, typing attacks
  - security requirements:
    - » key authentication (implicit or explicit) ← cryptographic protection
    - » key freshness ← timestamps, nonces, key agreement
  - main design principle:
    - » make sure that interpretation of messages does not implicitely depend on context
  - classification
    - » key transport (using only symmetroc key crypto or using public key crypto)
    - » key agreement

WiFi security

- security challenges in wireless networks
  - no physical protection of communication channels
  - broadcast nature of communications
- WEP
  - operation (station authentication, message integrity and confidentiality)
  - WEP flaws, possible attacks, lessons learned
- WPA, WPA2
  - new authentication framework (802.1X, EAP, key hierarchy)
  - TKIP (WPA) (design constraints and weaknesses)
  - AES-CCM (WPA2)

TLS

- TLS subprotocols and their functions

- TLS Record Protocol
  - uses strong algorithms (HMAC, AES)
  - good protection against passive eavesdropping
  - some protection against traffic analysis (random length padding)
  - vulnerable to some padding oracle attacks (Lucky 13, POODLE)

- TLS Handshake Protocol
  - uses strong algorithms (RSA, DH, DSS, well designed PRF based on HMAC)
  - some protection against passive and active attacks
  - possibly vulnerable to some active attacks (e.g., DROWN)

- attacks on SSL/TLS
  - CBC padding oracle attack and variants (e.g., Lucky 13, POODLE)
  - CBC predictible IV vulnerability (BEAST)
  - attacks exploiting compression ratio and timing side channels (BREACH, TIME, CRIME)
  - cross-protocol attacks (DROWN)

CRYPTOGRAPHY: THE STRONGEST LINK IN THE CHAIN

LEVENTE BUTTYÁN AND BOLDIZSÁR BENCSÁTH

IT security architectures that use cryptographic elements sometimes fail, but it is rarely cryptography to blame. The reason is more often the use of cryptography in an inappropriate way, or the use of algorithms that do not really qualify as cryptographic. High quality cryptography is in fact the strongest link in the chain, and there are good reasons for that.

What goes wrong in practice?

- key management issues
  - e.g., keys are generated with weak random number generators
- protocol weaknesses
  - e.g., crypto algorithms are used in wrong ways
- implementation issues
  - bugs
  - side channels (e.g., timing attacks)
- human stupidity
  - e.g., using home made ”crypto” algortihms

