

Practical Cryptography for Developers



Hashes (SHA2, SHA3, BLAKE2), MAC and HMAC, Key Derivation (KDF, Scrypt, Argon2), Secure Random Numbers (CSPRNG), Key Agreement (DHKE, ECDH), Symmetric Ciphers (Block Modes, AES-CTR, AES-GCM, ChaCha20-Poly1307), Asymmetric Ciphers (RSA, Elliptic Curve Cryptography, ECC, ECDH, ECIES, secp256k1), Hybrid Encryption Schemes (ECIES), Digital Signatures (RSA, DSA, ECDSA, EdDSA, Ed25519, Ed448), Quantum-Safe Cryptography, Digital Certificates, TLS, OTP, Code Examples (Python, JavaScript, C#, Java), Crypto Libraries

Svetlin Nakov, PhD

SoftUni – <https://softuni.org>

Table of Contents

Welcome	1.1
Preface	1.2
Cryptography - Overview	1.3
Hash Functions	1.4
Crypto Hashes and Collisions	1.4.1
Hash Functions: Applications	1.4.2
Secure Hash Algorithms	1.4.3
Hash Functions - Examples	1.4.4
Exercises: Calculate Hashes	1.4.5
Proof-of-Work Hash Functions	1.4.6
MAC and Key Derivation	1.5
HMAC and Key Derivation	1.5.1
HMAC Calculation - Examples	1.5.2
Exercises: Calculate HMAC	1.5.3
KDF: Deriving Key from Password	1.5.4
PBKDF2	1.5.5
Modern Key Derivation Functions	1.5.6
Scrypt	1.5.7
Bcrypt	1.5.8
Linux crypt()	1.5.9
Argon2	1.5.10
Password Encryption	1.5.11
Exercises: Password Encryption	1.5.12
Secure Random Generators	1.6
Pseudo-Random Numbers - Examples	1.6.1
Secure Random Generators (CSPRNG)	1.6.2
Exercises: Pseudo-Random Generator	1.6.3
Key Exchange and DHKE	1.7
Diffie–Hellman Key Exchange	1.7.1
DHKE - Examples	1.7.2
Exercises: DHKE Key Exchange	1.7.3
Encryption: Symmetric and Asymmetric	1.8
Symmetric Key Ciphers	1.9
Cipher Block Modes	1.9.1
Popular Symmetric Algorithms	1.9.2
The AES Cipher - Concepts	1.9.3
AES Encrypt / Decrypt - Examples	1.9.4
Ethereum Wallet Encryption	1.9.5

Exercises: AES Encrypt / Decrypt	1.9.6
ChaCha20-Poly1305	1.9.7
Exercises: ChaCha20-Poly1305	1.9.8
Asymmetric Key Ciphers	1.10
The RSA Cryptosystem - Concepts	1.10.1
RSA Encrypt / Decrypt - Examples	1.10.2
Exercises: RSA Encrypt / Decrypt	1.10.3
Elliptic Curve Cryptography (ECC)	1.10.4
ECDH Key Exchange	1.10.5
ECDH Key Exchange - Examples	1.10.6
Exercises: ECDH Key Exchange	1.10.7
ECC Encryption / Decryption	1.10.8
ECIES Hybrid Encryption Scheme	1.10.9
ECIES Encryption - Example	1.10.10
Exercises: ECIES Encrypt / Decrypt	1.10.11
Digital Signatures	1.11
RSA Signatures	1.11.1
RSA: Sign / Verify - Examples	1.11.2
Exercises: RSA Sign and Verify	1.11.3
ECDSA: Elliptic Curve Signatures	1.11.4
ECDSA: Sign / Verify - Examples	1.11.5
Exercises: ECDSA Sign and Verify	1.11.6
EdDSA and Ed25519	1.11.7
EdDSA: Sign / Verify - Examples	1.11.8
Exercises: EdDSA Sign and Verify	1.11.9
Quantum-Safe Cryptography	1.12
Quantum-Safe Signatures - Example	1.12.1
More Cryptographic Concepts	1.13
Digital Certificates - Example	1.13.1
TLS - Example	1.13.2
One-Time Passwords (OTP) - Example	1.13.3
Crypto Libraries for Developers	1.14
JavaScript Crypto Libraries	1.14.1
Python Crypto Libraries	1.14.2
C# Crypto Libraries	1.14.3
Java Crypto Libraries	1.14.4
Conclusion	1.15

Practical Cryptography for Developers - Free Book

A modern **practical book** about **cryptography for developers** with code examples, covering core concepts like: **hashes** (like SHA-3 and BLAKE2), **MAC codes** (like HMAC and GMAC), **key derivation functions** (like Scrypt, Argon2), **key agreement protocols** (like DHKE, ECDH), **symmetric ciphers** (like AES and ChaCha20, cipher block modes, authenticated encryption, AEAD, AES-GCM, ChaCha20-Poly1305), **asymmetric ciphers** and **public-key cryptosystems** (RSA, ECC, ECIES), **elliptic curve cryptography** (ECC, secp256k1, curve25519), **digital signatures** (ECDSA and EdDSA), **secure random numbers** (PRNG, CSRNG) and **quantum-safe cryptography**, along with crypto **libraries** and developer tools, with a lots of **code examples** in Python and other languages.

Author: **Svetlin Nakov**, PhD - <http://www.nakov.com>

This book is free and open-source, published under the [MIT license](#).

Official **GitHub** repo: <https://github.com/nakov/practical-cryptography-for-developers-book>.

Sofia, November 2018

Summary

- [Welcome](#)
- [Preface](#)
- [Cryptography - Overview](#)
- [Hash Functions](#)
 - [Crypto Hashes and Collisions](#)
 - [Hash Functions: Applications](#)
 - [Secure Hash Algorithms](#)
 - [Hash Functions - Examples](#)
 - [Exercises: Calculate Hashes](#)
 - [Proof-of-Work Hash Functions](#)
- [MAC and Key Derivation](#)
 - [HMAC and Key Derivation](#)
 - [HMAC Calculation - Examples](#)
 - [Exercises: Calculate HMAC](#)
 - [KDF: Deriving Key from Password](#)
 - [PBKDF2](#)
 - [Modern Key Derivation Functions](#)
 - [Scrypt](#)
 - [Bcrypt](#)
 - [Linux crypt\(\)](#)
 - [Argon2](#)
 - [Password Encryption](#)
 - [Exercises: Password Encryption](#)
- [Secure Random Generators](#)
 - [Pseudo-Random Numbers - Examples](#)
 - [Secure Random Generators \(CSPRNG\)](#)
 - [Exercises: Pseudo-Random Generator](#)
- [Key Exchange and DHKE](#)
 - [Diffie–Hellman Key Exchange](#)
 - [DHKE - Examples](#)
 - [Exercises: DHKE Key Exchange](#)
- [Encryption: Symmetric and Asymmetric](#)

- [Symmetric Key Ciphers](#)
 - [Cipher Block Modes](#)
 - [Popular Symmetric Algorithms](#)
 - [The AES Cipher - Concepts](#)
 - [AES Encrypt / Decrypt - Examples](#)
 - [Ethereum Wallet Encryption](#)
 - [Exercises: AES Encrypt / Decrypt](#)
 - [ChaCha20-Poly1305](#)
 - [Exercises: ChaCha20-Poly1305](#)
- [Asymmetric Key Ciphers](#)
 - [The RSA Cryptosystem - Concepts](#)
 - [RSA Encrypt / Decrypt - Examples](#)
 - [Exercises: RSA Encrypt / Decrypt](#)
 - [Elliptic Curve Cryptography \(ECC\)](#)
 - [ECDH Key Exchange](#)
 - [ECDH Key Exchange - Examples](#)
 - [Exercises: ECDH Key Exchange](#)
 - [ECC Encryption / Decryption](#)
 - [ECIES Hybrid Encryption Scheme](#)
 - [ECIES Encryption - Example](#)
 - [Exercises: ECIES Encrypt / Decrypt](#)
- [Digital Signatures](#)
 - [RSA Signatures](#)
 - [RSA: Sign / Verify - Examples](#)
 - [Exercises: RSA Sign and Verify](#)
 - [ECDSA: Elliptic Curve Signatures](#)
 - [ECDSA: Sign / Verify - Examples](#)
 - [Exercises: ECDSA Sign and Verify](#)
 - [EdDSA and Ed25519](#)
 - [EdDSA: Sign / Verify - Examples](#)
 - [Exercises: EdDSA Sign and Verify](#)
- [Quantum-Safe Cryptography](#)
 - [Quantum-Safe Signatures - Example](#)
- [More Cryptographic Concepts](#)
 - [Digital Certificates - Example](#)
 - [TLS - Example](#)
 - [One-Time Passwords \(OTP\) - Example](#)
- [Crypto Libraries for Developers](#)
 - [JavaScript Crypto Libraries](#)
 - [Python Crypto Libraries](#)
 - [C# Crypto Libraries](#)
 - [Java Crypto Libraries](#)
- [Conclusion](#)

Tags: [cryptography](#), [free](#), [book](#), [Nakov](#), [Svetlin Nakov](#), [hashes](#), [hash function](#), [SHA-256](#), [SHA3](#), [BLAKE2](#), [RIPEMD](#), [MAC](#), [message authentication code](#), [HMAC](#), [KDF](#), [key derivation](#), [key derivation function](#), [PBKDF2](#), [Scrypt](#), [Bcrypt](#), [Argon2](#), [password hashing](#), [random generator](#), [pseudo-random numbers](#), [CSPRNG](#), [secure random generator](#), [key exchange](#), [key agreement](#), [Diffie-Hellman](#), [DHKE](#), [ECDH](#), [symmetric ciphers](#), [asymmetric ciphers](#), [public key cryptosystems](#), [symmetric cryptography](#), [AES](#), [Rijndael](#), [cipher block mode](#), [AES-CTR](#), [AES-GCM](#), [ChaCha20-Poly1305](#), [authenticated encryption](#), [encryption scheme](#), [public key cryptography](#), [RSA](#), [ECC](#), [elliptic curves](#), [secp256k1](#), [curve25519](#), [EC points](#), [EC domain parameters](#), [ECDH key agreement](#), [asymmetric encryption scheme](#),

hybrid encryption, ECIES, digital signature, RSA signature, DSA, ECDSA, EdDSA, ElGammal signature, Schnorr signature, quantum-safe cryptography, digital certificates, TLS, OAuth, multi-factor authentication, crypto libraries, Python cryptography, JavaScript cryptography, C# cryptography, Java cryptography, C++ cryptography, PHP cryptography.

Preface

...

Most books about cryptography are written either in too **academic style** with a lot of theory, like <http://cacr.uwaterloo.ca/hac>.

...

Others are too old: ...

...

Others are not bad, but not free:

<https://leanpub.com/crypto>

<https://www.amazon.com/Cryptography-Developers-Tom-St-Denis/dp/1597491047>

Crypto libraries come with limited and not consistently organized documentation, e.g. the Crypto++ Wiki https://www.cryptopp.com/wiki/Main_page.

...

Now I am happy to publish a **developer-friendly practical cryptography book**. It holds just **what developers need to know** in order to **use cryptography in their every day work**. It does not cover the internals of the algorithms and how to design symmetric ciphers or authentication algorithms. It covers the basic understanding of the **core cryptographic concepts** and how to use them: **libraries, tools, code examples**.

...

The book author **Svetlin Nakov** is involved with applied cryptography from 2005, when he published the book "Java for Digitally Signing Documents of the Web" (in Bulgarian), following his master thesis on a similar topic.

...

It is not required to be strong mathematician to understand the cryptographic concepts from developer perspective. This book will teach you the basic concepts in almost math-free style.

Overview of Modern Cryptography

Cryptography has evolved from its first attempts (thousands years ago), through the first successful cryptographic algorithms for developers (like the now retired MD5 and DES) to modern crypto algorithms (like SHA-3, Argon2 and ChaCha20).

Let's first introduce very shortly the basic **cryptography concepts**, that developers should know, like cryptographic **hash functions** (SHA-256, SHA3, RIPEMD and others), **HMAC** (hashed message authentication code), password to **key derivation** functions (like **Scrypt**), the Diffie-Hellman key-exchange protocol, **symmetric key** encryption schemes (like the **AES** cipher with CBC and CTR block modes) and **asymmetric key** encryption schemes with public and private keys (like the **RSA** cipher and elliptic curves-based cryptography / **ECC**, the secp256k1 curve and the Ed25519 cryptosystem), **digital signatures** and **ECDSA**, as well as the concept of **entropy** and secure **random number** generation and **quantum-safe cryptography**.

Encrypt / Decrypt Message - Live Demo

As a simple **example**, we shall demonstrate message **encryption** + **decryption** using the **AES** encryption algorithm. Play with this online tool: <https://aesencryption.net>.



The screenshot shows a web browser window with the URL <https://aesencryption.net>. The page has an orange header with the text "AES encryption" and a hamburger menu icon. The main content area is a light gray box with the title "AES encryption" in a large, bold, italicized font. Below the title is the subtitle "Encrypt and decrypt text with AES algorithm". There are three input fields: a text area containing "secret message", a text input field containing "password123", and a dropdown menu set to "128 Bit". At the bottom right of the form are two buttons: "Encrypt" (orange) and "Decrypt" (dark gray).

We shall learn later that behind this simple **AES encryption**, there are **many algorithms and settings** hidden inside, like password to key-derivation function and its parameters, block cipher mode, cipher initial vector, message authentication code and others.

What is Cryptography?

Cryptography is the science of providing **security** and **protection** of information. It is used everywhere in our digital world: when you open a Web site, send an email or connect to the WiFi network. What's why developers should have **at least basic understanding of cryptography** and how to use crypto algorithms and crypto libraries, to understand hashing, symmetric and asymmetric ciphers and encryption schemes, as well as digital signatures and the cryptosystems and algorithms behind them.

Encryption and Keys

Cryptography deals with **storing and transmitting data in a secure way**, such that only those, for whom it is intended, can read and process it. This may involve **encrypting and decrypting data** using symmetric or asymmetric encryption schemes, where one or more **keys** are used to transform data from plain to encrypted form and back.

Symmetric encryption (like AES, Twofish and ChaCha20) uses the same key to encrypt and decrypt messages, while **asymmetric encryption** uses a **public-key cryptosystem** (like RSA or ECC) and a key-pair: private key (encryption key) and corresponding public key (decryption key). Encryption algorithms are often combined in encryption schemes (like AES-256-CTR-HMAC-SHA-256, ChaCha20-Poly1305 or ECIES-AES-128-GCM).

Cryptography deals with **keys** (large secret numbers) and in many scenarios these **keys are derived** from numbers, passwords or passphrases using **key derivation algorithms** (like PBKDF2 and Scrypt).

Digital Signatures and Message Authentication

Cryptography provides means of **digital signing of messages** which guarantee message authenticity, integrity and non-repudiation. Most digital signature algorithms (like DSA, ECDSA and EdDSA) use **asymmetric key pair** (private and public key): the message is **signed** by the private key and the signature is **verified** by the corresponding public key. In the bank systems **digital signatures** are used to sign and approve payments. In blockchain signed transactions allow users to transfer a blockchain asset from one address to another.

Cryptography deals with **message authentication** algorithms (like HMAC) and message authentication codes (MAC codes) to prove message authenticity, integrity and authorship. Authentication is used side by side with encryption, to ensure secure communication.

Secure Random Numbers

Cryptography uses **random numbers** and deals with **entropy** (unpredictable randomness) and secure generation of random numbers (e.g. using CSPRNG). **Secure random numbers** are unpredictable by nature and developers should care about them, because broken random generator means compromised or hacked system or app.

Key Exchange

Cryptography defines **key-exchange algorithms** (like Diffie-Hellman key exchange and ECDH) and **key establishment schemes**, used to securely establish encryption **keys** between two parties that intend to transmit messages securely using **encryption**. Such algorithms are performed typically when a new secure connection between two parties is established, e.g. when you open a modern Web site or connect to the WiFi network.

Cryptographic Hashes and Password Hashing

Cryptography provides **cryptographic hash functions** (like SHA-3 and BLAKE2), which transform messages to **message digest** (hash of fixed length), which cannot be reversed back to the original message and almost uniquely identifies the input. In **blockchain** systems, for example, hashes are used to generate blockchain addresses, transaction ID and in many other algorithms and protocols. In **Git** cryptographic hashes are used for generating unique ID for files and commits.

Password hashing and password to **key derivation functions** (like Scrypt and Argon2) protect user passwords and password encrypted documents and data by securely deriving a hash (or key) from a text-based passwords, injecting random parameters (salt) and using a lot of iterations and computing resources to make password cracking slow.

Confusion and Diffusion in Cryptography

In cryptography the hashing, encryption algorithms and random generators follow the Shannon's principles of [confusion and diffusion](#). **Confusion** means that each bit in the output from a cipher should depend on several parts of the key and input data and thus direct mapping cannot be established. **Diffusion** means that changing one bit in the input should change approximately half of the bits in the output. These principles are incorporated in most hash functions, MAC algorithms, random number generators, symmetric and asymmetric ciphers.

Cryptographic Libraries

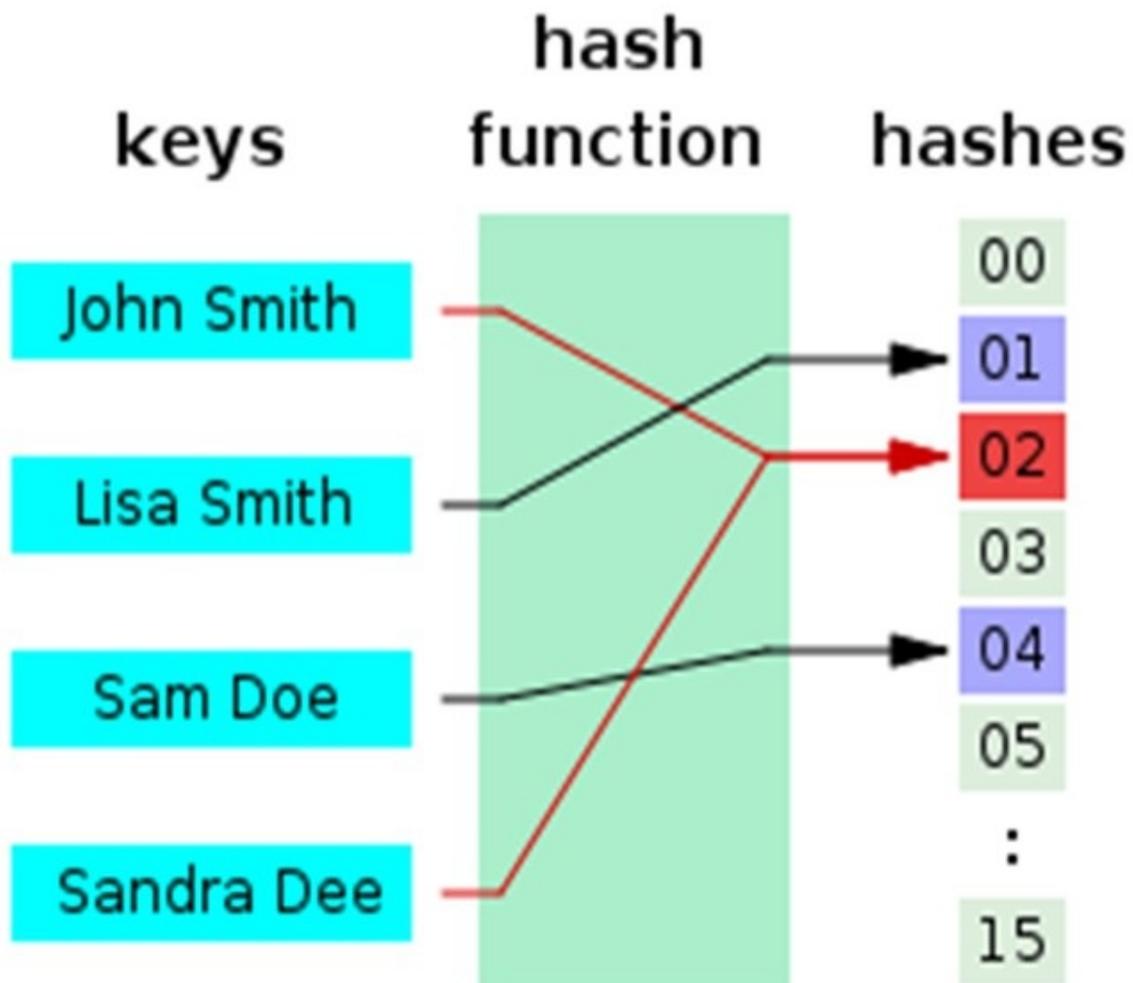
Developers should know the modern **cryptographic libraries** for their programming language and platform and how to use them. Developing with cryptography requires **understanding of the crypto-concepts**. Copy / pasting code from Internet or following an example from a blog may lead to insecure design and weak security. Cryptographic libraries are very useful, but you should **understand the concepts** first, then choose appropriate combination of algorithms and adjust carefully their parameters.

Hashing and Cryptographic Hash Functions

In computer programming **hash functions** map text (or other data) to integer numbers. Usually different inputs maps to different outputs, but sometimes a **collision** may happen (different input with the same output).

Hashing

The process of calculating the value of certain hash function is called "**hashing**".



In the above example the text `John Smith` is hashed to the hash value `02` and `Lisa Smith` is hashed to `01`. The input texts `John Smith` and `Sandra Dee` both are hashed to `02` and this is called "**collision**".

Hash functions are **irreversible by design**, which means that there is no fast algorithm to restore the input message from its hash value.

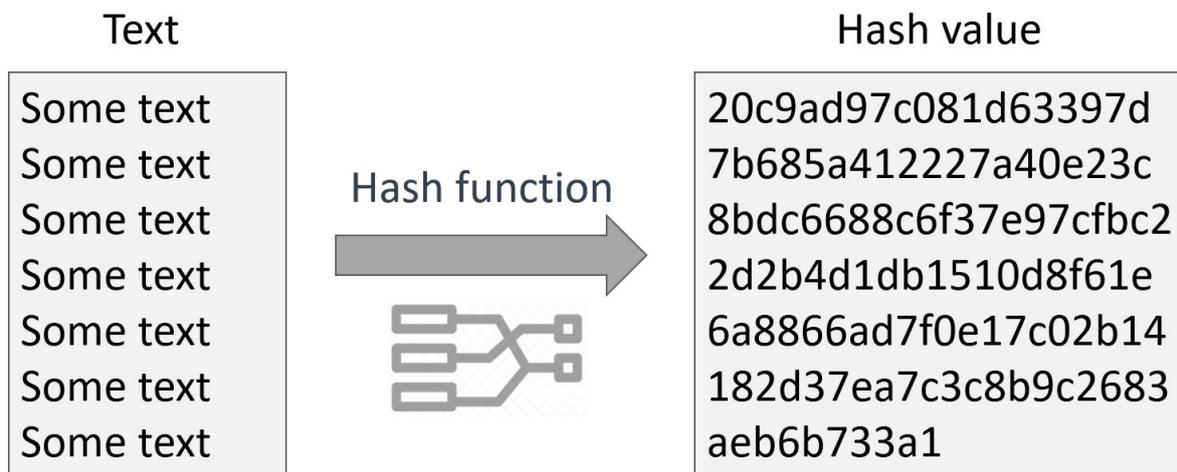
In programming **hash functions** are used in the implementation of the data structure "**hash-table**" (associative array) which maps values of certain input type to values of another type, e.g. map product name (text) to product price (decimal number).

A **naive hash function** is just to sum the bytes of the input data / text. It causes a lot of collisions, e.g. `hello` and `ehll0` will have the same hash code. **Better hash functions** may use the [Merkle–Damgård construction](#) scheme, which takes the first byte as **state**, then **transforms the state** (e.g. multiplies it by a prime number like 31), then **adds**

the next byte to the state, then again transforms the state and adds the next byte, etc. This significantly reduces the rate of collisions and produces better distribution.

Cryptographic Hash Functions

In cryptography, **hash functions** transform **input data** of arbitrary size (e.g. a text message) to a **result** of fixed size (e.g. 256 bits), which is called **hash value** (or hash code, message digest, or simply hash). Hash functions (hashing algorithms) used in computer cryptography are known as "**cryptographic hash functions**". Examples of such functions are **SHA-256** and **SHA3-256**, which transform arbitrary input to 256-bit output.



Cryptographic Hash Functions - Examples

As an **example**, we can take the cryptographic hash function `SHA-256` and calculate the hash value of certain text message `hello` :

```
SHA-256("hello") =
"2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824"
```

There is no efficient algorithm to find the input message (in the above example `hello`) from its hash value (in the above example `2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824`). It is well-known that cryptographic hash functions **cannot be reversed** back, so they are used widely to encode an input without revealing it (e.g. encode a private key to a blockchain address without revealing the key).

As another **example**, we can take the cryptographic hash function `SHA3-512` and calculate the hash value of the same text message `hello` :

```
SHA3-512("hello") = "75d527c368f2efe848ecf6b073a36767800805e9eef2b1857d5f984f036eb6df891d75f72d9b154518c1cd58835286d1da9a38deba3de98b5a53e5ed78a84976"
```

Cryptographic Hash Functions - Live Demo

Play with most popular cryptographic hash functions **online**: <https://www.fileformat.info/tool/hash.htm>.

← → ↻ Secure | https://www.fileformat.info/tool/hash.htm

Hash Functions

Calculate a hash (aka message digest) of data. Implementations are from Sun (java.security.MessageDigest) and [GNU](#).

If you want to get the hash of a file in a form that is easier to use in automated systems, try the [online md5sum tool](#).

