This work is licensed under a Creative Commons license

Practical Cryptography

A gentle introduction to its everyday use

Simone Aonzo, PhD

EURECOM
Sophia Antipolis

Last update: 2025-01-07

Index

Outline

Code

In communications and information processing

- **Code** is an algorithm to convert information
  - E.g., word, sound, image, or gesture
- Into another form
  - E.g., shortened (compression) or secret (cryptography)
- For communication through a communication channel or storage in a storage medium

The process of

- **Encoding** converts information from a source into symbols for communication/storage
- **Decoding** converts code symbols back

Character encoding

Assigning numbers to characters, especially the graphical characters of human languages. Such numbers are known as *code points* and collectively comprise a *character map*.

Encoding – ASCII

ASCII is a 7-bit character encoding standard for representing text using the integers $[0, 127]$

```python
source_str = 'Hello!'

ascii_encoded_str = [ord(c) for c in source_str]
assert ascii_encoded_str == [72, 101, 108, 108, 111, 33]

decoded_str = ''.join([chr(c) for c in ascii_encoded_str])
assert source_str == decoded_str
```

`ord()` and `chr()` functions encodes/decodes w.r.t. Unicode

- Unicode standard defines Unicode Transformation Formats (UTF)
- In Python, Strings are by default in UTF-8 format
- UTF-8 is backward-compatible with ASCII

Encoding – Hexadecimal String

Strings in memory are just numbers

```
source_str = 'Hello!'

hex_encoded_str = bytes(source_str, 'ASCII').hex()
assert hex_encoded_str == '48656c6c6f21'
assert int(hex_encoded_str, 16) == 79600447942433

decoded_str = bytes.fromhex(hex_encoded_str).decode('ASCII')
assert source_str == decoded_str
```

Encoding – Base64

Base64 is a group of binary-to-text encoding schemes

- Designed to carry binary data across channels that only support text content
  - E.g., email attachments and HTML documents
- Represent binary data as a valid ASCII string using 64 characters
  - $`A` = 0$, $`B` = 1$, . . . , $`a` = 26$, . . . , $`+` = 62$, $`/` = 63$
  - Different implementations have different constraints on the alphabet
- 1 character encodes 6 bits $\rightarrow$ 4 characters encode 24 bits ($=$ 3 bytes)

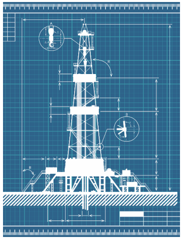
```python
import base64
source_bytes = b'\x48\x65\x6c\x6c\x6f\x21'
assert source_bytes == b'Hello!'

b64_encoded_str = base64.b64encode(source_bytes)
assert b64_encoded_str == b'SGVsbG8h'

b64_decoded_str = base64.b64decode(b64_encoded_str)
assert b64_decoded_str == source_bytes
```

Outline

Cryptography and Steganography

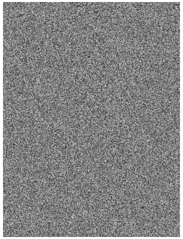
Cryptography, from ancient Greek

- Kryptós → hidden, secret
- Graphein → writing



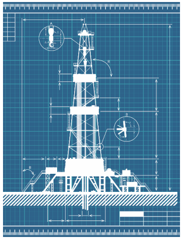
$+$

$=$



Steganography, from ancient Greek

- Steganós → covered, concealed
- Graphein → writing



$+$

$=$

Today's Cryptography

- Symmetric Encryption a.k.a. Symmetric Key
  - Block Ciphers (e.g., Advanced Encryption Standard)
  - Stream Ciphers (e.g., ChaCha)
- Asymmetric Cryptography a.k.a. Public-Key Cryptography
  - Key Exchange (e.g., Diffie-Hellman)
  - Digital Signature (e.g., Digital Signature Algorithm)
  - Key Exchange and Digital Signature (e.g., Rivest-Shamir-Adleman)
- Hashing (e.g., SHA-256)
- Message Authentication Code (e.g., HMAC)

DO NOT ROLL YOUR OWN CRYPTOGRAPHY

Why? *"is a bit like asking why you should not design your own aircraft engine"* [Runa Sandvik]