EXAMPLES

Early version of Netscape's PRNG

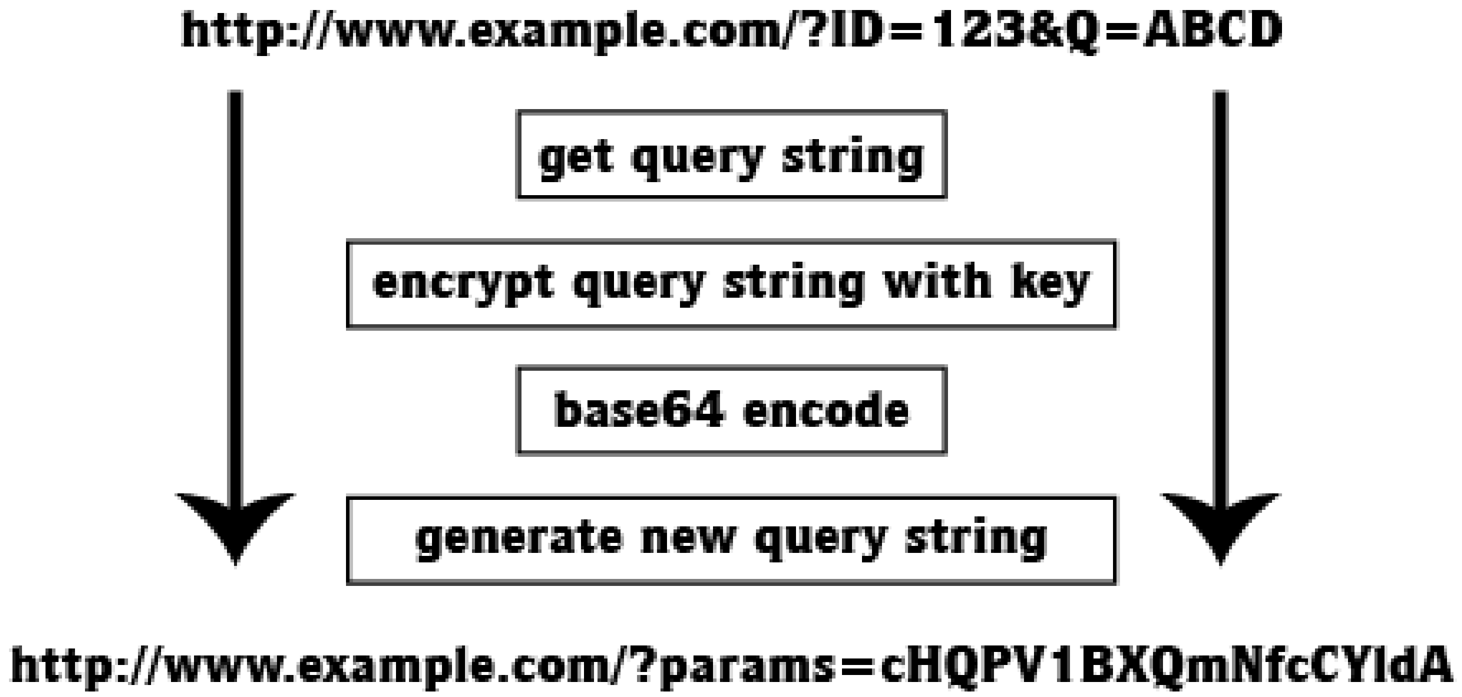
```
RNG_CreateContext()
    (seconds, microseconds) = time of day;
    pid = process ID; ppid = parent process ID;
    a = mklcpr(microseconds);
    b = mklcpr(pid + seconds + (ppid << 12) );
    seed = MD5(a|b);

mklcpr(x)
    return((0xDEECE66D*x + 0x2BBB62DC) >> 1)

RNG_GenerateRandomBytes()
    x = MD5(seed);
    seed = seed+1;
    return x;

create_key()
    RNG_CreateContext();
    RNG_GenerateRandomBytes(); RNG_GenerateRandomBytes();
    client_random = RNG_GenerateRandomBytes(); // sent in client_hello
    pre_master_secret =  RNG_GenerateRandomBytes();
```

Attacking the Netscape PRNG

- if an attacker has an account on the UNIX machine running the browser
  - *ps* command lists running processes → attacker learns pid, ppid
  - the attacker can guess the time of day with seconds precision (assumption)
  - only unknown is the value of microseconds → ~2^{20} possibilities
  - each possibility can be tested easily against the client_random sent in clear in the client_hello message
- if the attacker has no account on the machine running the browser
  - a has 20 bits of randomness, b has 27 bits of randomness → seed has 47 bits of randomness (compared to 128 bit advertised security)
  - ppid is often just a bit smaller than pid
  - sendmail generates message IDs from its pid
    - » send mail to an unknown user on the attacked machine
    - » mail will bounce back with a message ID generated by sendmail
    - » attacker learns the last process ID generated on the attacked machine
    - » this may reduce possibilities for pid

secureURL.php

- Nguyen Quoc Bao, Secure URL 2.0, www.phpclasses.org/quocbao_secureurl
- designed to hide URL parameters and protect their integrity


http://www.example.com/?ID=123&Q=ABCD
get query string
encrypt query string with key
base64 encode
generate new query string
http://www.example.com/?params=cHQPV1BXQmNfcCYIdA

Breaking secureURL.php

- we analyzed this package in the context of a penetration testing work that we conducted for request by a client
- the client's web site used the secureURL.php package for hiding URL parameters
- the entire design of the site's defense architecture heavily depended on the assumption that URL parameters were properly hidden
- we broke the "cryptographic" algorithm of secureURL.php, and this also allowed us to successfully break into the client's system (with some additional work, of course)
- in this case, the use of bad cryptography created a false impression of security for our client
- we reported the flaw at http://seclists.org/bugtraq/2011/Sep/139

Breaking secureURL.php

- encryption is based on XOR-ing the plaintext parameter string with the MD5 digest of a user defined secret key
  - repeated as many times as needed to mask the entire plaintext parameter string
- the same MD5 digest is used for every request!
- if we can guess a plaintext parameter string, and observe its encrypted version, then we can compute this MD5 digest, and we are done
- we could obtain plaintext parameter strings in multiple ways
  - some pages contained some parameter names and values in plaintext accidentally (e.g., comments or debug messages)
  - the web site contained open source web components, and we could examine the original program to get information on what parameter strings it used