String hash

Text:

Hash

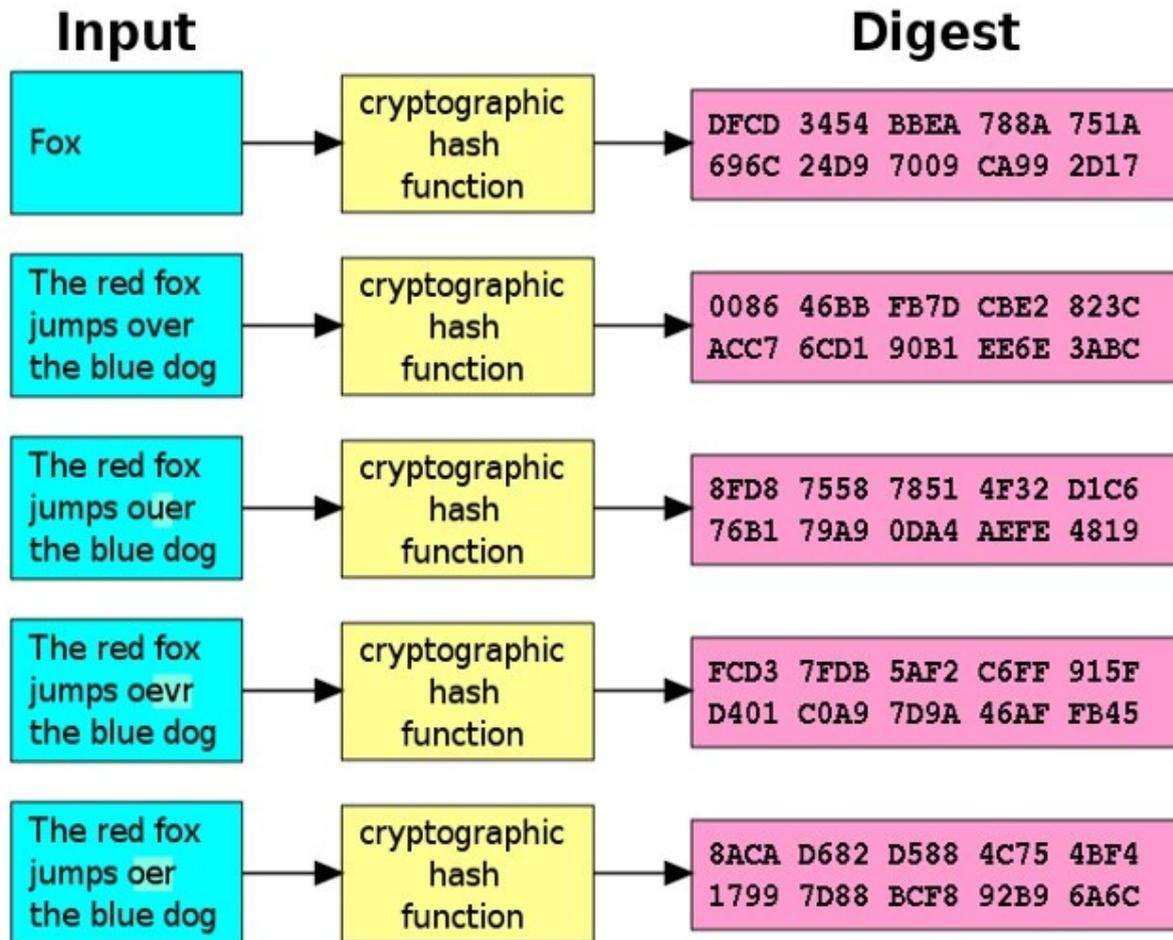
Results

Original text	hello
Original bytes	68:65:6c:6c:6f (length=5)
MD5	5d41402abc4b2a76b9719d911017c592
RipeMD160	108f07b8382412612c048d07d13f814118445acd
SHA-1	aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d
SHA-256	2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824

Cryptographic hash functions are widely used in cryptography, in computer programming and in blockchain systems.

Cryptographic Hash Functions and Collisions

Different input messages are expected to produce **different output** hash values (message digest).



Hash Collisions

Collision means the same hash value for two different inputs. For simple hash functions are easy to reach a collision. For example, assume a hash function $h(\text{text})$ sums of all character codes in a text. It will produce the same hash value (collision) for texts holding the same letters in different order, i.e. $h('abc') == h('cab') == h('bca')$. To avoid collisions, cryptographers have designed **collision-resistant** hash functions.

Cryptographic Hash Functions: No Collisions

Collisions in the cryptographic hash functions are **extremely unlikely** to happen, so crypto **hashes** are considered to almost uniquely identify their corresponding input. Moreover, it is extremely hard to find an input message that hashes to given value.

Cryptographic hash functions are **one-way hash functions**, which are **infeasible to invert**. The chance to find a collision for a strong cryptographic hash function (like SHA-256) is extremely little. Let's define this in more details:

- Let's have hash value $h = \text{hash}(p)$ for certain strong cryptographic hash function hash .
- It is expected to be **extremely hard** to find an input p' , such that $\text{hash}(p') = h$.
- For most modern strong cryptographic hash functions there are **no known collisions**.

The **ideal cryptographic hash function** should have the following properties:

- **Deterministic:** the same input message should always result in the same hash value.
- **Quick:** it should be fast to compute the hash value for any given message.
- **Hard to analyze:** a small change to the input message should totally change the output hash value.
- **Irreversible:** generating a valid input message from its hash value should be **infeasible**. This means that there should be no significantly better way than brute force (try all possible input messages).
- **No collisions:** it should be extremely hard (or practically impossible) to find two different messages with the same hash.

Modern cryptographic hash functions (like SHA2 and SHA3) match the above properties and are used widely in cryptography.

Cryptographic Hash Functions: Applications

Cryptographic hash functions (like SHA-256 and SHA3-256) are used in many scenarios. Let's review their most common applications.

Document Integrity

Verifying the **integrity** of files / documents / messages. E.g. a **SHA256 checksum** may confirm that certain file is original (not modified after its checksum was calculated).



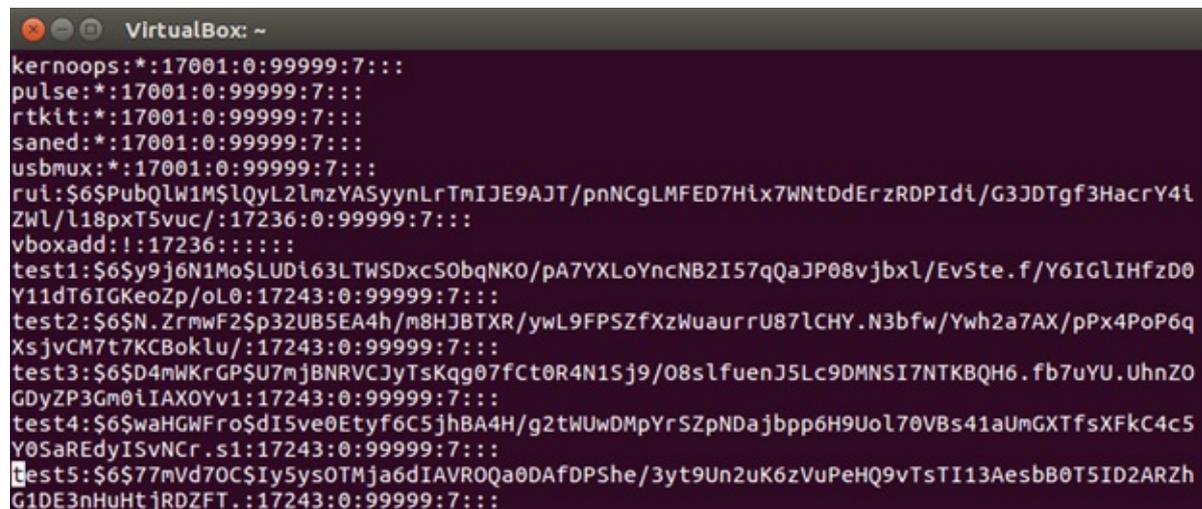
The screenshot shows a web browser window with the URL <https://www.openssl.org/source/>. Below the URL is a table listing files available for download, including their size in KBytes, the date and time they were last updated, and their SHA256 checksums. Each row also includes links for PGP signatures and SHA1 hashes.

KBytes	Date	File
8142	2018-Sep-11 13:21:29	openssl-1.1.1.tar.gz (SHA256) (PGP sign) (SHA1)
5213	2018-Aug-14 13:08:43	openssl-1.0.2p.tar.gz (SHA256) (PGP sign) (SHA1)
5325	2018-Aug-14 13:08:43	openssl-1.1.0i.tar.gz (SHA256) (PGP sign) (SHA1)
1457	2017-May-24 18:01:01	openssl-fips-2.0.16.tar.gz (SHA256) (PGP sign) (SHA1)
1437	2017-May-24 18:01:01	openssl-fips-ecp-2.0.16.tar.gz (SHA256) (PGP sign) (SHA1)

The above screenshot demonstrates how the **SHA256 checksums** ensure the **integrity** of the OpenSSL files at the official Web site of OpenSSL.

Storing Passwords

Storing **passwords** and verification of passwords. Instead of keeping a plain-text password in the database, developers usually keep **password hashes** or more complex values derived from the password (e.g. **Scrypt**-derived value).



The screenshot shows a terminal window with the following output, which represents the contents of the `/etc/shadow` file. Each line shows a username followed by a colon and a long string of characters representing a SHA-512 hash with salt.

```
kernoops:*:17001:0:99999:7:::
pulse:*:17001:0:99999:7:::
rtkit:*:17001:0:99999:7:::
saned:*:17001:0:99999:7:::
usbmux:*:17001:0:99999:7:::
rui:$6$PubQLW1M$1QyL2lmzYASyynLrTmIJE9AJT/pnNCgLMFED7Hlx7WNTDdErzRDPIdl/G3JDTgf3HacrY4lZWL/l18pxT5vuc/:17236:0:99999:7:::
vboxadd!:17236:::
test1:$6$y9j6N1Mo$LUdi63LTWSDxc50bqNK0/pA7YXLoYncNB2I57qQaJP08vjbXl/EvSte.f/Y6IGlIHfzD0Y11dT6IGKeoZp/oL0:17243:0:99999:7:::
test2:$6$N.ZrmwF2Sp32UB5EA4h/m8HJBTXR/ywL9FPSzFzXzWuaurrU87lCHY.N3bfw/Ywh2a7AX/pPx4PoP6qXsjvCM7t7KCBoklu/:17243:0:99999:7:::
test3:$6$D4mWkrGp$U7mjBNRVCJyTsKqg07fCt0R4N1Sj9/08slfuenJ5Lc9DMNSI7NTKBQH6.fb7uYU.UhnZ0GDyZP3Gm0iIAXOYv1:17243:0:99999:7:::
test4:$6$waHGWFro$D15ve0Etyf6C5jhBA4H/g2tWUwDmPyrSZpNDajbpb6H9UoL70VBs41aUmGXTfsXFkC4c5Y0SaREdyISvNcr.s1:17243:0:99999:7:::
test5:$6$77mVd70C$Iy5ys0THja6dIAVROQa0DAfDPShe/3yt9Un2uK6zVuPeHQ9vTsTI13AesbB0T5ID2ARZhG1DE3nHuHtjRDZFT.:17243:0:99999:7:::
```

The above example comes from the `/etc/shadow` file in a modern Linux system. The above passwords are stored as multiple-round SHA-512 hashes with salt.

Generate Unique ID

Generate an (almost) **unique ID** of certain document / message. Cryptographic hash functions almost uniquely identify documents based on their content. In theory **collisions are possible** with any cryptographic hash function, but are very unlikely to happen, so most systems (like **Git**) assume that the hash function they use is **collision free**.

Usually a document is **hashed** and the **document ID** (hash value) is used later to prove the existence of the document, or to retrieve the document from a storage system. Example of hash-based unique IDs are the commit hashes in **Git** and **GitHub**, based on the content of the commit (e.g.

3c3be25bc1757ca99aba55d4157596a8ea217698) and the **Bitcoin** addresses (e.g.

1BvBMSEYstWetqTFn5Au4m4GFg7xJaNVN2).

The screenshot shows the TortoiseGit interface for viewing commit logs. The selected commit is highlighted in red and its SHA-1 hash is displayed in a blue box. Below the log, a table lists the files added in this commit:

Path	Extension	Status	Lines added	Lines removed
cryptographic-hash-functions/crypto-hashes-and-collisions.md	.md	Added	30	0
cryptographic-hash-functions/exercises-calculate-hashes.md	.md	Added	41	0
cryptographic-hash-functions/hash-functions-applications.md	.md	Added	28	0

In the above example the SHA-1 unique ID identifies a certain commit in GitHub.

Pseudorandom Number Generation

Pseudorandom generation and key derivation. Hash values can serve as random numbers. A simple way to generate a random sequence is like this: start from a **random seed** (entropy collected from random events, such like keyboard clicks or mouse moves). Append "1" and calculate the hash to obtain the first random number, then append "2" and calculate the hash to obtain the second random number, etc. We shall give a Python example, implementing the described idea.

Proof-of-Work Algorithms

Proof-of-work (PoW) algorithms. Most proof-of-work algorithms calculate a hash value which is bigger than certain value (known as mining difficulty). To find this hash value, miners calculate billions of different hashes and take the biggest of them, because hash numbers are unpredictable. For example, the proof of work problem might be defined as follows: find a number p , such that $\text{hash}(x + p)$ holds 10 zero bits at its beginning.

Cryptographic Hashes are Part of Modern Programming

Cryptographic hash functions are so widely used, that they are often implemented as **build-in functions** in the standard libraries for the modern programming languages and platforms.

Secure Hash Algorithms

In the past, many **cryptographic hash algorithms** were proposed and used by software developers. Some of them were **broken** (like **MD5** and **SHA1**), some are still considered secure (like **SHA-2**, **SHA-3** and **BLAKE2**). Let's review the most widely used cryptographic hash functions (algorithms).

Secure Hash Functions

Modern cryptographic hash algorithms (like **SHA-3** and **BLAKE2**) are considered **secure** enough for most applications.

SHA-2, SHA-256, SHA-512

SHA-2 is a family of strong cryptographic hash functions: **SHA-256** (256 bits hash), **SHA-384** (384 bits hash), **SHA-512** (512 bits hash), etc. It is based on the cryptographic concept "**Merkle–Damgård construction**" and is considered **highly secure**. SHA-2 is published as official crypto standard in the United States.

SHA-2 is widely used by developers and in cryptography and is considered cryptographically strong enough for modern commercial applications.

SHA-256 is widely used in the **Bitcoin** blockchain, e.g. for identifying the transaction hashes and for the proof-of-work mining performed by the miners.

Examples of SHA2 hashes:

```
SHA-256('hello') = 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
SHA-384('hello') = 59e1748777448c69de6b800d7a33bbfb9fff1b463e44354c3553bcd9c666fa90125a3c79f
90397bdf5f6a13de828684f
SHA-512('hello') = 9b71d224bd62f3785d96d46ad3ea3d73319bfbcb2890caadae2dff72519673ca72323c3d99
ba5c11d7c7acc6e14b8c5da0c4663475c2e5c3adef46f73bcdec043
```

More Hash Bits == Higher Collision Resistance

By design, **more bits at the hash output** are expected to achieve **stronger security** and higher collision resistance (with some exceptions). As general rule, 128-bit hash functions are weaker than 256-bit hash functions, which are weaker than 512-bit hash functions.

Thus, SHA-512 is stronger than SHA-256, so we can expect that for SHA-512 it is more unlikely to practically find a collision than for SHA-256.

SHA-3, SHA3-256, SHA3-512, Keccak-256

SHA-3 (and its variants SHA3-224, SHA3-256, SHA3-384, SHA3-512), is considered **more secure than SHA-2** (SHA-224, SHA-256, SHA-384, SHA-512) for the same hash length. For example, SHA3-256 provides **more cryptographic strength than SHA-256** for the same hash length (256 bits).

The **SHA-3** family of functions are representatives of the "**Keccak**" hashes family, which are based on the cryptographic concept "**sponge construction**". Keccak is the winner of the **SHA-3 NIST competition**.

Unlike **SHA-2**, the **SHA-3** family of cryptographic hash functions are not vulnerable to the "**length extension attack**".

SHA-3 is considered **highly secure** and is published as official recommended crypto standard in the United States.

The hash function **Keccak-256**, which is used in the **Ethereum** blockchain, is a variant of SHA3-256 with some constants changed in the code.

The hash functions **SHAKE128(msg, length)** and **SHAKE256(msg, length)** are variants of the **SHA3-256** and **SHA3-512** algorithms, where the output message length can vary.

Examples of SHA3 hashes:

```
SHA3-256('hello') = 3338be694f50c5f338814986cdf0686453a888b84f424d792af4b9202398f392
Keccak-256('hello') = 1c8aff950685c2ed4bc3174f3472287b56d9517b9c948127319a09a7a36deac8
SHA3-512('hello') = 75d527c368f2efe848ecf6b073a36767800805e9eef2b1857d5f984f036eb6df891d75f7
2d9b154518c1cd58835286d1da9a38deba3de98b5a53e5ed78a84976
SHAKE-128('hello', 256) = 4a361de3a0e980a55388df742e9b314bd69d918260d9247768d0221df5262380
SHAKE-256('hello', 160) = 1234075ae4a1e77316cf2d8000974581a343b9eb
```

BLAKE2 / BLAKE2s / BLAKE2b

BLAKE / **BLAKE2** / **BLAKE2s** / **BLAKE2b** is a family of fast, highly secure cryptographic hash functions, providing calculation of 160-bit, 224-bit, 256-bit, 384-bit and 512-bit digest sizes, widely used in modern cryptography. BLAKE is one of the finalists at the [SHA-3 NIST competition](#).

The **BLAKE2** function is an improved version of **BLAKE**.

BLAKE2s (typically **256-bit**) is BLAKE2 implementation, performance-optimized for 32-bit microprocessors.

BLAKE2b (typically **512-bit**) is BLAKE2 implementation, performance-optimized for 64-bit microprocessors.

The **BLAKE2** hash function has similar security strength like SHA-3, but is less used by developers than SHA2 and SHA3.

Examples of BLAKE hashes:

```
BLAKE2s('hello') = 19213bacc58dee6dbde3ceb9a47cbb330b3d86f8cca8997eb00be456f140ca25
BLAKE2b('hello') = e4cfa39a3d37be31c59609e807970799caa68a19bfaa15135f165085e01d41a65ba1e1b14
6aeb6bd0092b49eac214c103ccfa3a365954bbbe52f74a2b3620c94
```

RIPEND-160

RIPEND-160 is a secure hash function, widely used in cryptography, e.g. in PGP and Bitcoin.

The **160-bit** variant of **RIPEND** is widely used in practice, while the other variations like RIPEND-128, RIPEND-256 and RIPEND-320 are not popular and have disputable security strengths.

As recommendation, **prefer using SHA-2 and SHA-3** instead of RIPEND, because they are more stronger than RIPEND, due to higher bit length and less chance for collisions.

Examples of RIPEND hashes:

```
RIPEND-160('hello') = 108f07b8382412612c048d07d13f814118445acd
RIPEND-320('hello') = eb0cf45114c56a8421fbc33430fa22e0cd607560a88bbe14ce70bdf59bf55b11a3906
987c487992
```

All of the above popular secure hash functions (SHA-2, SHA-3, BLAKE2, RIPEND) are not restricted by commercial patents and are **free for public use**.

Insecure Hash Functions

Old hash algorithms like **MD5**, **SHA-0** and **SHA-1** are considered **insecure** and were withdrawn due to **cryptographic weaknesses** (collisions found). **Don't use MD5, SHA-0 and SHA-1!** All these hash functions are proven to be cryptographically **insecure**.

You can find in Internet that **SHA1 collisions** can be practically generated and this results in algorithms for creating **fake digital signatures**, demonstrated by two different signed PDF documents which hold different content, but have the same hash value and the same digital signature. See <https://shattered.io>.

Avoid using of the following hash algorithms, which are considered **insecure** or have disputable security: **MD2**, **MD4**, **MD5**, **SHA-0**, **SHA-1**, **Panama**, **HAVAL** (disputable security, collisions found for HAVAL-128), **Tiger** (disputable, weaknesses found), **SipHash** (it is not a cryptographic hash function).

Other Secure Hash Functions

The below functions are popular strong cryptographic hash functions, alternatives to SHA-2, SHA-3 and BLAKE2:

- **Whirlpool** is secure cryptographic hash function, which produces 512-bit hashes.
- **SM3** is the crypto hash function, officialy standartized by the **Chinese government**. It is similar to SHA-256 (based on the Merkle–Damgård construction) and produces 256-bit hashes.
- **GOST** is secure cryptographic hash function, the Russian national standard. It produces 256-bit hashes.

The below functions are less popular alternatives to SHA-2, SHA-3 and BLAKE, finalists at the [SHA-3 NIST competition](#):

- **Skein** is secure cryptographic hash function, capable to derive 128, 160, 224, 256, 384, 512 and 1024-bit hashes.
- **Grøstl** is secure cryptographic hash function, capable to derive 224, 256, 384 and 512-bit hashes.
- **JH** is secure cryptographic hash function, capable to derive 224, 256, 384 and 512-bit hashes.

No Collisions for SHA-256, SHA3-256, BLAKE2s and RIPEMD-160 are Known

As of Oct 2018, **no collisions are known** for: **SHA256**, **SHA3-256**, **Keccak-256**, **BLAKE2s**, **RIPEMD160** and few others.

Brute forcing to find hash function collision as general costs: 2^{128} for SHA256 / SHA3-256 and 2^{80} for RIPEMD160.

Respectively, on a powerful enough **quantum computer**, it will cost less time: $2^{256/3}$ and $2^{160/3}$ respectively. Still (as of September 2018) so powerful quantum computers are not known to exist.

Learn more about cryptographic hash functions, their strength and **attack resistance** at:

<https://z.cash/technology/history-of-hash-function-attacks.html>

Hash Functions - Examples in Python

In this section we shall provide a few **examples** about calculating cryptographic hash functions in Python.

Calculating Cryptographic Hash Functions in Python

We shall use the standard Python library `hashlib`. The input data for hashing should be given as **bytes sequence** (bytes object), so we need to **encode the input string** using some text encoding, e.g. `utf8`. The produced **output data** is also a bytes sequence, which can be printed as hex digits using `binascii.hexlify()` as shown below:

```
import hashlib, binascii

text = 'hello'
data = text.encode("utf8")

sha256hash = hashlib.sha256(data).digest()
print("SHA-256: ", binascii.hexlify(sha256hash))

sha3_256 = hashlib.sha3_256(data).digest()
print("SHA3-256: ", binascii.hexlify(sha3_256))

blake2s = hashlib.new('blake2s', data).digest()
print("BLAKE2s: ", binascii.hexlify(blake2s))

ripemd160 = hashlib.new('ripemd160', data).digest()
print("RIPEMD-160:", binascii.hexlify(ripemd160))
```

The expected **output** from the above example looks like this:

```
SHA-256:      b'2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824'
SHA3-256:    b'3338be694f50c5f338814986cdf0686453a888b84f424d792af4b9202398f392'
BLAKE2s:     b'19213bacc58dee6dbde3ceb9a47cbb330b3d86f8cca8997eb00be456f140ca25'
RIPEMD-160: b'108f07b8382412612c048d07d13f814118445acd'
```

Calculating `Keccak-256` hashes (the hash function used in the Ethereum blockchain) requires non-standard Python functions. In the below example we use the `pycryptodome` package available from PyPI:

<https://pypi.org/project/pycryptodome>.

First install "pycryptodome" (<https://www.pycryptodome.org>)

```
pip install pycryptodome
```

Now write some Python code to calculate a **Keccak-256** hash:

```
from Crypto.Hash import keccak

keccak256 = keccak.new(data=data, digest_bits=256).digest()
print("Keccak256: ", binascii.hexlify(keccak256))
```

The **output** from the above examples is:

```
Keccak256:  b'1c8aff950685c2ed4bc3174f3472287b56d9517b9c948127319a09a7a36deac8'
```


Exercises: Calculate Hashes

In this exercise session, you are assigned to write some code to **calculate cryptographic hashes**. Write a program to **calculate hashes** of given text message: **SHA-224**, **SHA-256**, **SHA3-224**, **SHA3-384**, **Keccak-384** and **Whirlpool**. Write your code in programming language of choice.

Calculate SHA-224 Hash

Input	Output
hello	ea09ae9cc6768c50fcee903ed054556e5bfc8347907f12598aa24193

Calculate SHA-256 Hash

Input	Output
hello	2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824

Calculate SHA3-224 Hash

Input	Output
hello	b87f88c72702fff1748e58b87e9141a42c0dbedc29a78cb0d4a5cd81

Calculate SHA3-384 Hash

Input	Output
hello	720aea11019ef06440fbf05d87aa24680a2153df3907b23631e7177ce620fa1330ff07c0fddee54699a4c3ee0ee9d887

Calculate Keccak-384 Hash

Input	Output
hello	dcef6fb7908fd52ba26aaba75121526abbf1217f1c0a31024652d134d3e32fb4cd8e9c703b8f43e7277b59a5cd402175

Calculate Whirlpool (512 Bit) Hash

Input	Output
hello	0a25f55d7308eca6b9567a7ed3bd1b46327f0f1ffdc804dd8bb5af40e88d78b88df0d002a89e2fdbd5876c523f1b67bc44e9f87047598e7548298ea1c81cfd73

Hints: follow the Python examples, given earlier in this section or search in Internet.

Proof-of-Work Hash Functions: EThash and Equihash

Blockchain **proof-of-work mining** algorithms use a special class of hash functions which are **computational-intensive** and **memory-intensive**. These hash functions are designed to consume a lot of computational resources and a lot of memory and to be very hard to be implemented in a hardware devices (such as **FPGA** integrated circuits or **ASIC** miners). Such hash functions are known as "**ASIC-resistant**".