- Check if it is supported by the current platform/language
  - E.g., in Android, is very well done: `https://developer.android.com/guide/topics/security/cryptography`
- Use a certified/trusted open-source library otherwise
- Python?
  - Official modules
    - `https://docs.python.org/3/library/secrets.html`
    - `https://docs.python.org/3/library/hashlib.html`
    - ... many features are missing :(
  - `https://github.com/pyca/cryptography`
    - E.g., used by: `https://github.com/spesmilo/electrum`
  - `https://github.com/pyca/pynacl`
    - Python binding to `https://github.com/jedisct1/libsodium`

Outline

Encryption

Encryption is the principal application of cryptography

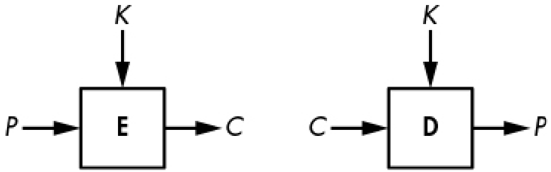
- It makes data “incomprehensible” in order to ensure its confidentiality
    - Incomprehensible $\rightarrow$ random-looking
- Encryption uses
    - An algorithm called a **cipher**
    - A secret value called the **key**
- Types
    - **Symmetric**: decryption key $=$ encryption key
    - **Asymmetric**: decryption key $\neq$ encryption key

Encoding is not encryption!

Encryption is an encoding operation, but the term encoding is generally used in cryptography to mean that secrecy is **not** involved

Symmetric Key Components

- **P**laintext (AKA cleartext) refers to the unencrypted message
- **C**iphertext to the encrypted message
- Symmetric $\implies$ sender and receiver both use the same secret **K**ey
- Cipher algorithm, two functions:
  - **E**ncryption(Key, Plaintext) $\rightarrow$ Ciphertext
  - **D**ecryption(Key, Ciphertext) $\rightarrow$ Plaintext



Permutation/Transposition – Scytale

- Reorder units of plaintext (typically characters or groups of characters)
- Produce a ciphertext which is a permutation of the plaintext
- The scytale was a tool used to perform a transposition cipher
  - Used by the Spartans (650 BC), Ancient Greek: skutálē "baton, cylinder"

  - A parchment is wrapped on a stick to encrypt/decipher the message
  - The key is the stick (diameter and length)
  - For example, to encrypt the message "`I am hurt very bad`"

```
      | I | a | m | h | u |__|
    __| r | t | v | e | r |
   |  | y | b | a | d |   |
```

  - Reading the parchment vertically gets the ciphertext: "`Iryatbmvahedur`"

Monoalphabetic substitution – Caesar Cipher (e.g., K=3)

Plain	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Cipher	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W

```
Plaintext:  THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG

Ciphertext: QEB NRFZH YOLTK CLU GRJMP LSBO QEB IXWV ALD
```

ROT13

ROTate by 13 places is a Caesar Cipher with $K = 13$ often used in online forums as a means of hiding spoilers, punchlines, puzzle solutions, and offensive materials from the casual glance.

Caesar Cipher in Python (e.g., K=8)

```python
import string

alphabet = list(string.ascii_uppercase)

def encrypt(key: int, plaintext: str) -> str:
    return ''.join(
        [alphabet[(alphabet.index(p) + key) % len(alphabet)] for p in plaintext])

def decrypt(key: int, ciphertext: str) -> str:
    return ''.join(
        [alphabet[(alphabet.index(c) - key) % len(alphabet)] for c in ciphertext])

secret = 8
source_str = 'HELLOWORLD'
assert encrypt(secret, source_str) == 'PMTTWEWZTL'
assert decrypt(secret, encrypt(secret, source_str)) == source_str
```

Attacking the Caesar Cipher

- Frequency analysis (i.e., counting letters)
  - In any given stretch of written language, certain letters and combinations of letters occur with varying frequencies
  - There is a characteristic distribution that is roughly the same for all samples of that language
  - For example, in the English language
    - Most frequent letters: E, T, A, and O. Bigrams: TH, ER, ON, and AN.
    - Most rare: Z, Q, X, and J.
- Brute-force (i.e., enumerating all candidates and checking each one)
  - Caesar Cipher key ranges [1, 25]
  - There are so few possibilities that even a human being can break it
  - 1883, *La cryptographie militaire* by Auguste Kerckhoffs
    - Encrypted instructions transmitted in wartime need not be held secret longer than 3/4 hours

Confusion and Diffusion

[1945] classified report *A Mathematical Theory of Cryptography* by Claude Shannon