Many hash functions are designed for proof-of-work mining algorithms, e.g. **EThash**, **Equihash**, **CryptoNight** and **Cookoo Cycle**. These hash functions are **slow to calculate**, and usually use **GPU** hardware (**rigs** of graphics cards like NVIDIA GTX 1080) or powerful **CPU** hardware (like Intel Core i7-8700K) and a lot of fast **RAM** memory (like DDR4 chips). The goal of these mining algorithms is to **minimize the centralization of mining** by stimulating the small miners (home users and small mining farms) and limit the power of big players in the mining industry (who can afford to build giant mining facilities and data centers). A big number of **small players means better decentralization** than a small number of big players.

The main weapon in the hands of the big mining corporations is considered the **ASIC miners**, so the design of modern cryptocurrencies and usually includes proof-of-work mining using an **ASIC-resistant hashing algorithm** or **proof-of-stake** consensus protocol.

EThash

Let's explain in brief the idea behind the **EThash** proof-of-work mining hash function used in the Ethereum blockchain.

- **EThash** is the proof-of-work hash function in the Ethereum blockchain. It is **memory-intensive** hash-function (requires a lot of RAM to be calculated fast), so it is believed to be **ASIC-resistant**.

How does EThash work?

- A "**seed**" is computed for each block (based on the entire chain until the current block).
- From the seed, a **16 MB pseudorandom cache** is computed.
- From the cache, a **1 GB dataset** is extracted to be used in mining.
- Mining involves hashing together random slices of the dataset.

Learn more about **EThash** at: <https://github.com/ethereum/wiki/wiki/Ethash>, <https://github.com/lukovkin/ethash>.

Equihash

Let's explain in briefly the idea behind the **Equihash** proof-of-work mining hash function used in Zcash, Bitcoin Gold and a few other blockchains.

- **Equihash** is the proof-of-work hash function in the Zcash and Bitcoin Gold blockchains. It is **memory-intensive** hash-function (requires a lot of RAM for fast calculation), so it is believed to be **ASIC-resistant**.

How does **Equihash** work?

- Uses **BLAKE2b** to compute **50 MB hash dataset** from the previous blocks in the blockchain (until the current block).
- Solves the "**Generalized Birthday Problem**" over the generated hash dataset (pick 512 different strings from 2097152, such that the binary XOR of them is zero). The best known solution (Wagner's algorithm) runs in exponential time, so it requires a lot of memory-intensive and computing-intensive calculations.
- **Double SHA256** the solution to compute the final hash.

Learn more about **Equihash** at: <https://www.cryptolux.org/images/b/b9/Equihash.pdf>, <https://github.com/tromp/equihash>.

More about ASIC-Resistant Hash Functions

Lear more about the **ASIC-resistant hash functions** at: <https://github.com/ifdefelse/ProgPOW>.

MAC Codes and Key Derivation Functions

Message authentication codes (MAC), **HMAC** (hash-based message authentication code) and **KDF** (key derivation functions) play important role in cryptography. Let's explain when we need **MAC**, how to calculate **HMAC** and how it is related to key derivation functions.

Message Authentication Code (MAC)

Message Authentication Code (MAC) is cryptographic code, calculated by given **key** and given **message**:

```
auth_code = MAC(key, msg)
```

Typically, it behaves **like a hash function**: a minor change in the message or in the key results to totally different **MAC value**. It should be practically infeasible to change the key or the message and get the same **MAC value**. MAC codes, like hashes, are **irreversible**: it is impossible to recover the original message or the key from the MAC code.

The MAC code is **digital authenticity code**, like a **digital signature**, but with **pre-shared key**. We shall learn more about digital signing and digital signatures later.

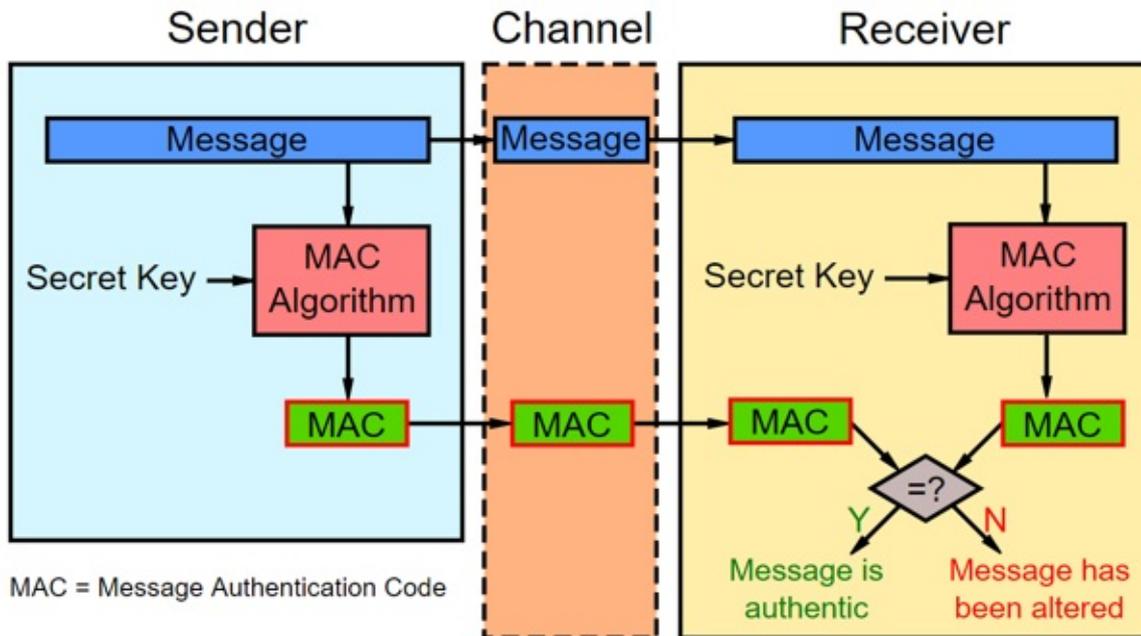
MAC Algorithms

Many **algorithms** for calculating message authentication codes (MAC) exist in modern cryptography. The most popular are based on **hashing** algorithms, like **HMAC** (Hash-based MAC, e.g. HMAC-SHA256) and **KMAC** (Keccak-based MAC). Others are based on **symmetric ciphers**, like **CMAC** (Cipher-based MAC), **GMAC** (Galois MAC) and **Poly1305** (Bernstein's one-time authenticator). Other MAC algorithms include **UMAC** (based on universal hashing), **VMAC** (high-performance block cipher-based MAC) and **SipHash** (simple, fast, secure MAC).

When We Need MAC Codes?

A sample scenario for using MAC codes is like this:

- Two parties exchange somehow a certain secret **MAC key** (pre-shared **key**).
- We receive a **msg** + **auth_code** from somewhere (e.g. from Internet, from the blockchain, or from email message).
- We want to be sure that the **msg** is **not tampered**, which means that both the **key** and **msg** are correct and match the MAC code.
- In case of **tampered message**, the MAC code will be incorrect.



Authenticated Encryption: Encrypt / Decrypt Messages using MAC

Another scenario to use **MAC codes** is for **authenticated encryption**: when we **encrypt a message** and we want to be sure the **decryption password is correct** and the decrypted message is the same like the original message before encryption.

- First, we **derive a key** from the password. We can use this key for the MAC calculation algorithm (directly or hashed for better security).
- Next, we **encrypt the message** using the derived key and store the ciphertext in the output.
- Finally, we calculate the **MAC code** using the derived key and the original message and we append it to the output.

When we **decrypt the encrypted message** (ciphertext + MAC), we proceed as follows:

- First, we **derive a key** from the password, entered by the user. It might be the correct password or wrong. We shall find out later.
- Next, we **decrypt the message** using the derived key. It might be the original message or incorrect message (depends on the password entered).
- Finally, we calculate a **MAC code** using the derived key + the decrypted message.
 - If the calculated MAC code matches the MAC code in the encrypted message, the **password is correct**.
 - Otherwise, it will be proven that the decrypted message is not the original message and this means that the **password is incorrect**

Some **authenticated encryption algorithms** (such as **AES-GCM** and **ChaCha20-Poly1305**) integrate the MAC calculation into the encryption algorithm and the MAC verification into the decryption algorithm. We shall learn more about these algorithms later.

The MAC is stored along with the ciphertext and it **does not reveal** the password or the original message. Storing the MAC code, visible to anyone is safe, and after decryption, we know whether the message is the original one or not (wrong password).

MAC-Based Pseudo-Random Generator

Another application of MAC codes is for **pseudo-random generator** functions. We can start from certain **salt** (constant number or the current date and time or some other randomness) and some **seed** number (last random number generated, e.g. **0**). We can calculate the **next_seed** as follows:

```
next_seed = MAC(salt, seed)
```

This **next pseudo-random number** is "randomly changes" after each calculation of the above formula and we can use it to generate the next random number in certain range.

HMAC and Key Derivation Functions (KDF)

Simply calculating `hash_func(key + msg)` to obtain a MAC (message authentication code) is considered **insecure** (see the [details](#)). It is recommended to use the **HMAC algorithm instead**, e.g. `HMAC-SHA256` or `HMAC-SHA3-512` or other secure MAC algorithm.

What is HMAC?

HMAC = Hash-based Message Authentication Code (MAC code, calculated using a cryptographic hash function):

```
HMAC(key, msg, hash_func) -> hash
```

The results MAC code is a **message hash** mixed with a secret key. It has the cryptographic properties of hashes: **irreversible, collision resistant**, etc.

The `hash_func` can be any cryptographic hash function like `SHA-256`, `SHA-512`, `RIPEMD-160`, `SHA3-256` or `BLAKE2s`.

HMAC is used for message **authenticity**, message **integrity** and sometimes for **key derivation**.

Key Derivation Functions (KDF)

Key derivation function (KDF) is a function which transforms a variable-length password to fixed-length key (sequence of bits):

```
function(password) -> key
```

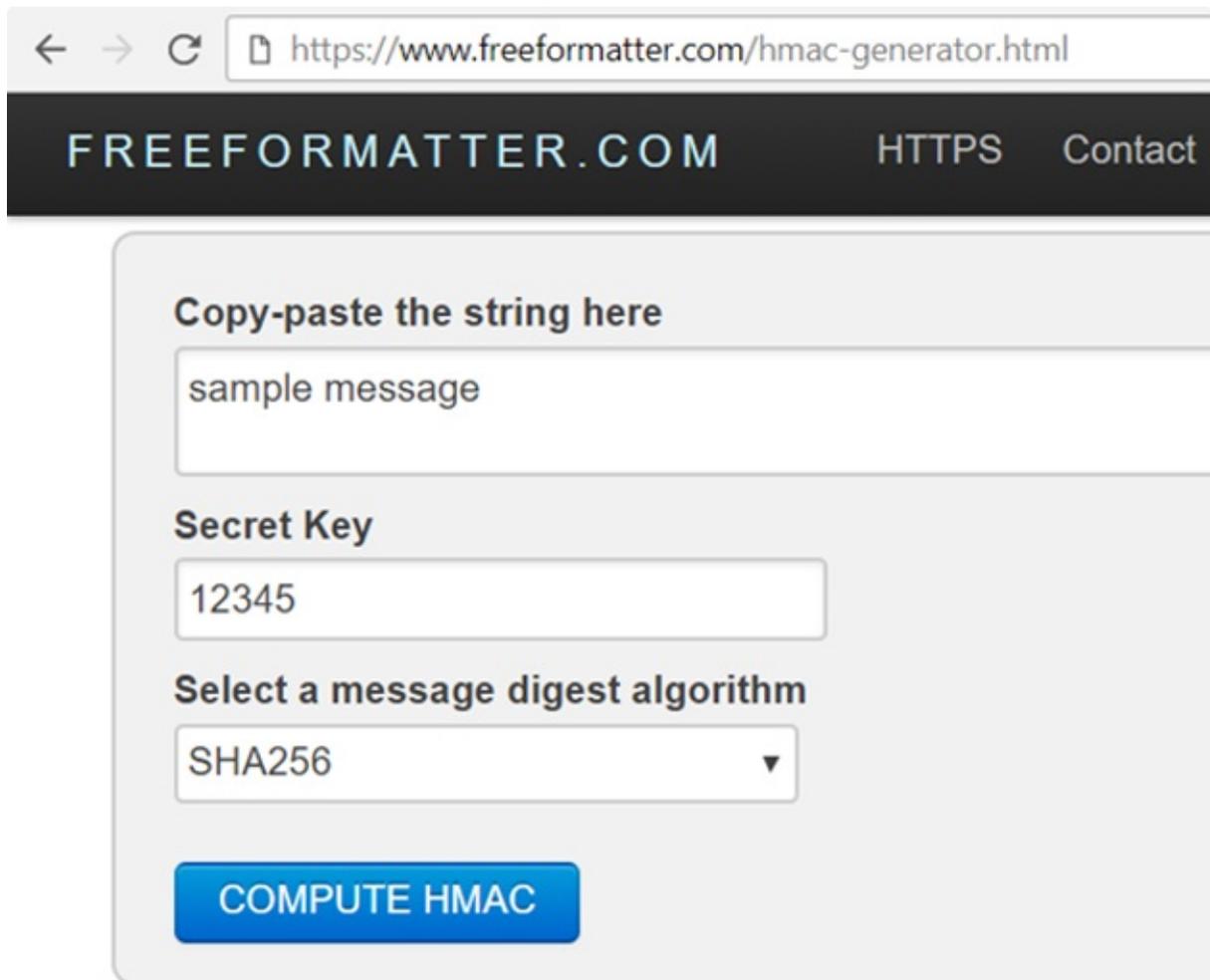
As **very simple KDF function**, we can use `SHA256`: just hash the password. Don't do this, because it is **insecure**. Simple hashes are vulnerable to **dictionary attacks**.

As more complicated KDF function, you can derive a password by calculating **HMAC(salt, msg, SHA256)** using some random value called "**salt**", which is stored along with the derived key and used later to derive the same key again from the password.

Using **HKDF (HMAC-based key derivation)** for key derivation is **less secure** than modern KDFs, so experts recommend using stronger key derivation functions like `PBKDF2`, `Bcrypt`, `Scrypt` and `Argon2`. We shall discuss all these KDF functions later.

HMAC Calculation - Example

To get a better idea of **HMAC** and how it is calculated, try this online tool: <https://www.freeformatter.com/hmac-generator.html>



← → ↻ <https://www.freeformatter.com/hmac-generator.html>

FREEFORMATTER.COM HTTPS Contact

Copy-paste the string here

Secret Key

Select a message digest algorithm

COMPUTE HMAC

Play with calculating **HMAC('sample message', '12345', 'SHA256')**:

```
HMAC('sample message', '12345', 'SHA256') =  
'ee40ca7bc90df844d2f5b5667b27361a2350fad99352d8a6ce061c69e41e5d32'
```

Try the above example yourself.

HMAC Calculation - Examples in Python

In **Python** we can **calculate HMAC** codes as follows (using the `hashlib` and `hmac` libraries):

```
import hashlib, hmac, binascii

def hmac_sha256(key, msg):
    return hmac.new(key, msg, hashlib.sha256).digest()

key = b"12345"
msg = b"sample message"
print(binascii.hexlify(hmac_sha256(key, msg)))
```

The above code will calculate and print the expected HMAC code (like in our previous example):

```
ee40ca7bc90df844d2f5b5667b27361a2350fad99352d8a6ce061c69e41e5d32
```

Try the code yourself and play with it.

Exercises: Calculate HMAC

Write a program to **calculate HMAC-SHA-384** of given text **message** by given **key**. Write your code in programming language of choice.

Input	Output
Message: he llo Key: cryptogr aphy	83d1c3d3774d8a32b8ea0460330c16d1b2e3e5c0ea86ccc2d70e603aa8c8151d675dfe339d8 3f3f495fab226795789d4
Message: he llo Key: again	4c549a549aa037e0fb651569bf271faa23cfa20e8a9d21438a6ff5bf6be916bebdbaa48001e0cd 6941ec74cd02be70e5

Hints: follow the Python examples, given earlier in this section or search in Internet.

Key Derivation Functions (KDF): Deriving a Key from Password

Now let's explain in details how to **securely derive a key from a password** and the most popular **key derivation functions (KDFs)** used in practice: [PBKDF2](#), [Bcrypt](#), [Scrypt](#) and [Argon2](#).

[TODO: explain the Linux crypt: SHA-512 key derivation]

We shall discuss the strong and weak sides of the above mentioned KDFs and when to use them.

Key Derivation Functions - Concepts

In cryptography we often use **passwords** instead of **binary keys**, because passwords are easier to remember, to write down and can be shorter.

When a certain algorithm needs a **key** (e.g. for encryption or for digital signing) a **key derivation function** (password -> key) is needed.

We already noted that using `SHA-256(password)` as key-derivation is insecure! It is vulnerable to many attacks: **brute-forcing**, **dictionary attacks**, **rainbow attacks** and others, which may reverse the hash in practice and attacker can obtain the password.

Cryptographic Key Derivation Functions

[PBKDF2](#), [Bcrypt](#), [Scrypt](#) and [Argon2](#) are significantly stronger key derivation functions and are designed to survive password guessing (brute force) attacks.

By design **secure key derivation functions** use **salt** (random number, which is different for each key derivation) + **many iterations** (to speed-down eventual password guessing process). This is a process, known as **key stretching**.

To calculate a secure KDF it takes some **CPU time** to derive the key (e.g. 0.2 sec) + some **memory (RAM)**. Thus deriving the key is "computationally expensive", so password cracking will also be computationally expensive.

When a modern KDF function is used with appropriate config parameters, **cracking passwords** will be **slow** (e.g. 5-10 attempts per second, instead of thousands or millions attempts per second).

All of the above mentioned key-derivation algorithms ([PBKDF2](#), [Bcrypt](#), [Scrypt](#) and [Argon2](#)) are not patented and **royalty-free** for public use.

Let's learn more about these modern KDF.

PBKDF2: Derive Key from Password

PBKDF2 is a simple cryptographic key derivation function, which is resistant to [dictionary attacks](#) and [rainbow table attacks](#). It is based on iteratively deriving **HMAC** many times with some padding. The **PBKDF2** algorithm is described in the Internet standard [RFC 2898 \(PKCS #5\)](#).

PBKDF2 takes several **input parameters** and produces the derived **key** as output:

```
key = pbkdf2(password, salt, iterations-count, hash-function, derived-key-len)
```

Technically, the **input data** for **PBKDF2** consists of:

- `password` – array of bytes / string, e.g. "p@\$Sw0rD~3" (8-10 chars minimal length is recommended)
- `salt` – securely-generated random bytes, e.g. "df1f2d3f4d77ac66e9c5a6c3d8f921b6" (minimum 64 bits, 128 bits is recommended)
- `iterations-count`, e.g. 1024 iterations
- `hash-function` for calculating **HMAC**, e.g. `SHA256`
- `derived-key-len` for the output, e.g. 32 bytes (256 bits)

The **output data** is the **derived key** of requested length (e.g. 256 bits).

PBKDF2 and Number of Iterations

PBKDF2 allows to configure the number of **iterations** and thus to configure the time required to derive the key.

- **Slower key derivation** means high login time / slower decryption / etc. and **higher resistance** to password cracking attacks.
- **Faster key derivation** means short login time / faster decryption / etc. and **lower resistance** to password cracking attacks.
- **PBKDF2** is **not resistant** to [GPU attacks](#) (parallel password cracking using video cards) and to [ASIC attacks](#) (specialized password cracking hardware). This is the main motivation behind more modern KDF functions.

PBKDF2 - Example

Try **PBKDF2 key derivation** online here: <https://asecuritysite.com/encryption/PBKDF2z>.

Try to **increase the iterations count** to see how this affects the speed of key derivation.

PBKDF2 Calculation in Python - Example

Now, we shall write some **code in Python** to derive a key from a password using the **PBKDF2** algorithm.

First, install the Python package `backports.pbkdf2` using the command:

```
pip install backports.pbkdf2
```

Now, write the Python code to calculate PBKDF2:

```
import os, binascii
from backports.pbkdf2 import pbkdf2_hmac

salt = binascii.unhexlify('aaef2d3f4d77ac66e9c5a6c3d8f921d1')
passwd = "p@$Sw0rD~1".encode("utf8")
key = pbkdf2_hmac("sha256", passwd, salt, 50000, 32)
print("Derived key:", binascii.hexlify(key))
```

The **PBKDF2** calculation function takes several **input parameters**: **hash function** for the HMAC, the **password** (bytes sequence), the **salt** (bytes sequence), **iterations** count and the output **key length** (number of bytes for the derived key).

The **output** from the above code execution is the following:

```
Derived key: b'52c5efa16e7022859051b1dec28bc65d9696a3005d0f97e506c42843bc3bdbc0'
```

Try to change the number of **iterations** and see whether and how the **execution time** changes.

When to Use PBKDF2?

Today **PBKDF2** is considered old-fashioned and less secure than modern KDF functions, so it is recommended to use **Bcrypt**, **Scrypt** or **Argon2** instead. We shall explain all these KDF functions in details later in this section.

Modern KDFs: Bcrypt, Scrypt and Argon2

PBKDF2 has a major weakness: it is **not GPU-resistant** and **not ASIC-resistant**, because it uses relatively small amount of RAM and can be efficiently implemented on GPU (graphics cards) or **ASIC** (specialized hardware).

Modern key-derivation functions (KDF) like **Scrypt** and **Argon2** are designed to be **resistant to dictionary attacks, GPU attacks and ASIC attacks**. These functions derive a key (of fixed length) from a password (text) and need a lot memory (RAM), which does not allow fast parallel computations on GPU or ASIC hardware.

Algorithms like **Bcrypt, Scrypt and Argon2** are considered more **secure** KDF functions. They use **salt** + many **iterations** + a lot of **CPU** + a lot of **RAM** memory and this makes very hard to design a custom hardware to significantly speed up password cracking.

It takes a lot of **CPU time** to derive the key (e.g. 0.2 sec) + a lot of **RAM memory** (e.g. 1GB). The calculation process is memory-dependent, so **the memory access is the bottleneck** of the calculations. Faster RAM access will speed-up the calculations.

When a lot of CPU and RAM is used to derive the key from given password, **cracking passwords is slow** and inefficient (e.g. 5-10 attempts / second), even when using very good password cracking hardware and software. The goal of the modern KDF functions is to make practically infeasible to perform a brute-force attack to reverse the password from its hash.

Let's discuss in more details **Scrypt, Bcrypt and Argon2**.

Scrypt Key Derivation

Scrypt (RFC 7914) is a strong cryptographic key-derivation function (KDF). It is memory-intensive, designed to prevent **GPU**, **ASIC** and **FPGA** attacks (highly efficient password cracking hardware).

The **Scrypt** algorithm takes several **input parameters** and produces the derived **key** as output:

```
key = Scrypt(password, salt, N, r, p, derived-key-len)
```

Scrypt Parameters

The **Scrypt config parameters** are:

- **N** – iterations count (affects memory and CPU usage), e.g. 16384
- **r** – block size (affects memory and CPU usage), e.g. 8
- **p** – parallelism factor (threads to run in parallel - affects the memory, CPU usage), usually 1
- **password** – the input password (8-10 chars minimal length is recommended)
- **salt** – securely-generated random bytes (64 bits minimum, 128 bits recommended)
- **derived-key-length** - how many bytes to generate as output, e.g. 32 bytes (256 bits)

The **memory** in Scrypt is accessed in strongly **dependent order** at each step, so the memory access speed is the algorithm's bottleneck. The **memory required** to compute Scrypt key derivation is calculated as follows:

```
Memory required = 128 * N * r * p bytes
```

Example: e.g. $128 * N * r * p = 128 * 16384 * 8 * 1 = 16 \text{ MB}$

Choosing parameters depends on how much you want to wait and what level of security (password cracking resistance) do you want to achieve:

- Sample parameters for **interactive login**: $N=16384, r=8, p=1$ (RAM = 16MB). For interactive login you most probably do not want to wait more than a 0.5 seconds, so the computations should be very slow. Also at the server side, it is usual that many users can login in the same time, so slow Scrypt computation will slow down the entire system.
- Sample parameters for **file encryption**: $N=1048576, r=8, p=1$ (RAM = 1GB). When you encrypt your hard drive, you will unlock the encrypted data in rare cases, usually not more than 2-3 times per day, so you may want to wait for 2-3 seconds to increase the security.

You can perform tests and choose the Scrypt parameters yourself during the design and development of your app or system. Always try to use the **fastest possible implementation of Scrypt** for your language and platform, because crackers will definitely use it. Some implementations (e.g. in Python) may be 100 times slower than the fastest ones!

In the **MyEtherWallet** crypto wallet, the default Scrypt parameters are $N=8192, r=8, p=1$. These settings are not strong enough for crypto wallets, but this is how it works. The solution is to use long and complex password to avoid password cracking attacks.

Scrypt - Example

You can play with **Scrypt** key derivation online here: <https://8gwifi.org/scrypt.jsp>.

Script Calculation in Python - Example

Now, we shall write some **code in Python** to derive a key from a password using the **Script** algorithm.

First, install the Python package `script` using the command:

```
pip install script
```

Now, write the Python code to calculate Script:

```
import script, binascii

salt = binascii.unhexlify('aa1f2d3f4d23ac44e9c5a6c3d8f9ee8c')
passwd = "p@$w0rD~7".encode("utf8")
key = script.hash(passwd, salt, 16384, 8, 1, 32)
print("Derived key:", binascii.hexlify(key))
```

The **Script** calculation function takes several **input parameters**: the **password** (bytes sequence), the **salt** (bytes sequence), **iterations** count, **block size** for each iteration, **parallelism** factor and the output **key length** (number of bytes for the derived key).

The **output** from the above code execution is the following:

```
Derived key: b'1660a97efe9ea0ee1cc843e99cc00c1643925ea5b8096371950ec5c2f4c3fe48'
```

Try to change the number of **iterations** or the **block size** and see how they affect the **execution time**. Have in mind that the above Python implementation is not very fast. You may find fast Script implementation in Internet.

Storing Algorithm Settings + Salt + Hash Together

In many applications, frameworks and tools, **Script encrypted passwords are stored together with the algorithm settings and salt**, into a single string (in certain format), consisting of several parts, separated by \$ character. For example, the password `p@ss~123` can be stored in the Script standard format like this (several examples are given, to make the pattern apparent):

```
16384$8$1$kytG1MHY1KU=$afc338d494dc89be40e317788e3cd9166d066709db0e6481f0801bd918710f46
16384$8$1$5gFG1ElztY0=$560f6229356c281a525fad4e2fc4c209bb55c21dec789381335a32bb84888a5a
32768$8$4$VGh1IHF1aWo=$54d657cec8b3aaca675b407e790bccf1dddb0a23665cd5f994820a736d4b58ba
```

When to Use Script?

When configured properly **Script** is considered a highly secure KDF function, so you can use it as general purpose password to key derivation algorithm, e.g. when encrypting wallets, files or app passwords.

Bcrypt Key Derivation

Bcrypt is another cryptographic KDF function, **older than Script**, and is **less resistant** to ASIC and GPU attacks. It provides configurable iterations count, but uses constant memory, so it is easier to build hardware-accelerated password crackers.

Bcrypt - Example

You can play with **Bcrypt** here: <https://www.dailycred.com/article/bcrypt-calculator>.

BCrypt Calculator

p@ss~123

Enter an example password to hash.

7

Select the maximum number of rounds which is tolerable, performance-wise, for your application. Bcrypt can support up to 31 rounds, but this demo cannot go above 12.

Calculate

\$2a\$07\$nvY0jT4D16W2a1dUbBYNMuWdDqizOusiefDjeks7F3GUH3bGYu91i

Storing Algorithm Settings + Salt + Hash Together

In many applications, frameworks and tools (e.g. in the database of WordPress sites), **Bcrypt encrypted passwords are stored together with the algorithm settings and salt**, into a single string (in certain format), consisting of several parts, separated by \$ character. For example, the password `p@ss~123` can be stored in the Bcrypt encrypted format like this (several examples are given, to make the pattern apparent):

```
$2a$07$wHidrK40LB0vk9r3fiseeYjQaCZ0bIeKY9qLsNep/I2nZAXb0b7m
$2a$12$UqBxs0PN/u106Fio1.FnD0hSRJztLz364AwpGemp1jt80nJYNSr.e
$2a$12$80v41fmZZbv805YKrXXCu.mdH9Dq9r72C5GnhVZbGnsIzTr8dSUfm
```

When to Use Bcrypt?

When configured properly **Bcrypt** is considered a **secure KDF function** and is widely used in practice. It is considered that **Script is more secure than Bcrypt**, so modern applications should **prefer Script** (or **Argon2**) instead of **Bcrypt**. Still, this recommendation is disputable, but I personally prefer **Argon2**.

The Linux crypt() KDF Function from glibc

... ..

[TODO: write a few words about the crypt() function in Linux]

See [https://en.wikipedia.org/wiki/Crypt_\(C\)](https://en.wikipedia.org/wiki/Crypt_(C))

... ..

Argon2: Secure, ASIC-Resistant KDF

Argon2 is modern **ASIC-resistant** and **GPU-resistant** secure key derivation function. It has better password cracking resistance (when configured correctly) than **PBKDF2**, **Bcrypt** and **Scrypt** (for similar configuration parameters for CPU and RAM usage).

Variants of Argon2

The **Argon2** function has several variants:

- **Argon2d** – provides strong GPU resistance, but has potential side-channel attacks (possible in very special situations).
- **Argon2i** – provides less GPU resistance, but has no side-channel attacks.
- **Argon2id** – **recommended** (combines the Argon2d and Argon2i).

Config Parameters of Argon2

Argon2 has the following **config parameters**, which are very similar to Scrypt:

- **password** P : the password (or message) to be hashed
- **salt** S : random-generated salt (16 bytes recommended for password hashing)
- **iterations** t : number of iterations to perform
- **memorySizeKB** m : amount of memory (in kilobytes) to use
- **parallelism** p : degree of parallelism (i.e. number of threads)
- **outputKeyLength** T : desired number of returned bytes

Argon2 - Example

You can **play with the Argon2** password to key derivation function online here: <http://antelle.net/argon2-browser>.

← → ↻ Not secure antelle.net/argon2-browser/

Argon2 in browser

Argon2 is new password hashing function, the winner of Password Hashing Competition.
Here, Argon2 library is compiled for browser runtime. [Statistics](#), [js library](#), [source and docs on GitHub](#).

Password

Salt

Memory KiB

Iterations

Hash length

Parallelism

Type Argon2d Argon2i

Result

```
[00.000] Testing Argon2 using PNaCl
[00.478] Encoded:
$argon2d$v=19$m=4096,t=50,p=1$c29tZSBzYWx0IDEyMyBhYmM$ms082wbPJVwr90gCTfwavgMqqsHdNKjyesHLRXjAAyc
[00.478] Hash: 9ac3bcd06cf255c2bf4e8024dfc1abe031baac1dd34a8f27ac1cb4578c00327
[00.478] Elapsed: 478ms
```

Argon2 Calculation in Python - Example

Now, we shall write some **code in Python** to derive a key from a password using the **Argon2** algorithm.

First, install the Python package `argon2_cffi` using the command:

```
pip install argon2_cffi
```

Now, write the Python code to calculate Argon2:

```
import argon2

argon2Hasher = argon2.PasswordHasher(time_cost=50, memory_cost=102400, parallelism=8, hash_1
en=32, salt_len=16)
hash = argon2Hasher.hash("s3kr3tp4ssw0rd")
print("Derived key:", hash)
```

The **Argon2** calculation takes several **input configuration settings**: **time_cost** (number of iterations), **memory_cost** (memory to use in KB), **parallelism** (how many parallel threads to use), **hash_len** (the size of the derived key), **salt_len** (the size of the random generated salt, typically 128 bits / 16 bytes).

Sample **output** from the above code execution:

```
Derived key: $argon2id$v=19$m=102400,t=50,p=8$JPoIjwAPECGiLFwdhcCMwQ$Mf9d8TtMA7b21/8VTyW+zEY
lzMo2TyPc1kf4qnNUzCI
```

Note that the above output is not the derived key, but a **hash string** in a standardized format, which holds the Argon2 algorithm config **parameters** + the derived **key** + the random **salt**. By design, the salt and the derived key should be different at each code execution.

Try to change the **time_cost** or the **memory_cost** settings and see how they affect the **execution time** for the key derivation.

Storing Algorithm Settings + Salt + Hash Together

In many applications, frameworks and tools, **Argon2 encrypted passwords are stored together with the algorithm settings and salt**, into a single string (in certain format), consisting of several parts, separated by `$` character. For example, the password `p@ss~123` can be stored in the Argon2 standard format like this (several examples are given, to make the pattern apparent):

```
$argon2d$v=19$m=1024,t=16,p=4$c2FsdDEyM3NhbHQxMjM$2dVtFVPCezhvjtYu2PaeX0eBR+RUZ6SqhtD/+QF4F1o
$argon2d$v=19$m=1024,t=16,p=4$YW5vdGh1cnNhbHRhbm90aGVyc2FsdA$KB7Nj7kK21YdGeEBQy7R3vKkYCz1cdR/I3QcArMh1/Q
$argon2i$v=19$m=8192,t=32,p=1$c21hbGxzYWx0$lm01aPPy3x0CcvrKpFLi1TL/uSVJ/e05hPHiWZFaWvY
```

When to Use Argon2?

When configured properly **Argon2** is considered a highly secure KDF function, **one of the best** available in the industry, so you can use it as general purpose password to key derivation algorithm, e.g. to when encrypting wallets, documents, files or app passwords. In the general case **Argon2 is recommended** over **Scrypt**, **Bcrypt** and **PBKDF2**.

Password Encryption: Encrypting User Passwords

In software development we constantly use **password-based user authentication**. For example, if we have a Web site, we typically have admin panel, accessible after **login**, based on **username + password**.

Developers often need to keep **user passwords** in the database for their sites, apps or other systems. There are many ways to implement **password-based authentication**. Let's review them and discuss the good and bad practices.

Clear-Text Passwords - Never Do Anti-Pattern

The easiest and **most highly insecure** method for password-based authentication is to use **clear-text passwords** written directly in the database.

- **Never do this!!!** It is anti-pattern for software development. It is **bad for many reasons**.
- To **check the password**, just compare the password for checking with password from the database.
- Admins will be able to see user's passwords, but some users use the same password for GMail, Facebook, Twitter, etc. **Admins should never know user's passwords**, but should be able to change them in case of emergency.
- Another problem is that if someone hacks the server and gain access to the database, he will **see all user's passwords** in plaintext.
- It is **very bad practice** to keep plaintext passwords in any information system / app in the world!

Simple Password Hash - Highly Insecure

A relative easy and **relatively insecure** method for password-based authentication is to use **password hash** like SHA-256(password), written directly in the database.

- **Avoid this!** It is highly **insecure** method. Why? Because hashes are vulnerable to **dictionary attacks**.
- To **check the password**, just compare the hash(password for checking) with the password hash from the database.
- Crackers who gain access to the database, can use a **dictionary** holding the hashes of the most commonly used 10 million passwords and most passwords will be decrypted. The dictionary attack process is **extremely fast**, because it compares the hashes from the dictionary with the password hash (trivial **string compare**).
- Search in Internet for [free dictionaries / wordlists for dictionary attack](#).

Salted Hashed Passwords - Secure, but Not Enough

More complicated and **relatively secure** method for password-based authentication is to use **salted hashed passwords**, written in the database as pair { **salt + hash(password + salt)** }. The hash function can be any cryptographic hash like SHA-256.

- The idea is to keep different random **salt**, along with different **password hash**, changed every time, when the password is written in the database. Thus the same password is encrypted every time as different ciphertext { **salt + hash** }.
- To **check the password**, **calculate the hash** from the password for checking with the **salt** from the database. Compare the **calculated hash** with the **hash from the database**.
- This method works well to prevent dictionary attacks, but does not prevent **GPU-based** and **ASIC-based** password cracking attacks. It has also the same **security problems** like using hash(key + msg) instead of HMAC(key, msg), e.g. [length-extension attack](#).
- Basically keeping **salted hashed passwords** is more secure than the previous ones, but still **avoid it**. Just use better password hashing function instead of simple hash.

Secure KDF-Based Password Hashing - Recommended

The most complicated and **most secure** method for password-based authentication is to use **KDF-based password hash**, written in the database as pair { **salt + KDF(password, salt)** }. The **key-derivation function** (KDF) should be strong and secure, e.g. **Scrypt** or **Argon2** with carefully selected parameters.

- The idea is to keep different random **salt** for each encrypted password, along with the **key** derived by a secure KDF-function, such as **Scrypt** or **Argon2** (with reasonable number of iterations and RAM consumption settings).
- To **check the password**, take the **salt** from the database and **derive a key** from the password for checking, using the same KDF function and KDF parameters like when the password was stored in the database. Compare the **derived key** with the **key from the database**.
- This method is **resistant to most attacks** and is considered as standard in the software industry. It is as **secure** as the KDF function with the selected KDF parameters.

Conclusion: use secure KDF functions like **Argon2** and **Scrypt** to keep encrypted passwords in the database. Never use plain-text passwords!

Exercises: Encrypt Passwords for User Register / Login

...

Secure Random Number Generators, PRNG and CSPRNG

In cryptography the **randomness** (entropy) plays very important role. In many algorithms, we need **random (i.e. unpredictable) numbers**. If these numbers are not truly random, the algorithms will be compromised.

For example, assume we need a **secret key**, that will protect our crypto assets. This secret key should be **randomly generated** in a way that nobody else should be able to generate or have the same key. If we generate the key from a **true random generator**, the it will be **unpredictable** and the system will be secure. Therefore "secure random" means "**unpredictable random**".

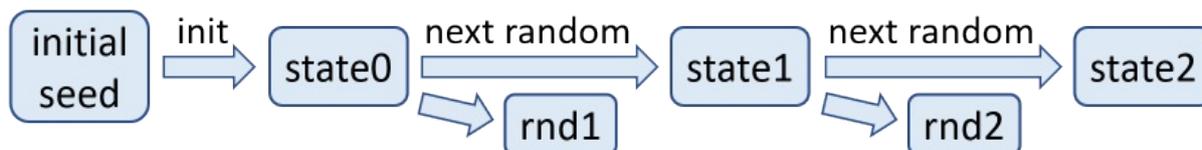
Let's discuss in bigger detail the **random numbers** in computer science and their role in **cryptography**, as well as pseudo-random numbers generators (**PRNG**), secure pseudo-random generators (**CSPRNG**) and some guidelines about how developers should generate and use random numbers in their code.

Random Generators and Cryptography

In computer science **random numbers** usually come from a **pseudo-random number generators** (PRNG), initialized by some unpredictable initial randomness (**entropy**).

Pseudo-Random Number Generators

PRNGs are functions that start from some **initial entropy** (seed) and calculate the next random number by some calculation which is unpredictable without the seed. Such calculations are called **pseudo-random functions**.



Pseudo-random functions usually use an internal **state**. At the start, the state is initialized by an **initial seed**. When the **next random number** is generated, it is calculated from the internal state (using some computation or formula), then the internal **state** of the pseudo-random function is changed (using some computation or formula). When the **next random number** is generated, it is again calculated based on the internal **state** of the function and this state is again changed and so on.

This process in its simplest form can be implemented as follows:

```

init(entropy):
    state = entropy, counter = 0
netNum():
    state = HMAC(state, ++counter)
    return state
  
```

Of course, the **HMAC** function can be changed by some **cryptographic hash** function or another mathematical transformation like the **Mersenne Twister**, but the main idea stays the same: pseudo-random generators have internal **state**, initialized with some **initial randomness** and over the time **change** their internal state and **generate pseudo-random numbers**, based on the current state. Good random number generators should be **fast** and should generate **statistical randomness** (see the [Diehard tests](#)), i.e. all numbers should have the same chance to be generated over the time.

The above idea to generate random pseudo-numbers based on **HMAC(key + counter)**, with some complications, is known as the **HMAC_DRBG algorithm**, described in the security standard [NIST 800-90A](#).

Initial Entropy (Seed)

To be secure, a **PRNG** (which is statistically random) should start by a **truly random initial seed**, which is absolutely **unpredictable**. If the seed is predictable, it will generate predictable sequence of random numbers and the entire random generation process will be **insecure**. That's why having unpredictable randomness at the start (secure seed) is very important.

How to initialize the pseudo-random generator in a secure way? The answer is simple: **collect randomness (entropy)**.

Entropy

In computer science "**entropy**" means **unpredictable randomness**, and is usually measured in bits. For example, if you move your computer's mouse, it will generate some hard-to-predict events, like the start location and the end location of the mouse cursor. If we assume that the mouse has changed its position in the range of [0...255] pixels, the entropy collected from this mouse movement should be about 8 bits (because $2^8 = 255$). Another example: if the user is asked to think of a number in the range [0...1000], this number will hold about 9-10 bits of entropy (because $2^{10} = 1024$). To collect 256 bits of entropy (e.g. to securely generate a 256-bit integer), you will need to take into account a sequence of several such events (like mouse movements and keyboard interactions from the user).

Collecting Entropy

Entropy can be collected from many **hard-to-predict events** in the computer: keyboard clicks, mouse moves, network activity, camera activity, microphone activity and others, combined with the time at which they occur. This collection of initial randomness is usually performed internally by the **operating system** (OS), which provides standard **API** to access it (e.g. reading from the `/dev/random` file in Linux). In desktop system, laptop or mobile phone entropy is easy to collect, while on some limited hardware devices (such as simple microcontrollers) entropy is hard or impossible to be collected.

Application software can **collect entropy explicitly**, by asking the user to move the mouse, type something at the keyboard, say something at the microphone or move in front of the camera for a while. A great example of this is the bitaddress.org wallet app, which combines mouse moves with keyboard events to collect entropy:

bitaddress.org

https://www.bitaddress.org/bitaddress.org-v3.3.0-SHA256-dec17c07685e1870960903d8f58090475b25af946fe95a734f88408cef4aa194.html

 **bitaddress.org**

Open Source JavaScript Client-Side Bitcoin Wallet Generator

51% 51% 51% Brain Wallet

51% 51% Wallet Details

Generating Bitcoin Address...
 MOVE your mouse around to add some extra randomness... 51%
 OR type some random characters into this text box

5542e51749c0c54206d1bc8fac4065ea95cd71fd6ad0881f01c2f23dab6e7a3c51a82
 41f5b4aba2ec2d267da97ffb3a8f65ac0244e0852ca8b89e339e25bcb80c07947021ab9
 174128db831be5526b5f26fdeb64ac142e4700e3c4783038ba7a794e30f3a5b868dd204
 1ab9281f3e495953ca00cf851ef053b3d096b89724d2d3f9f1abc842b2a419350a4e8f9
 f5d2763efec39497a7989efda233a2f135fe176ddc9a098474476ac055a6972d9cce9ac
 e4afa287a13a1f88e43dc49b6ba8a478db9ab83256b5ca3651787b4a6a115e7a2866b7e
 5eb3608d492044632ef1e893cfdbd94126dd748f11162feba484a18ae7ffcda5855c6c
 add672771b6fcdb

Once enough entropy is collected, it is used to initialize the random generator.

Insecure Randomness

Insecure / compromised randomness can compromise cryptography. A good example to learn from is the story of the stolen Bitcoins, due to **broken random generator in Android**: <https://goo.gl/PFE1kr>. That's why developers should care about randomness, when they use cryptography and ensure their **random generators are secure**.

Insecure Randomness - Examples

As example how easy it is to **compromise the random number security in Python** (in its old versions), we shall give this code example:

```
import random
print(random.randrange(1000000, 9999999))
```

The above code is assumed to generate a random number, but this number may be **predictable**. This is because the `random` library in Python (in its old versions) initializes the random generator **seed** by the **current time**. Thus, if you know the current time at the machine generating the random number (obviously you know this roughly), you will be able to predict the random seed and to predict the random numbers generated.

To better illustrate this, look at this more explicit example which generates two random 50-digit integers:

```
import random, time

random.seed(time.time())
r1 = random.randrange(1e49, 1e50-1)

random.seed(time.time())
r2 = random.randrange(1e49, 1e50-1)

print(r1)
print(r2)
```

The above code will print **two equal numbers**, both depending on the current time. It is obvious that the same time in the initial seed causes the same (predictable) pseudo-random numbers to be generated in the output. This is a sample output of the above code:

```
53285353661739398833155340591358345604323255820576
53285353661739398833155340591358345604323255820576
```

If you run this code through a **debugger** or in a slow environment, the produced numbers may be **different**, due to **time change** between the two random generation executions. Typically the Python interpreter at the **interactive console** produces two **different numbers**. To obtain the result, similar to the above, first save the code in a script file (e.g. `insecure-rnd.py`) and then execute the Python script file:

```
$ cat > insecure-rnd.py
import random, time

random.seed(time.time())
r1 = random.randrange(1e49, 1e50-1)

random.seed(time.time())
r2 = random.randrange(1e49, 1e50-1)

print(r1)
print(r2)

$ python insecure-rnd.py
21494400734295049649562103304952829380487090099110
21494400734295049649562103304952829380487090099110
```

Basically, when the initial random seed is initialized with a predictable number like the current time, crackers can **try all possibilities within the range of +/- 5 seconds** and find the exact initial seed and then compromise the security.

Randomness and Cryptography

Remember that **cryptography cannot work without unpredictable randomness!** If your random generator is compromised, it will generate predictable numbers and crackers will be able to decrypt your communication, reveal your private keys, tamper your digital signatures, etc. As a developer, you should always care how random numbers are generated in the cryptographic libraries you use.

Conclusion: Use Secure Random Generator

Always use **cryptographically secure random generator libraries**, like the `java.security.SecureRandom` in Java and the `secrets` library in Python:

```
import secrets
print(secrets.randbelow(int(1e50)))
```

The above code does not depend on the current time and basically generates an **unpredictable random number**, based on the entropy collected by the operating system.

Generating Pseudo Random Numbers - Example in Python

To get a better idea **how pseudo-random numbers are generated** in computer programming, let's play with at the following Python code, which generates 5 pseudo-random numbers in the range [10...20]:

```
import hashlib, time

startSeed = str(time.time()) + '|'
min = 10
max = 20
for i in range(5):
    nextSeed = startSeed + str(i)
    hash = hashlib.sha256(nextSeed.encode('ascii')).digest()
    bigRand = int.from_bytes(hash, 'big')
    rand = min + bigRand % (max - min + 1)
    print(nextSeed, bigRand, '-->', rand)
```

The above code produces time-dependent (predictable) **pseudo-random sequence**:

```
1539884529.7564313|0 80821949188459167822103620715837790870744533466506114260335306835341654
043374 --> 20
1539884529.7564313|1 74025479792630401388590516952955656999942018130178317853592496371994668
720404 --> 12
1539884529.7564313|2 82017697577161203981429946799250236982499988253633196542465974577893633
076425 --> 18
1539884529.7564313|3 10738699706699562929083446539486735923927571219474791024756709089122394
9362198 --> 13
1539884529.7564313|4 83874630241630198317549470506043001102325518306912594861433838548293113
930135 --> 10
```

The **initial pseudo-random seed** is taken from the current time. The first pseudo-random number in the sequence comes from the **SHA-256 hash** of the initial **seed** + the number `0`, the second pseudo-random number comes from the hash of the initial **seed** + the number `1` and so on. To get an output of certain **range [min...max]** the 256-bit **hash** is divided to **(max - min + 1)** and **min** is added to it. The number `i`, together with the value `startSeed` hold the internal **state** of the random generator, which changes for each next random number.

The above pseudo-random generator is based on the **random statistical distribution** of the **SHA-256** function. It is expected that the chance for each possible number to be generated is equal.

Creating a Secure Random Generator

The above random generator is **not secure**, because it is not initialized by an unpredictable source of entropy. **Let's fix this.**

We shall **initialize the initial randomness based on the keyboard events**. The user will be asked to enter something 5 times and the exact precise times of the moments of the user input, together with the data entered from the user will be joined as **initial randomness (seed)**. The collected text entropy can be shortened through SHA-256 hashing (this will reduce it to 256 bits). After the entropy is collected and the start seed is calculated, the same logic like at the previous example will be used to generate 5 random numbers in the range [10...20]. This is a sample Python implementation:

```
import hashlib, time, binascii
```

```

entropy = ''
for i in range(5):
    s = input("Enter something [" + str(i+1) + " of 5]: ")
    entropy = entropy + s + '|' + str(time.time()) + '|'
print("Entropy:", entropy)
startSeed = str(binascii.hexlify(hashlib.sha256(entropy.encode('ascii')).digest()))[2:-1]
print("Start seed = SHA-256(entropy) =", startSeed)

min = 10
max = 20
for i in range(5):
    nextSeed = startSeed + '|' + str(i)
    hash = hashlib.sha256(nextSeed.encode('ascii')).digest()
    bigRand = int.from_bytes(hash, 'big')
    rand = min + bigRand % (max - min + 1)
    print(nextSeed, bigRand, '-->', rand)

```

A **sample output** from the above code may look like this:

```

Enter something [1 of 5]: first
Enter something [2 of 5]: second
Enter something [3 of 5]: random text
Enter something [4 of 5]: dfasfdasfs
Enter something [5 of 5]: last
Entropy: first|1539885709.4494743|second|1539885713.687703|random text|1539885721.5754962|df
asfdasfs|1539885724.40904|last|1539885726.1286101|
Start seed = SHA-256(entropy) = f8a4eaceb16156b1a23f4b6d08e54665ffa4822949b22e01d6de4c5daae9
65e3
f8a4eaceb16156b1a23f4b6d08e54665ffa4822949b22e01d6de4c5daae965e3|0 8448277025956683909793686
6229004786554948913905882724148636325987196754263481 --> 19
f8a4eaceb16156b1a23f4b6d08e54665ffa4822949b22e01d6de4c5daae965e3|1 6700145465903016445734242
1011672033052466168976555224352709830050538321411120 --> 14
f8a4eaceb16156b1a23f4b6d08e54665ffa4822949b22e01d6de4c5daae965e3|2 1037391815072910725723150
34266940107849472122762876847172454548630886082729227 --> 12
f8a4eaceb16156b1a23f4b6d08e54665ffa4822949b22e01d6de4c5daae965e3|3 3011033199204097839903859
902789759740091959530467456042709372597822032778153 --> 16
f8a4eaceb16156b1a23f4b6d08e54665ffa4822949b22e01d6de4c5daae965e3|4 1004660947249247636598436
69256673300207383922129676800217664465341535622195997 --> 16

```

Note that the **collected entropy is very hard to be predicted**. The cracker should guess all the text entered by the user and also guess the exact time for each of the 5 inputs. If the above is repeated 20 instead of 5 times, it will be even harder to predict (the collected entropy will be bigger).

Some cryptographical software use similar techniques like in the above code example when generating keys, password and randomness as general and now you know why: to collect entropy in an unpredictable way.

Secure Random Generators (CSPRNG)

Cryptography secure pseudo-random number generators (**CSPRNG**) are random generators, which guarantee that the random numbers coming from them are **absolutely unpredictable**. Depending on the level of security required, CSPRNG can be implemented as software components or as hardware devices or as combination of both.