Two properties in cipher design to make ciphertext resistant to cryptanalysis

1. **Confusion** (Key $\leftrightarrow$ Ciphertext)
   - The relationship between ciphertext$\sim$key must be as complex and involved as possible
   - Each bit of the ciphertext should depend on several parts of the key
   - Changing a single bit of the key $\implies$ "about half" ciphertext's bits change
   - How? Substitution algorithms
     - ★ Bytes of plaintext are substituted with others in a defined manner (e.g., Caesar cipher)
2. **Diffusion** (Plaintext $\leftrightarrow$ Ciphertext)
   - Hides the statistical properties of the plaintext spreading them across the entire ciphertext
   - Changing a single bit of the plaintext $\implies$ "about half" ciphertext's bits change
   - How? Permutation/Transposition algorithms
     - ★ Scramble the positions of bytes without changing the bytes themselves (e.g., rail fence cipher)

XOR – eXclusive OR

Exclusive or is a logical operation that is true iff its arguments differ

x	**y**	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

- $x \oplus y = x + y \mod 2$
- Commutativity: $x \oplus y = y \oplus x$
- Associativity: $x \oplus (y \oplus z) = (x \oplus y) \oplus z$
- Self-Inverse: $x \oplus x = 0$
- Identity: $x \oplus 0 = x$
- Negation: $x \oplus 1 = \neg x$

XOR – The cryptography's workhorse

Let P be a random variable over $\{0,1\}^n$
Let K be an independent uniform variable on $\{0,1\}^n$
Then $C := K \oplus P$ is uniform variable on $\{0,1\}^n$

Perfect Encryption: The One-Time Pad

- K is random
- $len(P) = len(K)$ 😑
- $OTP_{enc}(P, K) = C = P \oplus K$
- $OTP_{dec}(C, K) = P = C \oplus K$

OTP guarantees *perfect secrecy*

- Even with unlimited computing power, it is impossible to learn anything about the plaintext except for its length
- However, it does not protect the integrity or authenticity of the message
- An attacker can tamper the ciphertext, and there is no way of detecting the manipulation

One-Time $\rightarrow$ It can only be used one-time

Each key K should be used only once.
If the same K is used to encrypt P1 and P2 to C1 and C2, then an eavesdropper can:
$C_1 \oplus C_2 = (P_1 \oplus K) \oplus (P_2 \oplus K) = P_1 \oplus P_2 \oplus K \oplus K = P_1 \oplus P_2$

One-Time Pad

```python
import secrets  # https://docs.python.org/3/library/secrets.html

def otp(a: bytes, b: bytes) -> bytes:
    assert len(a) == len(b)
    return bytes([x^y for x, y in zip(a, b)])

source_str = 'Hello world!'
secret = secrets.token_bytes(len(source_str))

ciphertext = otp(secret, source_str.encode('ASCII'))
assert otp(secret, ciphertext).decode('ASCII') == source_str
```

Outline

Random Process

A sequence of random variables whose outcomes follow a *probability distribution*

- If there are N possible events, there are N probabilities $p_1, p_2, \ldots, p_N$
- $p_1 + p_2 + \cdots + p_N = 1$
- A *uniform distribution* occurs when all probabilities in the distribution are equal
    - I.e., each event has probability $\frac{1}{N}$

If a 128-bit key is picked uniformly at random (namely, according to a uniform distribution) then each of the 2^{128} possible keys should have a probability of $\frac{1}{2^{128}}$

Entropy: A Measure of Uncertainty

The amount of “information”, “surprise”, “uncertainty” in the result of a random process

- Given a discrete random variable X
- With possible outcomes $x_1, \dots, x_n$
- Which occur with probability $\mathrm{P}(x_1), \dots, \mathrm{P}(x_n)$
- The Shannon entropy of X is formally defined as:

$$\mathrm{H}(x) = -\sum_{i=1}^{n} \mathrm{P}(x_i) \log_2(\mathrm{P}(x_i))$$

If we consider 1 Byte $= 8$ bits, thus $2^8 = 256$ outcomes

$$n = 256 \implies 0 \leq \mathrm{H}(x) \leq 8$$

Entropy: A practical example

```python
import math
import secrets
from collections import Counter

def entropy_H(data: bytes) -> float:
    entropy = 0.0
    for x in Counter(data).values():
        p_x = float(x) / len(data)
        entropy -= p_x * math.log(p_x, 2)
    assert 0.0 <= entropy <= 8.0
    return entropy

def otp(a: bytes, b: bytes) -> bytes:
    return bytes([x^y for x, y in zip(a, b)])

plaintext = 'Coordinated attack 2023-08-01T13:42-1200 43.6144N,7.0711E'
for i in range(100000):
    ciphertext = otp(secrets.token_bytes(len(plaintext)), plaintext.encode('ASCII'))
    assert entropy_H(plaintext) < entropy_H(ciphertext)
```