For example, in the credit card printing centers the formal security regulations require certified hardware random generators to be used to generate credit card PIN codes, private keys and other data, designed to remain private.

Modern operating systems (OS) **collect entropy** (initial seed) from the **environmental noise**: keyboard clicks, mouse moves, network activity, system I/O interruptions, etc. Sources of randomness from the environment in Linux, for example, include inter-keyboard timings, inter-interrupt timings from some interrupts, and other events which are both non-deterministic and hard to measure for an outside observer.

The collected in the OS randomness is usually accessible from `/dev/random` and `/dev/urandom`.

- Reading from the `/dev/random` file (the limited blocking random generator) **returns entropy** from the kernel's entropy pool (collected noise) and **blocks** when the entropy pool is empty until additional environmental noise is gathered.
- Reading the `/dev/urandom` file (the unlimited non-blocking random generator) returns entropy from the kernel's entropy pool or a pseudo-random data, generated from previously collected environmental noise, which is also unpredictable.

Usually a **CSPRNG** should start from a **truly random seed** from the operating system, from a specialized hardware or from external source. Random numbers after the seed initialization are typically produced by a **pseudo-random computation**, but this does not compromise the security.

Typically modern OS APIs combine the constantly collected **entropy** from the environment with the **internal state** of their built-in pseudo-random algorithm to guarantee maximal **unpredictability** of the generated randomness with high **speed** and **non-blocking** behavior in the same time.

Hardware Random Generators (TRNG)

Hardware random generators, known as **true random number generators (TRNG)**, typically capture physical processes or phenomena, such as the visible spectrum of the light, the thermal noise from the environment, the atmosphere noise, etc. The randomness from the physical environment is collected through specialized sensors, then amplified and processed by the device and finally transmitted to the computer through USB, PCI Express or other standard interface.

Modern **microprocessors** (CPU) provide a built-in hardware random generator, accessible through a special **CPU instruction** `RdRand`, which return a random integer into one of the CPU registers.

Most cryptographic applications today do not require a hardware random generator, because the entropy in the operating system is secure enough for general cryptographic purposes. Using a **TRNG** is needed for systems with higher security requirements, such as banking and finance applications, certification authorities and high volume payment processors.

How as a Developer to Access the CSPRNG?

Typically developers access the cryptographically strong random number generators (**CSPRNG**) for their OS from a **cryptography library** for their language and platform.

- In **Linux** and **macOS**, it is considered that both `/dev/random` and `/dev/urandom` sources of randomness are **secure enough for most cryptographic purposes** and most cryptographic libraries access them internally.

- In **Windows**, random numbers for cryptographic purposes can be securely generated using the `BCryptGenRandom` function from the [Cryptography API: Next Generation \(CNG\)](#) or higher level crypto libraries.
- In **C#** use `System.Security.Cryptography.RandomNumberGenerator.Create()` from .NET Framework or .NET Core.
- In **Python** use `os.urandom()` or the `secrets` library.
- In **Java** use the `java.security.SecureRandom` system class.
- In **JavaScript** use `window.crypto.getRandomValues(Uint8Array)` for client side (in the Web browser) or `crypto.randomBytes()` or external module like `node-sodium` for server-side (in Node.js).

Never use `Math.random()` or similar insecure RNG functions for cryptographic purposes!

Exercises: Implement a Pseudo-Random Generator

Write a code to generate **30 pseudo-random integers** in the range **[1...10]**, starting from certain **entropy**, taken as input, using **HMAC key derivation**.

From the **entropy** generate a **seed** (256-bit binary sequence) using **SHA-256**:

```
seed = SHA256(entropy)
```

Generate the **n**-th random number by the formula:

```
1 + HMAC-SHA256(n, seed) % 10
```

Print the numbers at the output, separated by space.

Sample **input** and corresponding **output**:

Input	Output
hello	8 4 10 5 5 3 5 7 10 6 4 9 2 3 2 8 3 3 10 6 8 10 9 10 1 3 6 4 4 10
random text	10 5 5 9 7 4 2 9 2 1 10 4 8 9 8 1 8 6 5 7 5 4 3 4 6 6 9 8 1 1
fun	6 5 9 2 2 5 1 6 10 10 10 1 8 10 6 9 2 1 5 10 1 4 8 5 6 3 8 4 2 1

Key Exchange / Key Establishment Schemes

In cryptography **key establishment** (**key exchange**, **key negotiation**) is a process or protocol, whereby a **shared secret** becomes available to two parties, for subsequent cryptographic use, typically for encrypted communication. Establishment techniques can be **key agreement** or **key transport** schemes.

- In a **key agreement** scheme both parties contribute to the negotiation of the shared secret. Examples of key agreement schemes are Diffie-Hellman (**DHKE**) and Elliptic-Curve Diffie-Hellman (**ECDH**).
- In a **key transport** scheme only one of the parties contributes to the shared secret and the other party obtains the secret from it. Key transport schemes are typically implemented through **public-key cryptography**, e.g. in the **RSA key exchange** the client encrypts a random session key by its private key and sends it to the server, where it is decrypted using the client's public key.

By design **key exchange** schemes securely exchange cryptographic keys between two parties, in a way that no one else can obtain a copy of the keys. Typically, at the start of an **encrypted conversation** (e.g. during the **TLS handshake** phase), the parties first negotiate about the encryption keys (the shared secret) to be used during the conversation. **Key exchange schemes** are really important topic in the modern cryptography, because keys are exchanged hundreds of times by million devices and servers in Internet.

A **key negotiation (key establishment)** scheme is executed every time when a laptop connects to the Wi-Fi network or a Web browser opens a Web site through the `https://` protocol. The key negotiation can be based on a anonymous key-exchange protocol (like DHKE), a password or pre-shared key (PSK), a digital certificate or a combination of many elements together. Some communication protocols establish a shared secret key once only, while others constantly change the secret key over the time.

Authenticated Key Exchange (AKE) is the exchange of session key in a key exchange protocol which also **authenticates the identities** of the involved parties (e.g. through a password, public key or digital certificate). For example, if you connect to a password-protected WiFi network, an authenticated key agreement protocol is used, in most cases **password-authenticated key agreement (PAKE)**. If you connect to a public WiFi network, **anonymous key agreement** is conducted.

Key Exchange / Key Agreement Algorithms

Many **cryptographic algorithms** exist for key exchange and key establishment. Some use public-key cryptosystems, others use simple key-exchange schemes (like the Diffie–Hellman Key Exchange), some involve server authentication, some involve client authentication, some use passwords, some use digital certificates or other authentication mechanisms.

Examples of key exchange schemes are: **Diffie–Hellman key exchange (DHKE)** and **Elliptic-curve Diffie–Hellman (ECDH)**, **RSA-OAEP** and **RSA-KEM** (RSA key transport), **PSK (pre-shared key)**, **SRP (Secure Remote Password protocol)**, **FHMVQ** (Fully Hashed Menezes–Qu–Vanstone), **ECMQV** (Elliptic-Curve Menezes–Qu–Vanstone) and **CECPQ1** (quantum-safe key agreement).

Let's start from the classical **Diffie–Hellman Key Exchange (DHKE)** scheme, which was one of the first public key protocols.

Diffie–Hellman Key Exchange (DHKE)

Diffie–Hellman Key Exchange (DHKE) is a cryptographic method to **securely exchange cryptographic keys** (key agreement protocol) over a public (insecure) channel in a way that overheard communication does not reveal the keys. The exchanged keys are used later for encrypted communication (e.g. using a symmetric cipher like AES).

DHKE was one of the first **public-key protocols**, which allows two parties to exchange data securely, so that if someone sniffs the communication between the parties, the information exchanged can be revealed.

The Diffie–Hellman (DH) method is **anonymous key agreement scheme**: it allows two parties that have no prior knowledge of each other to jointly establish a **shared secret key over an insecure channel**.

Note that the DHKE method is **resistant to sniffing attacks** (data interception), but it is vulnerable to **man-in-the-middle attacks** (attacker secretly relays and possibly **alters the communication** between two parties).

The **Diffie–Hellman Key Exchange** protocol can be implemented using **discrete logarithms** (the classical **DHKE** algorithm) or using **elliptic-curve cryptography** (the **ECDH** algorithm).

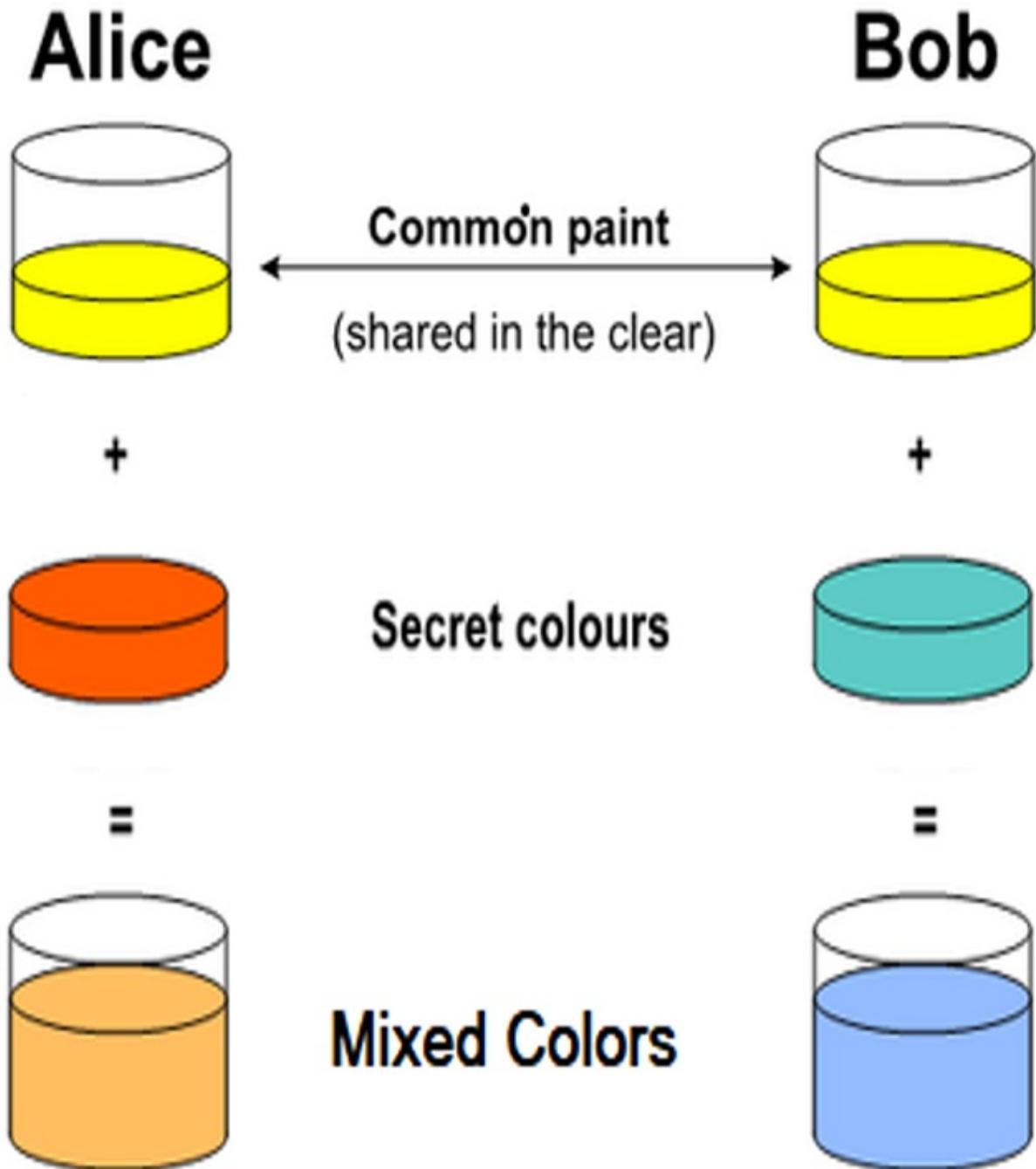
Key Exchange by Mixing Colors

The Diffie–Hellman Key Exchange protocol is very similar to the concept of "**key exchanging by mixing colors**", which has a good visual representation, which simplifies its understanding. This is why we shall first explain how to exchange a secret color by **color mixing**.

The design of color mixing key exchange scheme assumes that if we have two liquids of different colors, we can **easily mix the colors** and obtain a new color, but the reverse operation is almost impossible: **no way to separate the mixed colors** back to their original color components.

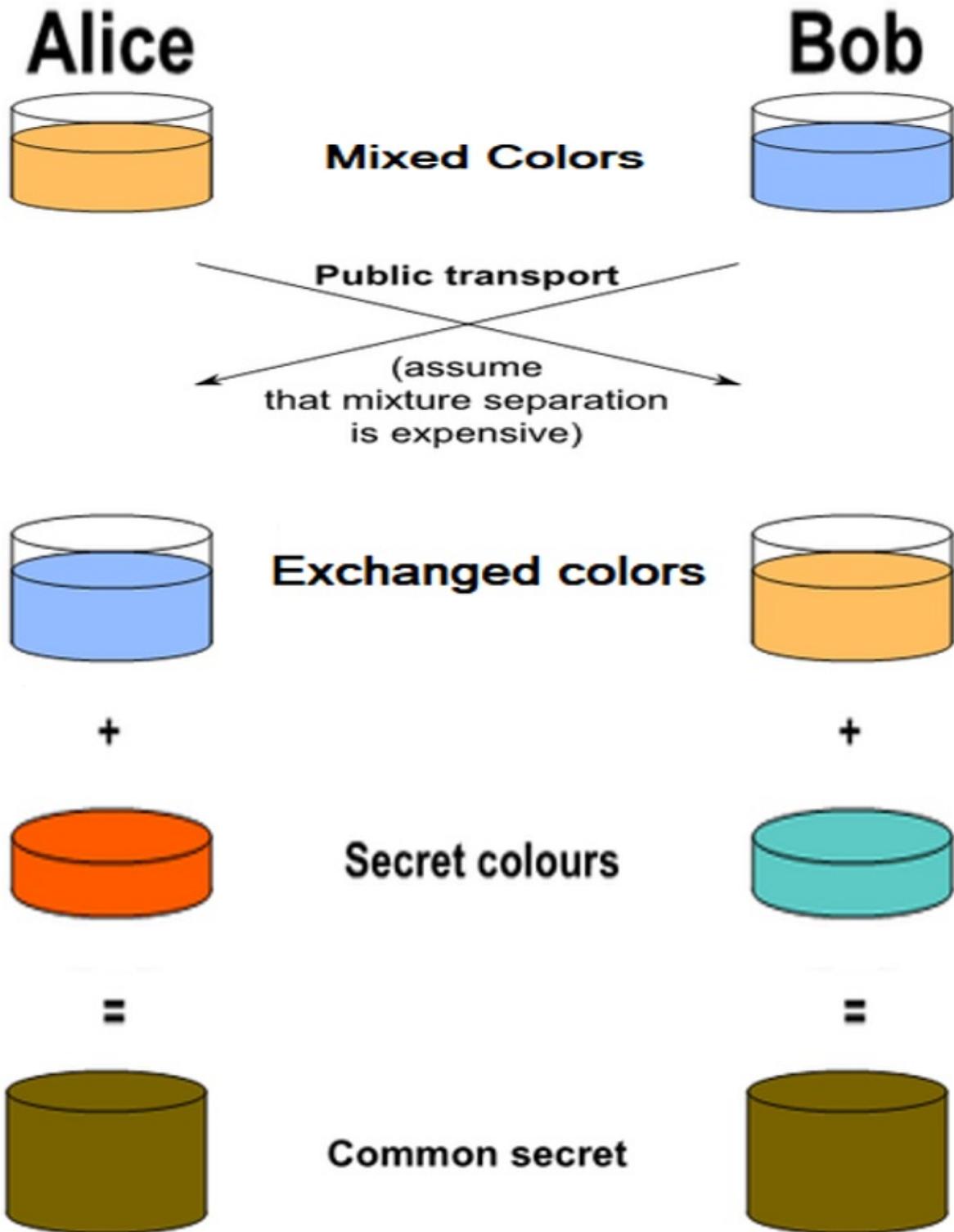
This is the color exchange **scenario**, step by step:

- **Alice** and **Bob**, agree on an arbitrary **starting (shared) color** that does not need to be kept secret (e.g. *yellow*).
- **Alice** and **Bob** separately select a **secret color** that they keep to themselves (e.g. *red* and *sea green*).
- Finally **Alice** and **Bob** **mix** their secret color together with their mutually shared color. The obtained mixed colors are ready for public exchange (in our case *orange* and *light sky blue*).



The next steps in the color exchanging scenario are as follows:

- **Alice** and **Bob** publicly **exchange** their two **mixed colors**.
 - We assume that there is no efficient way to extract (separate) the secret color from the mixed color, so third parties who know the mixed colors cannot reveal the secret colors.
- Finally, **Alice** and **Bob** mix together the color they received from the partner with their own secret color.
 - The result is the **final color mixture** (*yellow-brown*) which is identical to the partner's color mixture.
 - It is the **securely exchanged shared key**.



If a third parties have intercepted the color exchanging process, it would be computationally difficult for them to determine the secret colors.

The **Diffie-Hellman Key Exchange** protocol is based on similar concept, but uses [discrete logarithms](#) and [modular exponentiations](#) instead of color mixing.

The Diffie-Hellman Key Exchange (DHKE) Protocol

Now, let's explain how the **DHKE** protocol works.

The Math behind DHKE

DHKE is based on a simple property of [modular exponentiations](#):

$$(g^a)^b \bmod p = (g^b)^a \bmod p$$

where g , a , b and p are positive integers.

If we have $A = g^a \bmod p$ and $B = g^b \bmod p$, we can calculate $g^{ab} \bmod p$, without revealing a or b (which are called **secret exponents**).

In computing theory, there is no efficient algorithm which can find a secret exponent. If we have m , g and p from the below equation:

$$m = g^s \bmod p$$

there is no efficient (fast) algorithm to find the secret exponent s . This is known as the [Discrete Logarithm Problem \(DLP\)](#).

Discrete Logarithm Problem (DLP)

The **Discrete Logarithm Problem (DLP)** in computer science is defined as follows:

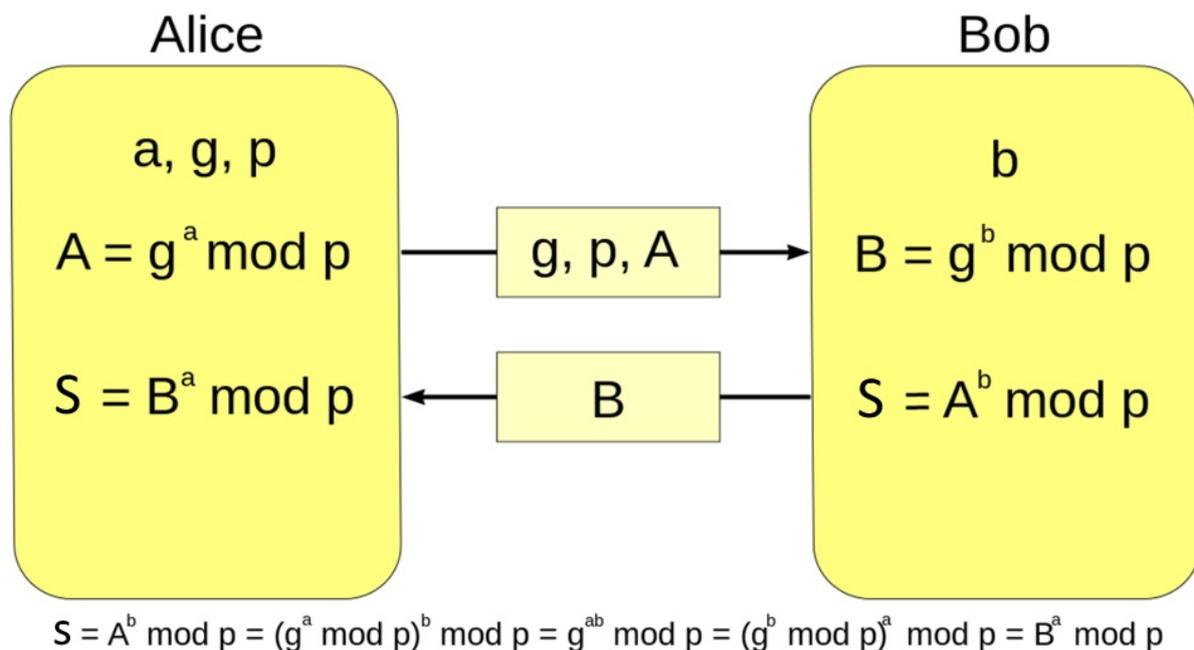
- By given element b and value $a = b^x$ find the exponent x (if it exists)

The exponent x is called [discrete logarithm](#), i.e. $x = \log_b(a)$. The elements a and b can be simple integers modulo p (from the [group \$\mathbb{Z}/p\mathbb{Z}\$](#)) or elements of [finite cyclic multiplicative group \$G\$](#) (modulo p), where p is typically a prime number.

In cryptography, many algorithms rely on the **computational difficulty of the DLP problem** over carefully chosen group, for which **no efficient algorithm exists**.

The DHKE Protocol

Now, after we are familiar with the above mathematical properties of the modular exponentiations, we are ready to explain the **DHKE protocol**. This is how it works:



Let's explain each step of this key-exchange process:

- Alice and Bob agree to use two public integers: **modulus p** and **base g** (where **p** is **prime**, and **g** is a **primitive root modulo p**).
 - For example, let $p = 23$ and $g = 5$.
 - The integers **g** and **p** are public, typically hard-coded constants in the source code.
- Alice chooses a **secret integer a** (e.g. $a = 4$), then calculates and sends to Bob the number $A = g^a \bmod p$.
 - The number **A** is public. It is sent over the public channel and its interception cannot reveal the secret exponent **a**.
 - In our case we have: $A = 5^4 \bmod 23 = 4$.
- Bob chooses a **secret integer b** (e.g. $b = 3$), then calculates and sends to Alice the number $B = g^b \bmod p$.
 - In our case we have: $B = 5^3 \bmod 23 = 10$
- Alice computes $s = B^a \bmod p$
 - In our example: $s = 10^4 \bmod 23 = 18$
- Bob computes $s = A^b \bmod p$
 - In our example: $s = 4^3 \bmod 23 = 18$
- Alice and Bob now share a **secret number s**
 - $s = A^b \bmod p = B^a \bmod p = (g^a)^b \bmod p = (g^b)^a \bmod p = g^{ab} \bmod p = 18$
 - The shared secret key **s** cannot be computed from the publicly available numbers **A** and **B**, because the secret exponents **a** and **b** cannot be efficiently calculated.

In the most common implementation of DHKE (following the [RFC 3526](#)) the base is $g = 2$ and the modulus **p** is a large **prime number** (1536 ... 8192 bits).

Security of the DHKE Protocol

The DHKE protocol is based on the practical difficulty of the [Diffie–Hellman problem](#), which is a variant of the well known in the computer science [DLP \(discrete logarithm problem\)](#), for which no efficient algorithm still exists.

DHKE exchanges a **non-secret sequence of integer numbers** over insecure, public (sniffable) channel (such as signal going through a cable or propagated by waves in the air), but does not reveal the secretly-exchanged shared private key.

Again, be warned that DHKE protocol in its classical form is **vulnerable to man-in-the-middle attacks**, where a hacker can intercept and modify the messages exchanged between the parties.

Finally, note that the integers **g**, **p**, **a** and **b** are typically very big numbers (1024, 2048 or 4096 bits or even bigger) and this makes the [brute-force attacks](#) non-sense.

DHKE - Live Example

As live example, you can play with this online DHKE tool: <http://www.irongeek.com/diffie-hellman.php>

← → ↻

Dirty Diffie-Hellman (Like dirty Santa, but geekier)

Crappy PHP script for a simple Diffie-Hellman key exchange calculator. I guess I could have used Javascript instead of PHP, but I had rounding errors.

Set these two for everyone
g: p:

Alice Bob
a: b:

a = 3
A = $g^a \bmod p = 10^3 \bmod 541 = 459$
b = 6
B = $g^b \bmod p = 10^6 \bmod 541 = 232$
Alice and Bob exchange A and B in view of Carl
key_a = $B^a \bmod p = 232^3 \bmod 541 = 347$
key_b = $A^b \bmod p = 459^6 \bmod 541 = 347$

ECDH - Elliptic Curves-based Diffie-Hellman Key Exchange Protocol

The [Elliptic-Curve Diffie-Hellman \(ECDH\)](#) is an anonymous key agreement protocol that allows two parties, each having an **elliptic-curve public-private key pair**, to establish a shared secret over an insecure channel.

ECDH is a variant of the classical **DHKE** protocol, where the **modular exponentiation** calculations are replaced with **elliptic-curve** calculations for improved security. We shall explain in details the **elliptic-curve cryptography (ECC)** section later.

Diffie–Hellman Key Exchange - Examples in Python

Let's give a simple code example in Python to demonstrate the classical **Diffie–Hellman Key Exchange (DHKE) algorithm**.