RNGs and PRNGs

1. Random Number Generators (RNGs)
    - Randomness cannot be generated by computer-based algorithms alone
    - We need a *source of entropy*
    - The environment is analog, chaotic, uncertain, and hence unpredictable
    - E.g., an RNG can harvest the entropy in a running operating system
        - ★ attached sensors, I/O devices, network or disk activity, and user activities like mouse movement
    - Problems: can be fragile (manipulated by an attacker) and slow to yield random bits
2. Pseudo-Random Number Generators (PRNGs)
    - A cryptographic algorithm to produce high-quality random bits from the source of entropy
    - Reliably produce many artificial random bits from a few true random bits

RNGs vs. PRNGs

RNGs slowly produce true random bits from analog sources, in a nondeterministic way, and with no guarantee of high entropy.
PRNGs produce random-looking bits quickly from digital sources, in a deterministic way, and with maximum entropy.

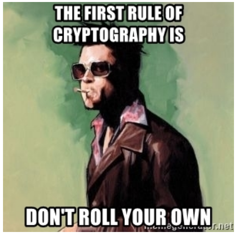
Cryptographic vs. Non-Cryptographic PRNGs

CAUTION: non-crypto PRNGs are only concerned with the quality of the bits' probability distribution and not with their predictability!

Refer to the documentation of the programming language, e.g.

- Python: `random` vs. `secrets`
- Java: `Random` vs. `SecureRandom`
- ...

What if a programming language does not natively support a Crypto-PRNG?



Outline

Block and Stream Ciphers

Modern symmetric ciphers fall into two main categories

1. **Block** ciphers
   - Break up a plaintext into fixed-length blocks
   - Send each *block* through an encryption function together with a secret key
   - PROs: offers many features (see modes of operation)
   - CONs: complicated to manage and interface with (e.g., padding)
2. **Stream** ciphers
   - Encrypt one byte of plaintext at a time
   - By XORing a pseudo-random key *stream* with the data
   - PROs: execute at a higher speed and have lower complexity
   - CONs: easily susceptible to security breaches if precautions are not followed

This distinction is not always clear-cut

In some modes of operation, a block cipher primitive is used in such a way that it acts effectively as a stream cipher (e.g., AES in CounTeR mode).

Block Ciphers – History

During the Cold War, the US and Soviets developed their own ciphers

- KGB developed GOST 28147-89
    - Kept secret until 1990 and still used today
- US created the Data Encryption Standard (DES)
    - Federal standard from 1979 to 2005
- US-National Institute of Standards and Technology (NIST) selected the successor to DES
    - In 2001, Advanced Encryption Standard (AES) won by Rijndael
    - Developed by two Belgian cryptographers (Vincent Rijmen and Joan Daemen)
    - So widespread that modern processors even contain special instructions for its operations
- AES, DES, and GOST 28147-89 have something in common: they are all block ciphers

Block Ciphers – Characteristics

Security depends on two values, which in turn characterize a block cipher

1. Key size
    - Obviously!
2. Block size
    - Most block ciphers have either 64-bit (e.g., DES) or 128-bit (e.g., AES) blocks

Blocks should not be too large in order to minimize

- The length of ciphertext; for example:
    - To encrypt a 16-bit message when blocks are $2^7 = 128$ bits
    - First, convert the message into a 128-bit block (*padding*)
    - Then, the block cipher processes it and returns a 128-bit ciphertext
- The memory footprint
    - The best solution is to make the block size fit the CPU registers

Modes of Operation

Permutation $+$ a mode of operation $\rightarrow$ handle messages of any length

- Electronic Codebook (ECB) Mode
  - ECB takes plaintext blocks $P_1, P_2, \ldots, P_N$
  - Processes each independently by computing $C_1 = E(K, P_1), C_2 = E(K, P_2), \ldots$
  - It is a simple operation but also an insecure one – do not use it!
- Cipher Block Chaining (CBC) Mode
  - Instead of encrypting the i_{th} block, P_i, as $C_i = E(K, P_i)$
  - CBC sets $C_i = E(K, P_i \oplus C_{i-1})$
  - When encrypting the first block, P_1, there is no previous ciphertext block to use
    - $\implies$ CBC takes a random Initial Value (IV)
- Counter (CTR) Mode
  - CTR turns a block cipher into a *stream cipher*
- Galois/Counter Mode (GCM) $=$ CTR $+$ *Authentication Tag* (Authenticity and Integrity)

Folklore techniques that nobody uses...

- Cipher FeedBack (CFB)
- Output FeedBack (OFB)

Example: Advanced Encryption Standard (AES)

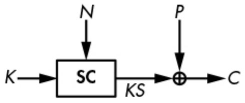
```python
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives import padding

import secrets; key = secrets.token_bytes(32); iv = secrets.token_bytes(16)

plaintext = 'This is a secret message'.encode('ASCII')
padding_PKCS7 = padding.PKCS7(128)
padder = padding_PKCS7.padder()
padded_data = padder.update(plaintext) + padder.finalize()

cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
encryptor = cipher.encryptor()
ciphertext = encryptor.update(padded_data) + encryptor.finalize()

decryptor = cipher.decryptor()
dec = decryptor.update(ciphertext) + decryptor.finalize()
unpadder = padding_PKCS7.unpadder()
data = unpadder.update(dec) + unpadder.finalize()
assert data == plaintext
```

Outline

Stream Ciphers

Stream ciphers do not mix plaintext and key bits like Block ciphers

- They produce a pseudorandom stream of bits called the **KeyStream** (KS)
- Are more akin to Deterministic Random Bit Generators (DRBGs)
- Encrypt the plaintext by XORing it with the pseudorandom bits – like the one-time pad
  - Each plaintext bit is encrypted one at a time with the corresponding bit of the KS
  - Encryption of each bit depends on the current state of the cipher $\Longrightarrow$ AKA *state ciphers*
- Take two values: a secret Key and a public **Nonce** (N)
  - NONCE = Number used only ONCE; sometimes also called the Initial Value (IV)
  - The nonce should be unique for each Key



Example: ChaCha20

```python
import secrets
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms

nonce = secrets.token_bytes(16)
key = b'Please,TheKeyMustBe32bytesLong!!'; assert len(key) == 32
plaintext = 'This is a secret message'.encode('ASCII')

cipher = Cipher(algorithms.ChaCha20(key, nonce), mode=None)
encryptor = cipher.encryptor()
ciphertext = encryptor.update(plaintext) + encryptor.finalize()

decryptor = cipher.decryptor()
decrypted_data = decryptor.update(ciphertext) + decryptor.finalize()
assert decrypted_data == plaintext
```

Outline

Hash functions

Take a long input and produce a short fixed-size output, called *hash value* or *digest*

$$hash : \{0,1\}^m \rightarrow \{0,1\}^n$$

Pigeonhole Principle

If you have m pigeons and n holes to put into those holes, and if $m > n$ at least one hole must contain more than one pigeon

1. Non-cryptographic
   - Data structures (e.g., hash tables)
   - Detect errors (e.g., Cyclic Redundancy Checks CRCs)
   - Detect similarities!
     - Context Triggered Piecewise Hashes (CTPH) aka fuzzy hashes (e.g., ssdeep)
   - ... and many more
2. Cryptographic
   - Verify data integrity
   - Designed to be one-way: functions that are practically impossible to invert
   - All the cryptographic strength stems from the unpredictability of their outputs

Cryptographic Hash Function Properties

- Avalanche effect
  - If a message is changed slightly, its hash changes significantly
- Pre-image resistance
  - Given a hash value h
  - it should be difficult to find any message m such that $h = hash(m)$
- Second pre-image resistance (aka *weak* collision resistance)
  - Given a message m_1
  - it should be difficult to find a m_2 s.t. $m_1 \neq m_2$ and $hash(m_1) = hash(m_2)$
- Collision resistance (aka *strong* collision resistance)
  - It should be difficult to find two different messages m_1 and m_2 s.t. $hash(m_1) = hash(m_2)$
  - Such a pair is called a cryptographic **hash collision**
  - $\implies$ second pre-image resistance
  - $\not\Longrightarrow$ pre-image resistance