First, install the Python package `PyDHE` :

```
pip install pyDHE
```

Next, write the code for the DHKE example:

```
import pyDHE

alice = pyDHE.new()
alicePubKey = alice.getPublicKey()
print("Alice public key:", hex(alicePubKey))

bob = pyDHE.new()
bobPubKey = bob.getPublicKey()
print("Bob public key:", hex(bobPubKey))

print("Now exchange the public keys (e.g. through Internet)")

aliceSharedKey = alice.update(bobPubKey)
print("Alice shared key:", hex(aliceSharedKey))

bobSharedKey = bob.update(alicePubKey)
print("Bob shared key:", hex(bobSharedKey))

print("Equal shared keys:", aliceSharedKey == bobSharedKey)
```

When you run the above code, it will generate and print two **2048-bit public keys** (for Alice and for Bob). Assume that Alice and Bob have exchanged their public keys (e.g. send them to each other through Internet). Once Alice has received Bob's public key, she can **calculate the shared secret** by combining it to her private key. Respectively, once Bob has received Alice's public key, he can **calculate the shared secret** by combining it to his private key. The sample output from the above example shows that the shared secret is always the same number (2048-bit integer):

```
Alice public key: 0xa26c2f1354a8f58abbf78172730595c4de8277962ebe92100793f99ea80f66abe5e75a14
a52e86ce1c086c1ca2e1662b3900510346d848b425d34279ceea92661fb1166b9438589c0b57eb4ebb69e0c3844e
be5ad4c0e316b637d47148d69dc2387c2968c82d198114a6c0f14a605a9e85110d24a9db4f11963b9b13dc788c05
38096cadffd258364c63621f6bb1a3e515d3741af4619e62452a394fab9d84be7cee255fdd7216401cafee6471b4
adbb77e93f878f1bb4df633e0632522b51fe70fc154e7d3e60a69f815a4e2a84506f05b1ccf0e1e873cd7dc51fb
a0b6eac66af1c0a7500f71af405a6c34ffd27a1239180c22fbddf8dc15d30c821c57307d
Bob public key: 0x822660dfff1af80c237402263dda9e0e417fa04547a4e36041a35a152df28b0ac66b059d9e
0034c7cd58b6b7edbc8a20bf1bdc2af6534bd6f2dbccffeb9a4aa9f038461994622f786258beb8f6493594e1559e5
ebf5a92ba60335f668a9cbbf8d6d87460f21d94938ac40cfd78d062571f68aa7e7fbabed4ba582e8e83128867000
4ae64be113a2c7b5b9a472ba4733ea4f29c1b1f30ead3729908d9bb54278a499b2c16cc62d4f330a28cdd302bf65
5f3d724b6d5b0655c9299ada183d8bed4e98c2f0d93339eb3c22c88c9d000de4ea3286b6be5b96e7d7cccb7b8d6a
079264e155c5b25b5aca21ccfed7d21d5dce79845fe5456419504ec9c2a896448572e7
Now exchange the public keys (e.g. through Internet)
Alice shared key: 0x60d96187ae1db8e8acac7795837a2964e4972ebf666eacfa09135371a2de5287db18c1a
30f2af840f04cac42fea21e42369af5ffbeeb235faa42da6bed24cd922ea4637ad146558f2d8b07b19a0084c19f04
1af5456a5826dd836d0c9c4f32ca0a5877da9493af36f66949e76af12e45a20b20c222a37a49b658066bd7b1f79b
```

```
cf81d1083e79c62c43e3ee11f8727e798e310a2683939c06b75ab80c531743d6c03c90007ab8a36af45b3573f4e4
1a2a41c9fdde962493f9ed860597ee527d978e41a413d13198aaac2b27e70aac5be15fd695592350c56b6d74b342
7dcf6888ee11cef4b4d8f5b3acbfbdad1d9b8d7425bc9446e1a6424a929d9136590161cfe
Bob shared key: 0x60d96187ae1db8e8acac7795837a2964e4972ebf666eaecfa09135371a2de5287db18c1a30
f2af840f04cac42fea21e42369af5fffb235faa42da6bed24cd922ea4637ad146558f2d8b07b19a0084c19f041a
f5456a5826dd836d0c9c4f32ca0a5877da9493af36f66949e76af12e45a20b20c222a37a49b658066bd7b1f79bcf
81d1083e79c62c43e3ee11f8727e798e310a2683939c06b75ab80c531743d6c03c90007ab8a36af45b3573f4e41a
2a41c9fdde962493f9ed860597ee527d978e41a413d13198aaac2b27e70aac5be15fd695592350c56b6d74b3427d
cf6888ee11cef4b4d8f5b3acbfbdad1d9b8d7425bc9446e1a6424a929d9136590161cfe
Equal shared keys: True
```

Note that your output will be different due to the **randomness** during the key generation process. The above code uses a 2048-bit public and private keys, as specified in the [RFC 3526 \(group 14\)](#). You can change the DHKE **key size** (from 1536-bits to 8192-bits) by specifying a different RFC 3526 group (e.g. 18 for 8192-bit keys). For example change these two lines:

```
alice = pyDHE.new(group=18)
bob = pyDHE.new(group=18)
```

The above changes will switch to **8192-bit keys** and will significantly slow-down the calculations. The output will look like this:

```
Alice public key: 0x86b2c2bda3982af803084b65d982c08f3462046d154c9ee6fb7c8dcd4a2922b72487c46
e42777ea8bbfad73ca2f340397ddc2b3ddb215891b4811fe014ae176918cc01817e4d9358e6053ed49790e224721
bd14abe7cdeac10be211782d0b1a110c5968654873b1eb3e591c6e5acd0197459aac04da06620d424b327124dee4
958fe49be3f44100591e8560a0e137abb9c47973e4701b3e127a05482934b3b9fdb4117365c476bb6665d867b2dd
58cab72073bcb6632883fba3043b8544a4726fcd013f1676963d612f634675674de1d295e90101d9a0523ae1717e
b2ea11a05e4902af572a9bfff0344c3383e8b85fa7db234927b053d098eda9fda0970c92917caa95fe4dc79376f6
b8f0ee4a9682c88870c36b345049b3ef89bdcfc0f8751b02afa88b22fd5b94d33a49bc6fd262255ac18e27e96675
f311b654f99fa31e060f7e2afbd888099bba072cefaab1e1c40a73845c139e3feaecee76965b71255473b485976e
7f7d87e2ff61a62ddadd5f7f02e9353f5d4f091360418eb7935e83d1e6355c82feff3583725017e8b8b6148af839
e3e7cfd3d549b679d9878544366676509b61b590ce25abbb440207b23fec9c04daef70590c46d720af273d6dafb3
4d2e5b68e24499a4b7ac254ee00712dd0e4ae72299fb103098b8d54c2c28a66e74d52db4853bb695cbff9a09f82
23c55f1e2fd351d419a091cc643b3abc42a477ef6f3eb9d2913e45bcb3ba76771bccfafa85abe3cf37c42bf1baf5
9f122785ec47b51b45c4ba0875e6a80230c5035e45c1cf32e8b7b52ee44e2c3b06330c29f047d5b0983ae8db34d
1ca1a127d1da72d4e0244690c63af4ecf3003152a1cfaa5b4139c361cd3cc54fb7e91bdcac9bf81498da90cf2496
21df90947ccbece28c5befec6bf832d873e18293e7b8e9562596c4b50c61e1aff9b8d13a02df25675c5045aec14d
3e83253d210ae7e6f2c62d622c7bcaf87cd6a4bb63a25d18cc0672fe3488eeffb058231daf17a570382ffb56e490b
1e5003284ca5a8978aa4c09d3e9a11eae379bd66fe86999c10fbe1eb6763d1b6b4f277e8347462e91f127a0f2fe8
a9a16381452e3515608e950587f74e1f85b10ab32e667248f8764d90a8b92eb6bce14cf7306fa56bc7852a0f281
1651665f2121a6253e3e4bfecb12b54c8cd11a54d74346c3b9f2c8c7b71ea60fce8eb1d3badcea909b7f082e4e4a
4ad4e2a501b2fd3a4c7acd48b416706dc3fbda180cec831bacd558fc15cfa3e19347bb5297ac7a4b931b6e19f9b0
dfb0c07696727402c1c5215c0822776147a9a9c7c10bc04d23d6cee974fc37a32fd758cd09bfff9f0b1cdd9e09734
aa0abe0dc9f3a74415c411ce2b07369445d6e4929a0132db60024cf260b17fb3401beb794a5a365a3be92677fb68
f60e091cb5cfc5d767290c4655d6922c2bd194671d5b
Bob public key: 0xbae8a1e6b00ee2df7996323f2d03dd650dcc19e5f2de8c77b4dcd0c611ab50e1bdd41c5d3b
8060a3047616b0a2e55aee0d8211b1d7b18e996e3cd02cf3580247ca42707f73a02266beb077f50b32940c2e09f0
8f1906f177bb1ce3fb6c8516d2f45091aba35a1afac904e694e4c844c3603fd7c8750c15ae349486160d4ce5fce0
c228c8edcd6599f0e680f6928ea7bbec0e9e3787f1476ce02692a22862df0213287dbc0864602c29314f3de68625
940d4dd1ac47d506015dbfee92cda106e5f13360b7d805973b03634726e2e0905bf61736d188cd3d90f667543547
496fa0d9b609320d84d09cde89ff5c1077e811664102f0c69cad41f620fb0ce9651708b8dc3caec2a78029d449e3
0976cbe943d39545a1a3979febbf3e890d2bb389180addcb5af1606baedc4ad2479fe840adae9a64df36de02b019
ff2b639dec3234d844656ef894273e07c272fbd1c650ea853bcd3518118bf78dc9959a83633e43a04245d563c2e
```

```
948be7fa1ffa21e1bb203ae9339e5d9e7a1e0c8ba53cd3c67fc8ba63b1a266299eeb4f66810854b5780e6cb232d0
4350079ffc58914ec8d9b3345321c1d55ab0b87fbc58c01d63d276497cdcfcf79615cac39af387322baeca6dd16
59f4646c487dcae7a84ca77d61fdbd99e81fab7111d6396eb387497a4f914dd45ca67a2e3c026ddd12f4446397af
8fe724228a9aad6e40fe6f788aae5999d60866934f81519b0f709818150b9f61a2a7f1e742423a6da12e05b30a6b
4f64f93d3eacda690ad390ec6358bcfc0de052fdff8c1ede1e3ea5dff104551771d8f3f4556ef8cb64df7b9a66d5
6e5964dc31ab28bdacd46d7a6ea994fbb6fe302b34ffa2cb095f5a4ee9bee18ae2f6ca29f269bb55995804f9925c
10a7e5e5ad3010734b01b192f047c433e04fd836e0ef77b3d6a05503e1692168c664058d5562bec8f53d3839a117
e170add42aa7cd941532cbc6eb6d5f411742cc436ceb679c8f827d538ccc3064dd41b91a77d5f3e68a44b63af94c
95bc93656cfdc7a6e9776db02c9ada793f8a1e16315f39b664564aa676d9cc8a304aa5ab1849b49b905cc18bb798c
2ac8db40a3e0533224dba5b0084ff5855cf840123b29d8738a2df891f32fd883d984b37aed8a3ffbf8c121e5a4e18
7dc8165d3aacf7698b01dc405590c14acd22e0e2a483d71a8d28d671f1b5f3c6ea06121b4c8adc6e261720b3dcd6
6748659cda7ddd8db727dfb58047386b32a3a3bb7288c85d8712a984abb68d7f364d5498c8be4e3e15b87a8b679
4d9fd19e36d416344659a7c427bd1723a5d4574bb6ac9be7181045ec4c1c8d2cd6ca9c7d7187647a6637e684cb57
fd16ea635c18de9845487db591db7bebd3373b5b62f623080a2e007061b0e7a481ffa53e8e6801cfa562feb8b579
4b4a363d3163ebcc2f7e69d8f3334d6564a5dd1020
```

Now exchange the public keys (e.g. through Internet)

```
Alice shared key: 0x964d9b37aa16599c0ce2442f887302555e91d4adb3ae42518a573d149bbdbf31d716e100
f7cab9b2c1aa1e02b6ecf770db0aa2a92b945a87c3c62764c0e44945322d358bd0b5ddc5517afbc88714c1d66bed
e6a209e69f66b23937bf3e2d38357a3365efe2f1624ff653adc76ecc98df66a67da7e93f4ec9ad5487412725f8a
b675f3a3234ac88c8585a6232385b69cfc0a02c520609e7df5fd19814e6d10c7bb0d040bc5b4f8927db9bf006c67
a797080f04aa740b6c1aa93c24c49e3bdf93b8911134fe07768b910b166516e560cbd12f3b20d293f83c6744e3bc
019ba5b46e0fb50e02d7e74da46c3870027c870e4fb81f23a073355069b01feb5b1c445a6231a59f5a67a84c7334
a9d635ddb33644c05a1f5f22d8e47d214d99d797660f3691bc55a616a0d2ef9c8f8845385ae808f9aef35b94e710
c58691be4819a7e1db320fc933dd8eded761bfee1a021b169c734a486f46154cebceff4e47b83099080bbbb21db2
c7042de10be5305901ec5f56056618ae063d1ac7e5351a0774c1ae898f64897cd41e553041f4cd3aa5786c8b998b
eb3ddabf6129df9207b52270a6ed48d612a1909634967d552b3216a2189904ed9f75ffa319a6d911e0a39c0bf45
cd0b0b55a90060ff642b038d12fa97125e46a1473ec50a01cd90a24af5f55f3841514a3fbc304ed9501a03bb28be
d0ab23651d496748170f2f769fd997e6cd638b7267e7ee58e7f6a866f3e2ab94be5ddb675fa741d2784ea9025ab9
9dc639a6af90e32a1634dc9bcba5aef6e0ec6f1138cf9d170fafc6c2aee8f1c8af9a0cb1fd2b3932e7c35d87c3cd
e1034213fefef7495e927109cc0d0c7b4f1ee7588ae85da923c8d761241fcc98300d03a1d41a81fb716896fd2d0d4
c95a416651d64568ea0b164d97fe28d0a4645cebde9038d5c376accbaaaca7f140b4ee960d85b1811bc108a33f91
86521388f8addccf356bb8c03aa430f4193c1ffdd6af9e431847029b83d0379f23fd8a353fa35f1b13f5df53c243
bd4bab1df6c586612145c0743f29a0b6939d6bb4082feae75ef89f04fbacefba862ff8efc216241fffffbf55b91
394a488a20c7bab19eaf2336f1785a70b2cc2f27be5054b10681e829c0958622d9e686c226e8160795190abb87da
710c46e032ca314b3f3699044642c8669c72c06596fbdabe5eb502e8d51fb0f4812750e465761f5266f2ecd1537
396d53c9218aa21aaeda3564241a99305f312d58fb053926e08f06c315d9877454006b6b6d8f4dd75c744d27c302
617d43577f5a03577fc7b70cec0f01f53445baf0fd6f4d90cb75fa5e1da591874c4e486e1c18a3097b0c4d00a8a6
9306551eb8b4138b085942a3f4dfdf3ae62e510eab6ead63473db09c373a7915ccaf8c0441a8c35e1cd21be057a5
a1e8203ca687c1bd89d2fe6b82f83716f3b14b7be192
```

```
Bob shared key: 0x964d9b37aa16599c0ce2442f887302555e91d4adb3ae42518a573d149bbdbf31d716e100f7
cab9b2c1aa1e02b6ecf770db0aa2a92b945a87c3c62764c0e44945322d358bd0b5ddc5517afbc88714c1d66bede6
a209e69f66b23937bf3e2d38357a3365efe2f1624ff653adc76ecc98df66a67da7e93f4ec9ad5487412725f8ab6
75f3a3234ac88c8585a6232385b69cfc0a02c520609e7df5fd19814e6d10c7bb0d040bc5b4f8927db9bf006c67a7
97080f04aa740b6c1aa93c24c49e3bdf93b8911134fe07768b910b166516e560cbd12f3b20d293f83c6744e3bc01
9ba5b46e0fb50e02d7e74da46c3870027c870e4fb81f23a073355069b01feb5b1c445a6231a59f5a67a84c7334a9
d635ddb33644c05a1f5f22d8e47d214d99d797660f3691bc55a616a0d2ef9c8f8845385ae808f9aef35b94e710c5
8691be4819a7e1db320fc933dd8eded761bfee1a021b169c734a486f46154cebceff4e47b83099080bbbb21db2c7
042de10be5305901ec5f56056618ae063d1ac7e5351a0774c1ae898f64897cd41e553041f4cd3aa5786c8b998beb
3ddabf6129df9207b52270a6ed48d612a1909634967d552b3216a2189904ed9f75ffa319a6d911e0a39c0bf45cd
0b0b55a90060ff642b038d12fa97125e46a1473ec50a01cd90a24af5f55f3841514a3fbc304ed9501a03bb28bed0
ab23651d496748170f2f769fd997e6cd638b7267e7ee58e7f6a866f3e2ab94be5ddb675fa741d2784ea9025ab99d
c639a6af90e32a1634dc9bcba5aef6e0ec6f1138cf9d170fafc6c2aee8f1c8af9a0cb1fd2b3932e7c35d87c3cde1
034213fefef7495e927109cc0d0c7b4f1ee7588ae85da923c8d761241fcc98300d03a1d41a81fb716896fd2d0d4c9
```

```
5a416651d64568ea0b164d97fe28d0a4645cebde9038d5c376accbaaaca7f140b4ee960d85b1811bc108a33f9186
521388f8addccf356bb8c03aa430f4193c1ffdd6af9e431847029b83d0379f23fd8a353fa35f1b13f5df53c243bd
4bab1df6c586612145c0743f29a0b6939d6bb4082feae75ef89f04fbacefba862ff8efc216241ffffffbf55b9139
4a488a20c7bab19eaf2336f1785a70b2cc2f27be5054b10681e829c0958622d9e686c226e8160795190abb87da71
0c46e032ca314b3f3699044642c8669c72c06596fbda1be5eb502e8d51fb0f4812750e465761f5266f2ecd153739
6d53c9218aa21aaeda3564241a99305f312d58fb053926e08f06c315d9877454006b6b6d8f4dd75c744d27c30261
7d43577f5a03577fc7b70cecf01f53445bafd0fd6f4d90cb75fa5e1da591874c4e486e1c18a3097b0c4d00a8a693
06551eb8b4138b085942a3f4dfdf3ae62e510eab6ead63473db09c373a7915ccaf8c0441a8c35e1cd21be057a5a1
e8203ca687c1bd89d2fe6b82f83716f3b14b7be192
Equal shared keys: True
```

Enjoy to modify and experiment with the above code to learn the DHKE protocol.

Exercises: DHKE Key Exchange

...

TODO

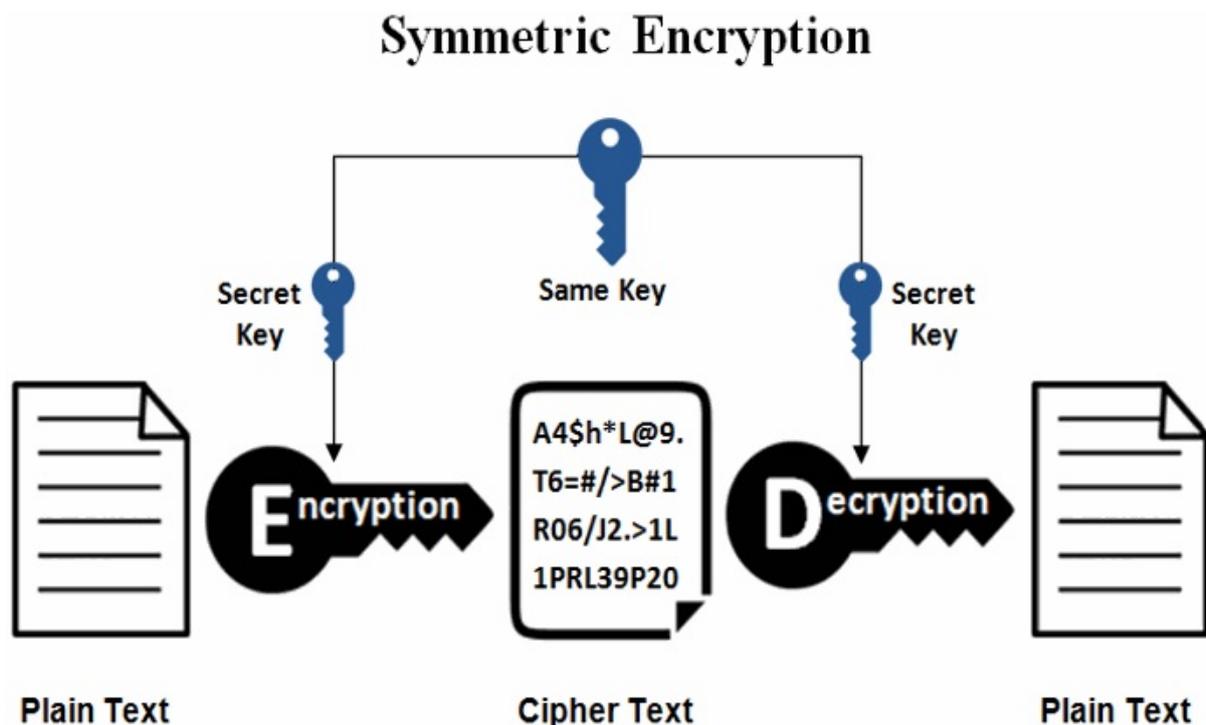
...

Symmetric and Asymmetric Encryption - Overview

In cryptography two major types of encryption schemes are widely used: **symmetric encryption** (where a single **secret key** is used to encrypt and decrypt data) and **asymmetric encryption** (where a public key cryptosystem is used and encryption and decryption is done using a **pair of public and corresponding private key**). Let's explain these fundamental crypto-concepts in details.

Symmetric Encryption - Concepts and Algorithms

Symmetric encryption schemes use **the same key** (or password) to **encrypt** data and **decrypt** the encrypted data back to its original form:



Secret Keys

The **secret key** used to **cipher** (encrypt) and **decipher** (decrypt) data is typically of size 128, 192 or 256 bits and is sometimes referred as "**shared key**", because both sending and receiving parties should know it.

Most applications use a **password-to-key-derivation** scheme to extract a **secret key** from certain **password**, because users tend to remember passwords easier than binary data. Additionally, message authentication is often incorporated along with the encryption to provide integrity and authenticity (this encryption approach is known as "**authenticated encryption**").

How does a **private key** look like? Let's start from a simple example of **256-bit secret key**, encoded as **hex string**:

```
02c324648931b89e3e8a0fc42c96e8e3be2e42812986573a40d46563bceaf75110
```

In many blockchain systems keys are encoded as **base58** or **base64** for shorter string representation.

For example, the above key looks like this in **base58**:

```
pbPRqYDxnKZfs8j4KKiqYmx6nzipAjTJf1oCD1WKgy99
```

The same key looks like this in **base64**:

```
AsMkZIkxuJ4+ig/ELJbo474uQoEph1c6QNR1Y7zq91EQ
```

In **decimal** system, the above key is the following integer number:

```
319849484316084980661994213716306415989897600164422912728298459349458028548368
```

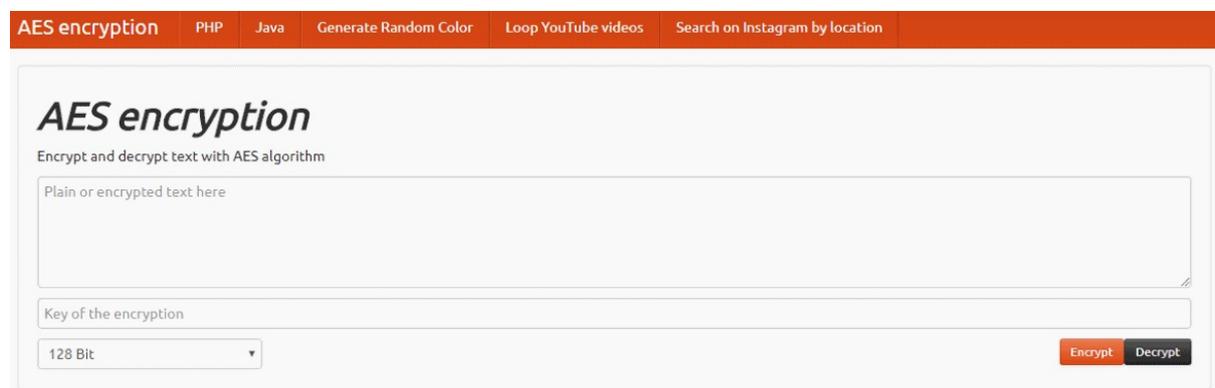
Modern Symmetric Encryption Algorithms

Widely used in modern cryptography symmetric encryption algorithms (ciphers) are: **AES** (AES-128, AES-192, AES-256), **ChaCha20**, **Twofish** and **IDEA**.

We shall give more details and code examples using these algorithms a bit later.

Symmetric Encryption - Online Demo

In order to better understand the idea behind the symmetric encryption, you can play with some **online symmetric encryption tool** to encrypt and decrypt a sample message by sample secret key (or password). You can play a bit with this site: <https://aesencryption.net>.



The screenshot shows the 'AES encryption' web application. At the top, there is a navigation bar with links for 'AES encryption', 'PHP', 'Java', 'Generate Random Color', 'Loop YouTube videos', and 'Search on Instagram by location'. The main content area has the title 'AES encryption' and the subtitle 'Encrypt and decrypt text with AES algorithm'. Below this, there is a large text input field labeled 'Plain or encrypted text here'. Underneath that is a smaller input field for 'Key of the encryption'. To the left of the key field is a dropdown menu currently set to '128 Bit'. To the right of the key field are two buttons: 'Encrypt' (in orange) and 'Decrypt' (in dark grey).

It demonstrates how we can encrypt and decrypt messages, using the **AES cipher** (with some default settings) and certain password-to-key-derivation function. In the above example if we encrypt "**secret msg**" by the password "**p@ss**", we will get the **base64-encoded** binary data "**jVJwOBmH+qMqHdg22KwMyg==**" as output. After decryption with the same secret key we get back the original text "**secret msg**".

Note that the above encrypted text is dependent to many algorithm parameters and settings, so if you encrypt the same at another "**AES live example**" web site, the result most likely will be different.