Famous Cryptographic Hash Functions

- Message Digest 5 (MD5) $\rightarrow$ 128 bits
  - Most popular hash function from 1992, definitively broken around 2005
  - It is possible to compute collisions for the full MD5 hash $\implies$ do not use it!
- Secure Hash Algorithm (SHA) families
  - Defined by NIST and considered worldwide standards
  - SHA-1 (based on Merkle tree) $\rightarrow$ 128 bits
    - Definitively broken around 2017 $\implies$ do not use it!
  - SHA-2 (based on Merkle tree)
    - SHA-256 $\rightarrow$ 256 bits, and SHA-512 $\rightarrow$ 512 bits
    - Researchers were concerned due to its similarity to SHA-1
    - We have yet to see a successful attack
  - SHA-3 – decided through a competition
    - Winner: Keccak (based on sponge construction)
    - SHA3-224, SHA3-256, SHA3-384, and SHA3-512
    - Not faster than SHA-2
- BLAKE2, a SHA-3 finalist
  - Faster than all previous hashes (parallel variants use multiple CPU cores), including MD5
  - But it is based on Merkle tree, at the time of the SHA-3 competition this was a disadvantage

Hash Functions Example

```python
import os
from os.path import dirname, realpath
from hashlib import sha256, blake2s

h1 = sha256('Hello world!'.encode('utf-8')).hexdigest()
h2 = sha256('Hello world?'.encode('utf-8')).hexdigest()

assert h1 == 'c0535e4be2b79ffd93291305436bf889314e4a3faec05ecffcbb7df31ad9e51a'
assert h2 == '43f497ee7ac09843d631362ef9aca26a0cab437acaea8a98e44afa7ad65a2d41'

found = False
for fname in os.listdir(dirname(realpath(__file__))):
    fname_h = blake2s(fname.encode('utf-8')).hexdigest()
    # This is the hash of the file's name, NOT the file's content
    if fname_h == 'd85380bdf6e710faa30b38dcb029385b0ad2b040b0b6099ad777ecaeace1f124':
        found = True
        break
assert found
```

Applications of Hash Functions

- Data Integrity
  - Main application
- Password Verification (e.g., Argon2)
  - Cryptographically strong, *slow* to compute, and *salt* is included in the algorithm
- Data Structures (e.g., hashmap)
  - Often non-cryptographic – performances are crucial
- Algorithms (e.g., Rabin-Karp)
  - Extremely variable scenarios
- Proof-of-Work in Blockchain (e.g., Bitcoin with SHA-256)
  - If the cryptography falls, the whole blockchain falls!
  - Idea: given n, find an x s.t. $sha256(x) > n$
- Message Authentication Codes (MACs)

Outline

Message Authentication Codes (MACs)

- Protects a message's Integrity and Authenticity
- By creating the authentication *tag* $T = MAC(K, M)$
- You can validate that a message has not been modified if you know a MAC's **K**ey
- If a MAC is secure has the property of *unforgeability*
  - An attacker cannot create a tag of some message if they do not know the key
  - Such a valid made-up message/tag pair is called a *forgery*
- Attacker models
  - Known-message attacks: passively (eavesdropping) collects messages and their tags
  - Chosen-message attacks: get tags for messages of their choice

MACs in Secure Communication

- Cipher + MAC $\rightarrow$ protect the message's confidentiality, integrity, and authenticity
  - E.g., Secure Shell (SSH) generates a MAC for each network packet transmitted
- However, sometimes an authentication tag can add unacceptable overhead
  - E.g., packets encoding voice calls in 3G and 4G mobile standards
  - If an attacker damages an encrypted voice packet, it will decrypt to noise

MAC Example with HMAC (keyed-Hash MAC)

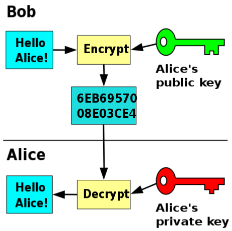
```python
import hmac
import hashlib
import binascii

secret = binascii.unhexlify('DEADBEEF')  # Shared secret key between Alice and Bob

def hmac_sha256(key: bytes, msg: bytes) -> str:
    hash_function = hashlib.sha256  # Concerted hash function (n.b., NOT a secret!)
    return hmac.new(key, msg, hash_function).hexdigest().upper()

# Bob received a message and a tag from Alice
message_from_alice = 'Hi, I am a message. Pleased to meet you!'.encode('utf-8')
tag_from_alice = 'B7CBE1343FBD1BCDDB322E6E9A02209E4A4C914B7DB01D9691AECDCAB8C76CE2'

tag_of_bob = hmac_sha256(secret, message_from_alice)  # Bob computes his tag

# Finally, Bob verifies that his tag and Alice's tag are the same
assert hmac.compare_digest(tag_from_alice, tag_of_bob)
# compare_digest(a, b) instead of a == b ? Reduces the vulnerability to timing attacks
```

Outline

Public-Key Encryption

Public-Key (also called Asymmetric) Encryption uses two keys

1. Public Key, which can be used by anyone
    - *Encryption*(PublicKey, Plaintext) → Ciphertext
2. Private Key, which is required in order to decrypt messages encrypted using the public key
    - *Decryption*(PrivateKey, Ciphertext) → Plaintext

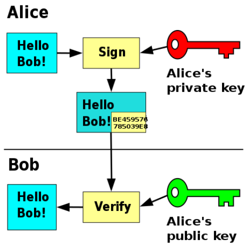


Asymmetric encryption is slower than Symmetric encryption

Today's cryptosystems (e.g., TLS, Secure Shell) use a **hybrid** approach, by using asymmetric encryption to securely exchange a secret key which is then used for symmetric encryption

Digital Signatures

- The owner of the private key is the only one able to sign a message
  - *Sign*(PrivateKey, Message) $\rightarrow$ Signature
- The public key enables anyone to verify the signature's validity
  - *Verify*(PublicKey, Signature) $\rightarrow \{True, False\}$



Performance always matters!

Signing long messages is computationally demanding $\Longrightarrow$ *Sign*(PrivateKey, **Hash**(Message))

Rivest-Shamir-Adleman (RSA)

- RSA revolutionized cryptography when it emerged in 1977 – still widely used today
- Public-Key Encryption AND Digital Signatures
  - Encryption and signature keys should be kept disjoint
- RSA is above all an arithmetic trick
  - Creates a mathematical object called a **trapdoor function**
  - Transforms a number P to a number C in the same range (**modular exponentiation**) s.t.
  - Computing C from P is “easy” using the public key
  - However, computing P from C is “practically impossible in a reasonable time” unless you know the private key
  - **Factoring problem** the “difficulty” of factoring the product of two large prime numbers

$C = 6,895,601$ is the product of two prime numbers. What are those numbers?

- Brute-force: try dividing 6895601 by several prime numbers until finding the answer
- Private Key is 1931 $\implies P = 6895601 \div 1931 = 3571$

RSA Encryption Example

```python
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding, rsa

private_key = rsa.generate_private_key(
    # always use 65537 because it indicates one property of the key generation
    public_exponent=0x10001,
    key_size=3072)  # NSA, NIST, ANSSI, FIPS, etc., recommend at least 3072
public_key = private_key.public_key()

message = 'This is a secret message! Protect me!'.encode('utf-8')
pad = padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None)

ciphertext = public_key.encrypt(message, pad)
plaintext = private_key.decrypt(ciphertext, pad)
assert plaintext == message
```

RSA Signature Example

```python
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding, rsa
from cryptography.exceptions import InvalidSignature

private_key = rsa.generate_private_key(public_exponent=0x10001, key_size=4096)
public_key = private_key.public_key()
chosen_hash = hashes.SHA256()

message = 'Hello, sign me :)'.encode('utf-8')
pad = padding.PSS(
    mgf=padding.MGF1(chosen_hash),
    salt_length=padding.PSS.MAX_LENGTH)

signature = private_key.sign(message, pad, chosen_hash)  # Alice signs the message
try:
    public_key.verify(signature, message, pad, chosen_hash)  # Bob verifies it
except InvalidSignature:
    assert False
```

Diffie-Hellman Key Exchange

- "New Directions in Cryptography", 1976 – one year prior to RSA
- They introduced the notion of public-key encryption and signatures
- However, they did not actually have any of those schemes
- Diffie-Hellman (DH) protocol is a **key exchange** algorithm $\neq$ public key encryption
  - Based on assumption that the **discrete logarithm problem** has no efficient solution
  - Allows two parties to establish a shared secret by exchanging information visible to an eavesdropper
  - "More or less" equivalent to using public key encryption on a random message
  - In fact, Diffie-Hellman can be turned into public key encryption (ElGamal encryption)

Diffie-Hellman vs. RSA

Both use modular exponentiation to provide the main functionality, but the underlying problem (in fact, RSA is threatened by integer factorization, while DH by discrete logarithms), the key pair generation, and the security properties of the input/output are different