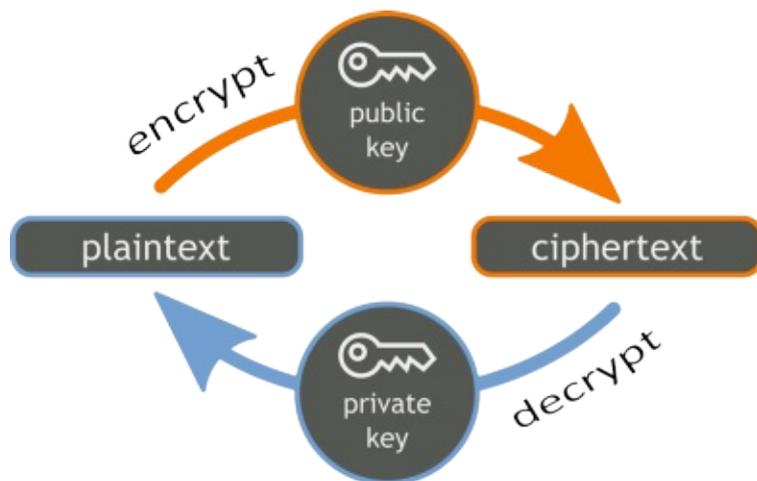
Public Key Cryptography - Concepts

Before introducing the **asymmetric key encryption** schemes and algorithms, we should first understand the concept of **public key cryptography**.

The **public key cryptography** uses a different key to encrypt and decrypt data (or to sign and verify messages). Keys always come as **public + private key pairs**.

Asymmetric Encryption / Decryption

Data **encrypted by a public key** is **decrypted** by the corresponding **private key**:



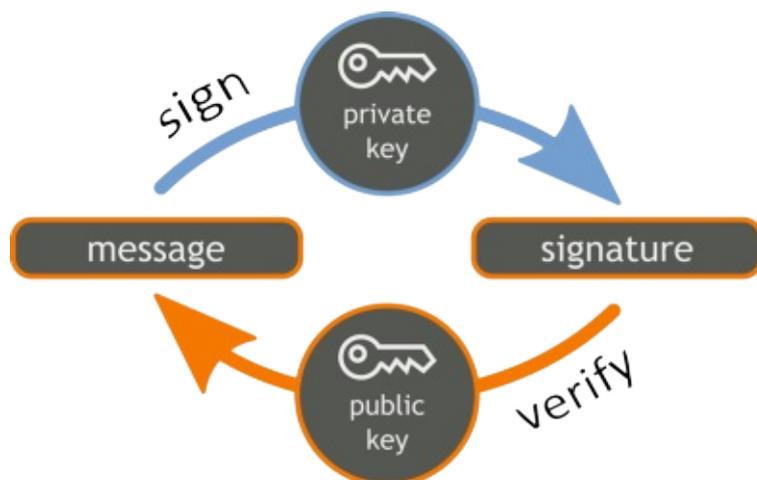
The encrypted data, obtained as result of encryption is called "**ciphertext**". The ciphertext is a binary sequence, unreadable by humans and typically cannot be restored without the decryption key.

Public key encryption can work also in the opposite scenario: **encrypt data by a private key and decrypt it by the public key**. Thus someone can prove that he is owner of certain private key, while revealing only its corresponding public key.

Typically, public-key cryptosystems can **encrypt messages of limited length** only and are slower than symmetric ciphers. For encrypting longer messages (e.g. PDF documents) usually a **public-key encryption scheme** is used, which combines **symmetric** and **asymmetric encryption** like this: a random symmetric key sk is generated, the message is symmetrically encrypted by sk , then sk is asymmetrically encrypted using the recipient's public key. For decryption, first the sk key is asymmetrically decrypted using the recipient's private key, then the ciphertext is decrypted symmetrically using sk . This process can be simplified using a **key encapsulation mechanism (KEM)** which encapsulates a random symmetric key into an asymmetrically encrypted message.

Signatures: Asymmetric Signing / Verification

In the context of **digital signatures**, a message **signed by a private key** (digital signature) is later **verified** by the corresponding **public key**.



Digital signatures will be explained in more details later, but in short: a message can be **signed** by certain **private key** and the obtained **signature** can be later **verified** by the corresponding **public key**. A **signed message** cannot be altered after signing. A message signature proves that certain message (e.g. blockchain transaction) is created by the owner of certain public key. Digital signatures provide message **authentication**, message **integrity** and **non-repudiation**.

In blockchain, transactions are typically signed by the owner of certain blockchain address (which corresponds to certain public key and has corresponding private key). So a **signed blockchain transaction holds a proof of authorship**: it is guaranteed mathematically that the signature is created by the holder of certain blockchain address and the transaction was not modified after the signing. This works perfectly for the scenario of **digital payments** and digital signing of documents and contracts.

Key Pairs

The **public key cryptography** uses a **pair of keys: public key + private key**. These keys are mathematically connected and are used together as **pair**.

In some public key cryptosystems (like the Elliptic-Curve Cryptography - **ECC**), the public key can be calculated from the private key. In other cryptosystems (like **RSA**), the public key and the private key are generated together but cannot be directly calculated from each other.

Usually, a **public / private key pair** is randomly generated in a secure environment (e.g. in a hardware wallet) and the public key is revealed, while the private key is securely stored in a crypto-wallet and is protected by a password or by multi-factor authentication.

Example of 256-bit public key and its corresponding 256-bit private key (both based on the classical elliptic curves cryptosystem, used in Bitcoin and Ethereum):

```
privKey: 648fc1fa828c7f185d825c04a5b21af9e473b867eeee1acea4dbab938433e158
pubKey: 02c324648931b89e3e8a0fc42c96e8e3be2e42812986573a40d46563bceaf75110
```

Private Keys

Message **encryption** and **signing** is done by a **private key**. The private keys are always kept **secret** by their owner, just like passwords. In the blockchain systems the private keys usually stay in specific software or hardware apps or devices called "**crypto wallets**", which store securely a set of private keys.

Example of 256-bit private key:

```
648fc1fa828c7f185d825c04a5b21af9e473b867eeee1acea4dbab938433e158
```

Public Keys

Message **decryption** and **signature verification** is done by the **public key**. Public keys are by design public information (not a secret). It is mathematically infeasible to calculate the private key from its corresponding public key.

In blockchain systems public keys are usually published as parts of the blockchain transactions to help identify who has signed each transaction.

Example of 256-bit public key:

```
02c324648931b89e3e8a0fc42c96e8e3be2e42812986573a40d46563bceaf75110
```

In most blockchain systems the **blockchain address** is derived from the public key, so if you have someone's public key, you are assumed to have his blockchain address as well.

A certain **public key** can be connected to certain **person** or **organization** or is used anonymously. You can never know who is the owner of the private key, corresponding to certain public key, unless you have additional proof, e.g. a [digital certificate](#).

Public Key Cryptosystems

Public key cryptosystems provide mathematical framework and algorithms to generate public + private key pairs, to **sign, verify, encrypt** and **decrypt** messages and **exchange keys**, in a cryptographically secure way.

Well-known public-key cryptosystems are: [RSA](#), [ECC](#) and [ElGamal](#).

The RSA Cryptosystem

The [RSA public-key cryptosystem](#) is based on the math of **modular exponentiations** (numbers raised to a power by modulus) and some additional assumptions, together with the computational difficulty of the [integer factorization problem](#). We shall discuss it later in details, along with examples.

The ECC Cryptosystem

The [elliptic-curve cryptography \(ECC\) public-key cryptosystem](#) is based on the math of the on the algebraic structure of the **elliptic curves** over finite fields and the difficulty of the [elliptic curve discrete logarithm problem \(ECDLP\)](#). The ECC usually comes together with the [ECDSA algorithm](#) (elliptic-curve digital signature algorithm). We shall discuss ECC and ECDSA in details, along with examples.

ECC is Recommended in the General Case

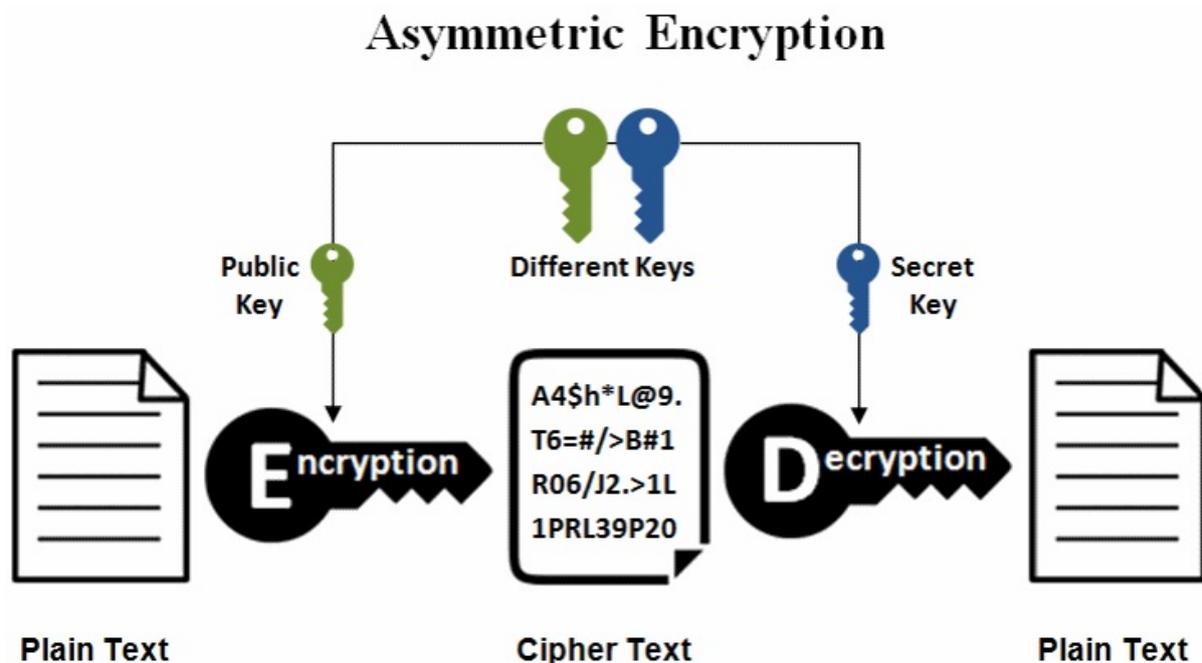
ECC uses smaller keys, ciphertexts and signatures than RSA and is recommended for most applications. It is mathematically proven that a **3072-bit RSA key** has similar cryptographic strength to a **256-bit ECC key**. Key generation in ECC is significantly faster than with RSA.

Due to the above reasons most blockchain networks (like Bitcoin and Ethereum) use elliptic-curve-based cryptography (ECC) to secure the transactions.

Note that both **RSA** and **ECC** cryptosystems are **not quantum-safe**, which means that if someone has enough powerful quantum computer, he will be able to derive the private key from given public key in just few seconds.

Asymmetric Encryption - Concepts and Algorithms

Asymmetric encryption schemes use a pair of cryptographically related **public** and **private keys** to **encrypt** the data (by the public key) and **decrypt** the encrypted data back to its original forms (by the private key).



Asymmetric cryptography deals with **signing** messages, **verifying** signatures, **encrypting** and **decrypting** messages using a public / private key cryptosystem.

Asymmetric Encryption

Asymmetric encryption works for **small messages** only (limited from the public / private key length). To encrypt larger messages **key encapsulation mechanisms** or other techniques can be used, which encrypt asymmetrically a random secret key, then use it to symmetrically encrypt the larger messages. In practice, modern **asymmetric encryption schemes** involve using a symmetric encryption algorithm together with a public-key cryptosystem, key encapsulation and message authentication.

Popular **asymmetric encryption schemes** are: **RSA-OAEP** (based on RSA and OAEP padding), **RSAES-PKCS1-v1_5** (based on RSA and PKCS#1 v1.5 padding), **DLIES** (based on discrete logarithms and symmetric encryption) and **ECIES** (based on elliptic curve cryptography and symmetric encryption).

Asymmetric Crypto Algorithms

Popular asymmetric algorithms are: **RSA**, **ECC**, **EIGamal**, **Diffie-Hellman**, **DSA**, **ECDSA** and **EdDSA**.

We shall discuss the **RSA** and **ECC** cryptosystems in details later. Now, it is important to learn that symmetric and asymmetric cryptosystems work differently and are used in different scenarios.

Asymmetric Encryption - Online Demo

In order to better understand the idea behind the **asymmetric encryption**, you can play with some online public key encryption tool to encrypt / decrypt a sample message by sample RSA private / public key. You can play a bit with this site: <http://travistidwell.com/jsencrypt/demo/>.

In the above online demo you can **generate RSA public / private key pairs** and **encrypt / decrypt** text messages. Note that the message size is limited by the key length, so you can't encrypt long text. Internally, the above site uses the **RSAES-PKCS1-v1_5** public key encryption scheme as specified in [RFC3447](https://tools.ietf.org/html/rfc3447).

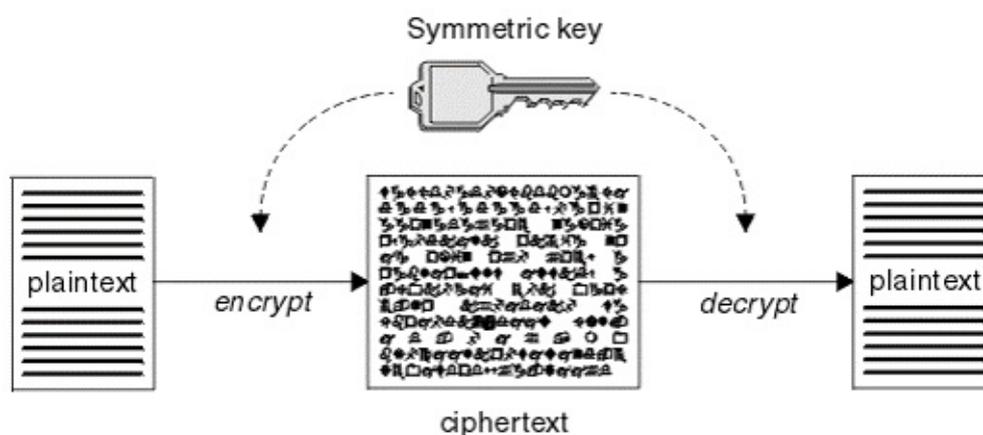
Symmetric Key Ciphers - Overview

Symmetric key ciphers (like **AES**, **ChaCha20**, **RC6**, **Twofish**, **CAST** and many others) use the same key (or password) to **encrypt** and **decrypt** data. They are often used in combination with other algorithms into a **symmetric encryption schemes** (like **ChaCha20-Poly1305** and **AES-128-GCM** and **AES-256-CTR-HMAC-SHA256**), often with password to **key derivation** algorithms (like **Scrypt** and **Argon2**). Symmetric key ciphers are **quantum-resistant**, which means that powerful quantum computers will not be able to break their security (when big enough key lengths are used).

Symmetric Encryption / Decryption

Symmetric encryption and decryption uses a **secret key** or passphrase (to derive the key from it). The **secret key** used to encrypt and decrypt the data is usually 128 bits or 256 bits and is called "**encryption key**". Sometimes it is given as hex or base64-encoded integer number or is derived through a **password-to-key derivation scheme**.

When the input data is encrypted, it is transformed to **encrypted ciphertext** and when the ciphertext is decrypted, it is transformed back to the original input data.



Symmetric Encryption Uses a Set of Algorithms

It is important to know as a concept that symmetric-key encryption algorithms usually do not work standalone. They work together with other related crypto algorithms, into a **symmetric encryption scheme / symmetric encryption construction**.

In most encryption schemes an **encryption** is combined with password to **key derivation** algorithm and **message authentication** scheme (see [authenticated encryption](#)). Typically a symmetric encryption procedure uses a sequence of steps, involving different crypto algorithms:

- **Password-to-key derivation** algorithm (like **Scrypt** or **Argon2**): to allow using a password instead of a key and to make password cracking hard and slow to be performed.
- **Block to stream cipher transformation** algorithm (block cipher mode like **CBC** or **CTR**) + **message padding** algorithm like **PKCS7** (in some modes): to allow encrypting data of arbitrary size using a block cipher algorithm (like **AES**).
- **Block cipher algorithm** (like **AES**): to securely encrypt data blocks of fixed length using a secret key.
- **Message authentication** algorithm (like **HMAC**): to check whether after decryption the obtained result matches the original message before the encryption.

Later in this section we shall give **more details and examples** about how to configure and use symmetric block ciphers (like **AES**) along with the all above described algorithms to securely encrypt and decrypt messages of arbitrary size.

Block Ciphers, Stream Ciphers, Block Modes and Padding

In cryptography **block ciphers** (like AES) are designed to **encrypt a block** of data of **fixed size** (e.g. 128 bits). The size of the input block is usually the same as the size of the encrypted output block, while the key length may be different.

Stream ciphers are more flexible: they are designed to encrypt **data of arbitrary size** (e.g. a PDF document), that may sometimes come as a **stream** (sequence of bytes or frames, e.g. video streaming).

Most of the popular symmetric key encryption algorithms are **block ciphers**, but cryptographers have proposed several schemes to **transform a block cipher into a stream cipher** and encrypt data of arbitrary size. These schemes are known as "**block cipher modes of operation**".

Block Cipher Modes (CBC, CTR, GCM, ...)

The main idea behind the **block cipher modes** (like CBC, CFB, OFB, CTR, EAX, CCM and GCM) is to repeatedly apply a cipher's single-block encryption / decryption to securely encrypt / decrypt amounts of data larger than a block.

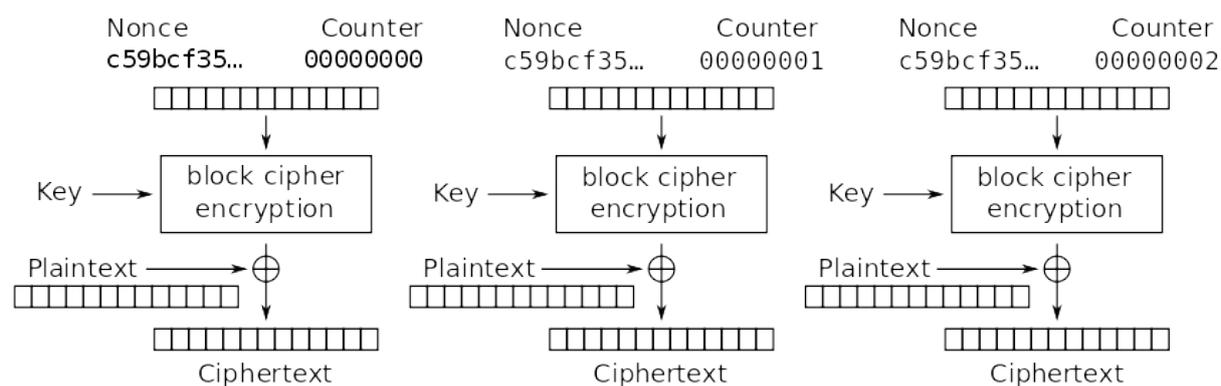
Some block modes (like CBC) require the input to be **split into blocks** and the final block to be **padded** to the block size using a **padding algorithm** (e.g. add a special padding character). Other block modes (like CTR, CFB, OFB, CCM, EAX and GCM) **do not require padding** at all, because they perform XOR between portions of the plaintext and the internal cipher's state at each step.

Basically, **encrypting a large input data** works like this: the encryption algorithm **state** is initialized, then the first portion of data (e.g. a block or part of block) is **encrypted**, then **the encryption state is transformed** (using the **encryption key** and other parameters), then the next portion is **encrypted**, then the encryption state is **transformed** again and the next portion is then **encrypted** and so on, until all the input data is processed. The **decryption** works in a very similar way.

This is what developers should know about the "**block cipher modes of operation**" in order to use them correctly.

- Commonly used **secure block modes** are **CBC** (Cipher Block Chaining), **CTR** (Counter) and **GCM** (Galois/Counter Mode), which require a random initialization vector (**IV**) at the start.
- The "**Counter (CTR)**" block mode is **good choice** in the most cases because of strong security, arbitrary input data length (without padding) and parallel processing capabilities. It does not provide authentication and integrity, just encryption.
- The **GCM** (Galois/Counter Mode) block mode takes all the advantages of the **CTR** mode and adds message **authentication** (produces a cryptographic message authentication tag). **GCM** is fast and efficient way to implement **authenticated encryption** in symmetric ciphers and it is highly **recommended** in the general case.
- In **CBC** mode many **padding algorithms** can be used to make the last block the same length after splitting the input data into blocks. Most applications use the **PKCS7 padding scheme** or **ANSI X.923**.
- Well-known **insecure block mode** is **ECB** (Electronic Codebook), which encrypts equal input blocks as equal output blocks (does not provide **cryptographic diffusion**). **Don't use it!** It may compromise the entire encryption.

The diagram below illustrates how portions (blocks) of the plaintext are encrypted one after another in the **CTR block mode** of operation using a block cipher:



Counter (CTR) mode encryption

For each block in CTR mode a new unpredictable **keystream block** is generated based on the **initial vector** (IV, sometimes called "nonce") + the **current counter** (01, 02, 03, ...) + the secret **encryption key** and the **input block** is merged by **XOR** with the current keystream block to produce the **output block**. In the CTR mode the final portion of the input data can be shorter than the cipher block size, so padding is not needed. The input data (before encryption) and the output data (after encryption) have the **same length**.

The **CTR** and **GCM** encryption modes have many advantages: they are **secure** (no significant flaws are currently known), can encrypt data of **arbitrary length** without padding, can encrypt and decrypt the blocks **in parallel** (in multi-core CPUs) and provide **random (unordered) access** to the encrypted blocks, so they are suitable for encrypting crypto-wallets, documents and streaming video (where users can seek by time). **GCM** provides also message authentication and is **the recommended choice** for cipher block mode in the general case.

Note that the **GCM**, **CTR** and other block modes **reveal the length of the original message**. The length of the plaintext message is the same as the ciphertext length. If you want to avoid revealing the original plaintext length, you can add some random bytes to the plaintext before the encryption and remove them after decryption (this will be some kind of padding).

Authenticated Encryption

In cryptography the concept of "**authenticated encryption**" (**AE**) refers to a scheme to **encrypt data** and simultaneously calculate an **authentication code** (authentication tag / MAC), used to provide message **authenticity** and **integrity**. If authenticated encryption scheme is used, at the moment of decryption it will be known if the **decryption is successful** (i.e. whether the decryption key / password was correct and whether the encrypted data was not tampered). Authenticated encryption (AE) is related to the similar concept **authenticated encryption with associated data** (**AEAD**), which a more secure variant of AE. **AEAD** binds associated data (AD) to the ciphertext and to the **context** where it's supposed to appear, so that attempts to "cut-and-paste" a valid ciphertext into a different context can be detected and rejected.

Some encryption schemes (like **ChaCha20-Poly1305** and **AES-GCM**) provide **integrated authenticated encryption** (AEAD), while others (like **AES-CBC** and **AES-CTR**) need authentication to be added additionally.

Popular Symmetric Encryption Algorithms

Symmetric key encryption algorithms (like **AES**) are designed by mathematicians and cryptographers with the idea, that it should be **infeasible to decrypt the ciphertext** without having the encryption key. This is true for the modern **secure symmetric encryption algorithms** (like AES and ChaCha20) and may be disputable or false for others, which are considered **insecure symmetric encryption algorithms** (like DES and RC4).

Some popular symmetric encryption algorithms are: **AES**, **ChaCha20**, **CAST**, **Twofish**, **IDEA**, **Serpent**, **RC5**, **RC6**, **Camellia** and **ARIA**. All these algorithms are considered **secure** (when configured and used correctly).

AES (Rijndael)

AES (Advanced Encryption Standard, also known as **Rijndael**) is the most popular and widely used **symmetric encryption algorithm** in the modern IT industry. This is because AES is proven to be **highly secure**, fast and well standardised and very well supported on virtually all platforms. AES is 128-bit **block cipher** and uses 128, 192 or 256-bit secret keys. It is usually used in a **block mode** like **AES-CTR** or **AES-GCM** to process streaming data. In the most block modes AES require also a random 128-bit initial vector (nonce).

Rijndael was the winner in the [AES competition organized by NIST](#) (1997-2000) and it was announced officially under the name "**AES**" (the next official symmetric block cipher after DES). In 2001 AES was adopted as official recommendation by the **US government** and no significant weakness or attack was found since this moment. The Rijndael (AES) algorithm is **free for any use**: public or private, commercial or non-commercial.

Salsa20 / ChaCha20

Salsa20, along with its improved variants **ChaCha** (**ChaCha8**, **ChaCha12**, **ChaCha20**) and **XSalsa20**, are a family of modern, fast, **symmetric stream ciphers**, designed by the distinguished cryptographer [Daniel Bernstein](#). The **Salsa20** cipher was one of the finalists in the [eSTREAM contest](#) for designing of new symmetric stream ciphers (2004-2008) and was widely adopted afterwards, together with the related **BLAKE** hash function. **Salsa20** and its variants are **royalty-free**, not patented.

The **Salsa20** cipher takes as input a **128-bit** or **256-bit symmetric secret key** + randomly generated **64-bit nonce** (initial vector) and a stream of data of unlimited length and produces as output an encrypted stream of data with the same length as the input stream.