Elliptic-Curve Cryptography

- In 1985, Elliptic-Curve Cryptography (ECC) revolutionized public-key cryptography
  - ECC was not adopted by standardization bodies until the early 2000s
  - Was not seen in major toolkits until much later (OpenSSL in 2005, and OpenSSH in 2011)
- ECC is a family of algorithms that can perform *encryption*, generate *signatures*, *key agreement*, and offer *advanced* cryptographic functionalities
  - E.g., Elliptic-curve Diffie-Hellman (ECDH) is a variant of Diffie-Hellman using ECC
- ECC is more powerful and efficient than alternatives like RSA and DH
  - ECC with a 256-bit key is stronger than RSA with a 4096-bit key
  - Like RSA, ECC multiplies large numbers
  - Unlike RSA, it does so to combine points on a mathematical curve, called an **elliptic curve**
  - Based on the "Elliptic Curve Discrete Logarithm Problem" (ECDLP)
    - Ability to compute a point multiplication
    - Inability to compute the multiplicand given the original and product points
  - More complex: different types, efficient/inefficient, and secure/insecure elliptic curves
  - Most cryptographic applications use *NIST curves* or *Curve25519*

Outline

Crypto Wars 1/2

Crypto Wars is an unofficial name for the attempts of the US and allied governments to limit the public's and foreign nations' access to strong cryptography

- Diffie and Hellman received the prestigious Turing Award in 2015 for their 1976 invention
- In 1974, researchers at GCHQ (the British equivalent of the NSA) discovered the principles behind RSA and DH
- However, this would only be declassified decades later

The Clipper chip

- The Clipper chip was a chipset developed and promoted by NSA as an encryption device that secured "voice and data messages"
- It had a built-in backdoor intended to "allow Federal, State, and local law enforcement officials the ability to decode intercepted voice and data transmissions"
- At the heart of the concept was *key escrow*
- I.e., under certain circumstances an authorized third party may gain access to the keys
- In the chip factory, any device would be given a key provided to the government in escrow

Crypto Wars 2/2

Dual Elliptic Curve Deterministic Random Bit Generator (Dual_EC_DRBG) by NSA

- September 2013, report of both The Guardian and The New York Times
- In 2006, NIST allowed the NSA to insert a crypto PRNG in a NIST standard (SP 800-90)
- This PRNG, named Dual_EC_DRBG, had a backdoor
- The NSA can use it to covertly predict the future outputs of the generator
  - Thereby allowing the surreptitious decryption of data!

NIST Curves – by NSA

- The most common NIST curves are the prime curves
- The most common is P-256, and its equation is $y^2 = x^3 - 3^x + b$
- The *b* coefficient is a 256-bit number
- Only the NSA, creator of the curves, knows the origin of this coefficient in their equations
- Most experts do not believe the curve's origin hides any weakness...

Nothing-up-my-sleeve number

Any numbers which, by their construction, are above suspicion of hidden properties

- Hashes and ciphers often need randomized constants for mixing or initialization
- Pick numbers that demonstrate they were not selected for a nefarious purpose
- Examples (key schedule constants):
  - Blowfish cipher uses the binary representation of π, while RC5 binary digits from e
  - KASUMI cipher uses `0x123456789ABCDEFFEDCBA9876543210`

A "good" counterexample

The Data Encryption Standard (DES) has constants that were given out by NSA. They turned out to be far from random but instead made the algorithm resilient against differential cryptanalysis, a method not publicly known at the time.

Takeaways

- Do not roll your own crypto
- Encoding $\neq$ Encrypting
- Cipher $\rightarrow$ Confidentiality
  - Encryption(encKey, Plaintext) $\rightarrow$ Ciphertext
    - Ciphertext must be indistinguishable from random
  - Decryption(decKey, Ciphertext) $\rightarrow$ Plaintext
  - encKey $=$ decKey $\implies$ Symmetric Encryption
    - Block: fixed-length groups of bits
    - Stream: wannabe one-time pad
  - encKey $\neq$ decKey $\implies$ Asymmetric Cryptography a.k.a. Public-Key Cryptography
    - Slower encryption than Symmetric
    - Elliptic-Curve uses far smaller keys
    - Bonus features: Key Exchange and Digital Signature
- Hash $\rightarrow$ Integrity
- Cipher $+$ MAC $\rightarrow$ protect the message's Confidentiality, Authenticity, and Integrity