Other Popular Symmetric Ciphers

Other **modern secure symmetric ciphers**, used more rarely than EAS and ChaCha20, but still popular in the software developer and information security communities, are the following:

- **Camellia** - secure symmetric key block cipher (block size: 128 bits; key sizes: 128, 192 and 256 bits), patented, but free for non-commercial use
- **RC5** - secure symmetric-key block cipher (key size: 128 to 2040 bits; block size: 32, 64 or 128 bits; rounds: 1 ... 255), insecure with short keys (56-bit key successfully brute-forced), was patented, now royalty-free
- **RC6** - secure symmetric-key block cipher, similar to RC5, but more complicated (key size: 128 to 2040 bits; block size: 32, 64 or 128 bits; rounds: 1 ... 255), was patented until 2017, now royalty-free
- **Serpent** - secure symmetric-key block cipher (key size: 128, 192 or 256 bits), public domain, not patented
- **Twofish** - secure symmetric-key block cipher (key sizes: 128, 192 or 256 bits), royalty-free, not patented
- **IDEA** - secure symmetric-key block cipher (key size: 128 bits), was patented until 2012, now royalty-free
- **CAST** (**CAST-128 / CAST5**, **CAST-256 / CAST6**) - family of secure symmetric-key block ciphers (key sizes: 40 ... 256 bits), royalty-free basis for commercial and non-commercial use
- **ARIA** - secure symmetric-key block cipher, similar to AES (key size: 128, 192 or 256 bits), official standard in South Korea, free for public use

- **SM4** - secure symmetric-key block cipher, similar to AES (key size: 128 bits), official standard in China, free for public use

Insecure Symmetric Algorithms

Some other symmetric encryption algorithms were popular in the past, but are now considered **insecure (broken algorithms)** or having **disputable security** and are **not recommended** to be used any more:

- **DES** - 56-bit key size, practically broken, can be brute-forced
- **3DES** (Triple DES) - 64-bit cipher, considered broken
- **RC2** - 64-bit cipher, considered broken
- **RC4** - stream cipher, broken, practical attacks demonstrated
- **Blowfish** - old 64-bit cipher, broken, practical attacks demonstrated
- **GOST** - Russian 64-bit block cipher, disputable security, considered risky

Symmetric Encryption Schemes / Constructions

In addition to the above mentioned symmetric key ciphers, cryptographers have proposed many **symmetric encryption schemes** (constructions), like the most popular authenticated encryption (AEAD) schemes:

- **ChaCha20-Poly1305**
 - The **ChaCha20** stream cipher with integrated **Poly1305** authenticator (integrated authenticated AEAD encryption)
 - Requires a **256-bit key** and random **96-bit nonce**
 - Extremely **high performance**
 - Implemented by the most modern crypto-libraries
- **AES-256-GCM**
 - **AES-GCM** is the AES (Rijndael) block cipher in GCM block mode (integrated authenticated AEAD encryption), behaves like a stream cipher
 - Required **256-bit key** and random **128-bit nonce** (initial vector)
 - Implemented by the most modern crypto libraries

Most applications today should **prefer some of the above encryption schemes** for symmetric encryption, instead of constructing their own encryption scheme. The above schemes are highly-secure, proven, well tested and come out-of-the box from the crypto libraries.

Note that **ChaCha20-Poly1305** is high-performance cipher (**3 times faster** than AES-128-GCM on mobile devices), so it is **recommended** to be used instead of AES-GCM.

The AES Symmetric-Key Cipher - Concepts

The **Advanced Encryption Standard (AES)** cipher, also known as "**Rijndael**" is a popular, secure, widely used **symmetric key block cipher** algorithm, used officially as recommended encryption technology standard in the United States. **AES** operates using **block size of 128 bits** and symmetric **keys of length 128, 160, 192, 224 and 256 bits**.

AES is Secure and Very Popular Symmetric Encryption Algorithm

The **AES** symmetric encryption algorithm is considered **highly secure** (when configured correctly) and no significant practical attacks are known for AES in its history.

AES is used internally by the most Internet Web sites today for serving `https://` content as part of the **TLS** (Transport Layer Security) and **SSL** (Secure Sockets Layer) standards for secure host to host communication on the Web.

Due to its wide use in the Internet secure communication, modern CPU hardware implements **AES instructions** at the microprocessor level to speed-up the AES encryption and decryption.

AES Algorithm Parameters

The **AES** algorithm can operate with different **key lengths**, but the block size is always 128 bits. For most application **128-bit AES** encryption (AES-128) is enough, but for higher encryption level, it is recommended to use **AES-256** (256-bit key length).

Like any other block ciphers, **AES** can use one of several **modes of operation** (CBC, ECB, CTR, ...) to allow encryption of data of arbitrary length. The recommended mode for the general case and for encrypting blockchain wallets is "**CTR**".

Most modes of operation require an **initial vector (IV)**. When using a counter mode (CTR), i.e. **AES-128-CTR** (128-bit) or **AES-256-CTR** (256-bit) for example, first a non-secret **random salt (IV)** should be generated and saved along with the encrypted ciphertext output. The **size of the IV** is always the same as the size of the block, i.e. **128 bits** (16 bytes).

The **AES encryption**, combined with **CTR** block mode and random **IV** causes the encryption algorithm to produce different encrypted ciphertext each time, when the same input data is encrypted. This ensures that nobody can construct a dictionary to reverse back the encrypted ciphertext.

AES encryption in **CBC mode** uses a **padding algorithm** (like **PKCS7** or **ANSI X.923**) to help splitting the input data into blocks of fixed block-size (e.g. 128 bits) before passing the blocks to the AES-CBC algorithm. Most developers use the **CTR mode** of operation for AES, so they don't need padding.

Without using a block mode, the **ciphertext**, generated by the **AES** algorithm is exactly **128 bits** (16 bytes), just like the block size. The **input data** is also exactly 128 bits.

The **ciphertext**, generated by the **AES-CTR** algorithm (AES in CTR cipher block mode) has the same size like the size of the input data. No padding is required.

The **ciphertext**, generated by the **AES-CBC** algorithm (AES in CBC ciphertext mode), has size of **128 bits** (16 bytes) or multiple of 128 bits. The input data should be **padded** before encryption and **unpadded** after decryption.

The AES algorithm often is used along with a **password-to-key derivation** function, e.g. `Scrypt(passwd) -> key` or `PBKDF2(passwd) -> key`.

Integrated Message Authentication Code (MAC)

The **AES** algorithm may use **MAC (message authentication code)** to check the password validity, e.g. `HMAC(text, key)`.

The **MAC code** is typically **integrated** (see the concept of **integrated encryption**) in the algorithm's output. It is calculated from the input message, together with the encryption key. From the calculated MAC, it is impossible to reveal the input message or the key, so the MAC itself is not a secret. Some block cipher modes (like **AES-GCM**) integrate message authentication in the obtained ciphertext as part of their work, so you don't need to add MAC explicitly.

Typically **MAC** is calculated and used like this:

- Before the encryption, the **MAC** is calculated as: `mac = HMAC-SHA256(input_msg, key)`.
- The **input data is encrypted** and the **ciphertext** is stored along with the **random salt (IV)** and the **MAC**.
- After decryption, the **MAC** is calculated again and is **compared** with the **MAC** stored along with the encrypted message.
 - If the MAC is the same, **the decryption is successful**: correct ciphertext + decryption key + algorithms settings (IV, block mode, padding algorithm).
 - If the MAC is different, **the decryption is not successful**: incorrect key / password or broken ciphertext, incorrect MAC or different algorithms settings (IV, block mode, padding, etc.)

The MAC can be calculated and verified using several [approaches to integrated encryption](#): **Encrypt-then-MAC**, **Encrypt-and-MAC**, **MAC-then-Encrypt**.

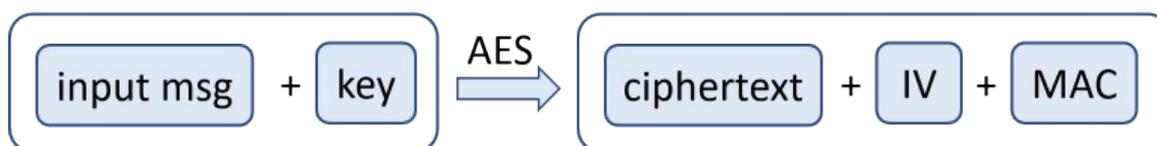
The AES Encryption Process

The entire **AES encryption** process (password-based authenticated encryption) looks like this:

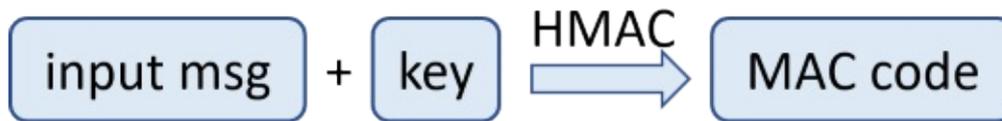
1. **Algorithm parameters** are selected (e.g. AES, 128-bit, CTR mode + Scrypt + Scrypt parameters + MAC algorithm). These parameters can be **hard-coded** in the AES algorithm implementation source code or can be specified as input for the AES encrypt and decrypt. Always use the same parameters for encryption and decryption.
2. The encryption **key** is derived from the encryption **password** using a key-derivation function (**KDF**), e.g. **Scrypt** (with certain parameters):



3. The **AES** encryption scheme takes as **input** the **input msg** + the encryption **key**. It produces as output the **ciphertext** + the randomly generated **IV** (128-bit salt) + the **MAC** code:



- In case of **authenticated encryption** (e.g. AES-GCM), the MAC is already calculated automatically during the AES encryption process.
- If the encryption scheme is **not authenticated encryption** (e.g. AER-CTR), the MAC code is not calculated automatically by the AES encryption process and should be calculated additionally. The **MAC code** can be calculated from the **input msg**, using the encryption **key** (or some transformation of it) and some **MAC function** (like HMAC-SHA-256):



- The **ciphertext** is calculated through the **AES encryption algorithm**. It first **generates a random salt (IV)** and uses it to transform the **input msg** using the **encryption key**, through the AES cipher encryption logic:

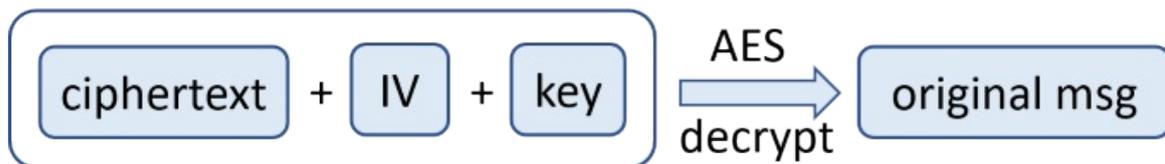


4. Finally, the encrypted output is generated. It holds the **ciphertext + IV + MAC**. Optionally, it holds also the algorithm settings.

The AES Decryption Process

The opposite **AES decryption** process (password-based authenticated decryption) looks like this:

1. Initialize the same **AES algorithm parameters** for the decryption process, exactly like the ones used during the encryption.
2. Use the decryption **key** and the **IV** from the encrypted message to decrypt the **ciphertext** using the **AES** algorithm decryption logic. The output is the **original message** (the **input msg**, which was previously passed through AES encryption):



- In case of **authenticated encryption** (like AES-GCM), the integrated **MAC code is verified** during the decryption process.
 - In case of **unauthenticated encryption** (like AES-CTR), the MAC code should be calculated and verified additionally, as it is described in the next few step.
3. Calculate **HMAC** of the original message (obtained during the decryption):



4. Compare the **encryption MAC** (the MAC of the input message, before the encryption) with the **decryption MAC** (the MAC of the original message, recovered by the decryption):



- If the **MAC codes are the same**, the decryption was correct and the original message is obtained.
- If the **MAC codes are different**, the decryption was failed and the original message is not the obtained one. This may happen due to many reasons, most likely "*wrong password*". Other reasons: incorrect ciphertext, incorrect IV, incorrect algorithm settings, incorrect KDF function or KDF parameters, etc.

Now it is time to illustrate the above described concepts through working **source code** to AES encrypt / decrypt an **input msg** by given **password**.

AES Encryption / Decryption - Examples in Python

Let's illustrate the **AES encryption** and **AES decryption** concepts through working **source code** in Python. The example below will illustrate a **password-based AES encryption**, without message authentication (**unauthenticated encryption**).

Install Python Libraries pyaes and pbkdf2

First, install the Python library `pyaes` that implements the **AES** symmetric key encryption algorithm:

```
pip install pyaes
```

Next, install the Python library `pbkdf2` that implements the **PBKDF2** password-to-key derivation algorithm:

```
pip install pbkdf2
```

Now, let's play with a simple AES encrypt / decrypt example.

Password to Key Derivation

First start by **key derivation**: from password to 256-bit encryption key.

```
import pyaes, pbkdf2, binascii, os, secrets

# Derive a 256-bit AES encryption key from the password
password = "s3cr3t*c0d3"
passwordSalt = os.urandom(16)
key = pbkdf2.PBKDF2(password, passwordSalt).read(32)
print('AES encryption key:', binascii.hexlify(key))
```

The above code **derives a 256-bit key** using the **PBKDF2** key derivation algorithm from the password `s3cr3t*c0d3`. It uses a random password derivation **salt** (128-bit). This salt should be stored in the output, together with the ciphertext, because without it the decryption key cannot be derived again and the decryption will be impossible.

The output from the above code may look like this:

```
AES encryption key: b'7625e224dc0f0ec91ad28c1ee67b1eb96d1a5459533c5c950f44aae1e32f2da3'
```

The derived **key** consists of **64 hex digits** (32 bytes), which represents a **256-bit** integer number. It will be different if you run the above code several times, because a random salt is used every time. If you use the same salt, the same key will be derived.

AES Encryption (CTR Block Mode)

Next, generate a **random 256-bit initial vector (IV)** for the AES CTR block mode and perform the **AES-256-CTR encryption**:

```
# Encrypt the plaintext with the given key:
# ciphertext = AES-256-CTR-Encrypt(plaintext, key, iv)
iv = secrets.randbits(256)
plaintext = "Text for encryption"
aes = pyaes.AESModeOfOperationCTR(key, pyaes.Counter(iv))
ciphertext = aes.encrypt(plaintext)
```

```
print('Encrypted:', binascii.hexlify(ciphertext))
```

The output from the above code may look like this:

```
Encrypted: b'53022cf12c5959ddf3e733128930dd3d52e3ea'
```

The **ciphertext** consists of **38 hex digits** (19 bytes, 152 bits). This is the size of the input data, the message `Text for encryption`.

Note that after AES-CTR encryption the **initial vector (IV)** should be stored along with the ciphertext, because without it, the decryption will be impossible. The **IV** should be randomly generated for each AES encryption (not hard-coded) for higher security.

Note also that if you encrypt the same **plaintext** with the same encryption **key** several times, the output will be **different** every time, due to the randomness in the **IV**. This is intended behavior and it increases the security, e.g. resistance to dictionary attacks.

AES Decryption (CTR Block Mode)

Now let's see how to **decrypt a ciphertext** using the AES-CTR-256 algorithm. The input consists of **ciphertext** + encryption **key** + the **IV** for the CTR counter. The output is the original **plaintext**. The code is pretty simple:

```
# Decrypt the ciphertext with the given key:
# plaintext = AES-256-CTR-Decrypt(ciphertext, key, iv)
aes = pyaes.AESModeOfOperationCTR(key, pyaes.Counter(iv))
decrypted = aes.decrypt(ciphertext)
print('Decrypted:', decrypted)
```

The output of the above should be like this:

```
Decrypted: b'Text for encryption'
```

Note that the `aes` object should be **initialized again**, because the CTR cipher block mode algorithm keeps an internal **state** that changes over the time.

Note also that the above code **cannot detect wrong key**, wrong **ciphertext** or wrong **IV**. If you use an incorrect key to decrypt the ciphertext, you will get a wrong unreadable text. This is clearly visible by the code below:

```
key = os.urandom(32) # random decryption key
aes = pyaes.AESModeOfOperationCTR(key, pyaes.Counter(iv))
print('Wrongly decrypted:', aes.decrypt(ciphertext))
```

The output of the above incorrect decryption attempt might be like this:

```
Wrongly decrypted: b'\xe6!\n\x9a\xa9\x15\x12\xd9\xcb\x9c5\x86\xcc\xe1\x1d\x1a\x8b1w'
```

Now it is your time to **play with the above code example**. Try to to encrypt and decrypt different messages, to change the input message, the key size, to hard-code the IV, the key and other parameters, switch to CBC mode, and see how the results change. Enjoy learning by playing.

Ethereum UTC / JSON Wallet Encryption (AES + Scrypt + MAC)

To illustrate the application of the **AES cipher in action**, we shall look into one **real-world example**: the standard **encrypted wallet file format** for the **Ethereum** blockchain. We shall see how **AES-128-CTR** cipher is combined with **Scrypt** and **MAC** to securely implement authenticated symmetric key encryption by text-based password.

Ethereum UTC / JSON Wallets

In public blockchain networks (like Bitcoin and Ethereum) the private keys of the blockchain asset holders are stored in special keystores, called **crypto wallets**. Typically these crypto-wallets are files on the local hard disk, encrypted by a password.

In the **Ethereum blockchain** crypto wallets are internally stored in a special **encrypted** format known as "**UTC / JSON Wallet (Keystore File)**" or "**Web3 Secret Storage Definition**". This is the **wallet file format**, used in **geth** and **Parity** (the leading protocol implementations for Ethereum), in **MyEtherWallet** (popular online client-side Ethereum wallet), in **MetaMask** (widely used in-browser Ethereum wallet), in the **ethers.js** and **Nethereum** libraries and in many other Ethereum-related technologies and tools.

The Ethereum **UTC / JSON keystores** keep the **encrypted private key** (or wallet seed words) as **JSON text document**, specifying the encrypted data, encryption algorithms and their parameters.

UTC / JSON Keystore - Example

Let's look into a **sample UTC / JSON keystore file**, which holds a password-protected 256-bit private key.

```
{
  "version": 3,
  "id": "07a9f767-93c5-4842-9afd-b3b083659f04",
  "address": "aef8cad64d29fcc4ed07629b9e896ebc3160a8d0",
  "Crypto": {
    "ciphertext": "99d0e66c67941a08690e48222a58843ef2481e110969325db7ff5284cd3d3093",
    "cipherparams": { "iv": "7d7fabf8dee2e77f0d7e3ff3b965fc23" },
    "cipher": "aes-128-ctr",
    "kdf": "scrypt",
    "kdfparams": {
      "dklen": 32,
      "salt": "85ad073989d461c72358ccea3551f7ecb8e672503cb05c2ee80cfb6b922f4d4",
      "n": 8192,
      "r": 8,
      "p": 1
    },
    "mac": "06dcf1cc4bffe1616fafe94a2a7087fd79df444756bb17c93af588c3ab02a913"
  }
}
```

The above JSON document is a classical example of **authenticated symmetric encryption**.

What Is Inside the UTC / JSON File?

Typically a UTC / JSON keystore holds the following data:

- **Key-derivation function (KDF)** used to transform the text-based wallet encryption **password** into an AES symmetric **key**, used to encrypt the wallet contents. Usually the KDF function is **"scrypt"**.

- The **KDF parameters** - the parameters used in the KDF function to derive the password (e.g. iterations count, salt, etc.)
- The **ciphertext** - the encrypted wallet content (typically holds an encrypted 256-bit private key).
- Symmetric **cipher algorithm** + its **parameters**, e.g. **AES-128-CTR** + initial vector (**IV**).
- **MAC** - message authentication code used (MAC) to check the message integrity after it is decrypted (to know whether the wallet decryption password was correct or not).
 - Ethereum calculates the MAC by calculating **keccak-256** hash of the concatenations of the second-leftmost 16 bytes of the derived key together with the full **ciphertext**.
- Additional **metadata**: wallet format **version**, wallet unique **id** (uuid) and the blockchain **address**, controlled by this wallet.

By default the key-derivation function is scrypt and uses **weak script parameters** (n=8192 cost factor, r=8 block size, p=1 parallelization), so it is recommended to use long and complex passwords to avoid brute-force wallet decryption attacks.

MyEtherWallet: Play with UTC / JSON Keystore Files

To learn better the file format behind the Ethereum UTC / JSON keystore files, play with **MyEtherWallet**.

Follow the steps below to create a new **random Ethereum crypto wallet** and view its encrypted JSON content:

- Open the **MyEtherWallet** web site: <https://myetherwallet.com>.
- Choose a **password** and create a **new wallet**.
- Download the **Keystore File** (UTC / JSON).
- See **what's inside** the downloaded file.
- Try to **make some changes**, try to decrypt it with wrong password and other changes.
- Enjoy **learning by playing**.

Exercise: Symmetric Key Encryption / Decryption (using AES + Scrypt + HMAC)

In this exercise we shall **encrypt** and **decrypt** a text message using a symmetric cipher **AES-CBC-256**, combined with **Scrypt** password-to-key derivation and **HMAC** message authentication code. In fact we shall implement a **password-based symmetric authenticated encryption scheme**.

Symmetric Encryption (AES + Scrypt + HMAC)

Write a program to **encrypt** a text **message** using given **password**. Use the following steps:

- Derive a **512-bit key** from the **password** using **Scrypt** ($n=16384$, $r=16$, $p=1$) with random **salt** (128 bits).
 - Split the derived key into two **256-bit** sub-keys: **encryption key** and **HMAC key**.
- **Pad** the input message using the **PKCS7** algorithm to length, which is multiple of **16 bytes** (128 bits).
- **Encrypt** the padded message using **AES-256-CBC** using the **encryption key**. The obtained result is the **ciphertext**. Its length should be a multiple of 16 bytes (128 bits), which is the block size in the AES cipher.
 - Use a randomly generated 128-bit initial vector (**IV**).
- Calculate message authentication code (**MAC**) using **HMAC-SHA256(hmac_key, ciphertext)**.

Input: **message + password** (space separated).

Output: **JSON** document (see the example below), holding the following assets:

- The Scrypt randomly-generated **salt** (in hex format).
- The randomly-generated **iv** (in hex format), used for the AES cipher.
- The encrypted message **ciphertext** (in hex format) from the **AES** cipher.
- The message authentication code - **mac** (in hex format).

Write your code in programming language of choice.

Input	Output
p@sSw0rd ~123 Secret Msg	<code>{"salt": "9757a3a22a9937ca0e0f2b5f2a4a11b4", "iv": "2ce8c035d50f7a6ee6509c14fe11725a", "ciphertext": "bb435d8ad048c240b50f0e4a191605d9", "mac": "02cf870ad1f7453c339dac06edb648c455f5e8abbf6f2716cbc2d164b644200"}</code>
stupid!pas s longer-me ssage-for-e ncryption	<code>{"salt": "b243f0ac10ef358ff0d37f1e30ef19c2", "iv": "fdeff97e89705289d99751f079e2a308", "ciphertext": "ea76bc60799c5824627a8c1276b48ab70e24011b6654f8ffb019a4f6876485af", "mac": "34085e1a47ae53e154b7466336efee386c2f1ed61a0105183ef016af794da58f"}</code>

Note that the above input will be different in your case, because of the randomly generated **salt** and **iv**.

Symmetric Decryption (AES + Scrypt + HMAC)

Write a program to **decrypt** an **encrypted message** (coming as input) using given **password**.

- Derive a **512-bit key** from the **password** using **Scrypt** ($n=16384$, $r=16$, $p=1$) with the **salt** (from the JSON).
 - Split the derived key into two 256-bit sub-keys: **encryption key** and **HMAC key**.
- Calculate message authentication code (**MAC**) using **HMAC-SHA256(hmac_key, ciphertext)**.
 - **Compare** the MAC with the MAC in the JSON document.
 - Same MAC means "correct password / successful decryption".

- o Different MAC means "wrong password / incorrect input data".
- **Decrypt** the **ciphertext** from the input using **AES-256-CBC** using the **encryption key** and the **IV** from the JSON.
- **Unpad** the decrypted message using the **PKCS7** algorithm from length, which is multiple of **16 bytes** (128 bits) to its original length (usually smaller).

Input: password + JSON (space separated). The JSON is in exactly the same format, like in the output from the previous exercise (it holds **salt**, **iv**, **ciphertext** and **mac**, all as **hex** numbers)

Output: Decrypted: + the original **decrypted message** or the text `Decryption failed!` in case of wrong password or other problem.

Write your code in programming language of choice.

Input	Output
<code>p@sSw0rd~123 {"salt": "9757a3a22a9937ca0e0f2b5f2a4a11b4", "iv": "2ce8c035d50f7a6ee6509c14fe11725a", "ciphertext": "bb435d8ad048c240b50f0e4a191605d9", "mac": "02cf870ad1f7453c339dac06edbd648c455f5e8abff6f2716cbc2d164b644200"}</code>	Decrypted: SecretMsg
<code>wrong!pass {"salt": "9757a3a22a9937ca0e0f2b5f2a4a11b4", "iv": "2ce8c035d50f7a6ee6509c14fe11725a", "ciphertext": "bb435d8ad048c240b50f0e4a191605d9", "mac": "02cf870ad1f7453c339dac06edbd648c455f5e8abff6f2716cbc2d164b644200"}</code>	Decryption failed!

ChaCha20-Poly1305

[TODO]

The AEAD construction **ChaCha20-Poly1305** combines the **ChaCha20** stream cipher paired with the **Poly1305** authenticator...

Chacha20-Poly1305 - Example in Python

<https://github.com/ph4r05/py-chacha20poly1305>

[TODO]

Exercise: Symmetric Key Encryption / Decryption (ChaCha20-Poly1305)

[TODO]

Asymmetric Key Ciphers and Public-Key Cryptography - Overview

Asymmetric key cryptosystems / public-key cryptosystems (like [RSA](#), [elliptic curve cryptography \(ECC\)](#), [Diffie-Hellman](#), [ElGamal](#), [McEliece](#), [NTRU](#) and others) use a pair of mathematically linked keys: **public key** (encryption key) and **private key** (decryption key).

The asymmetric key cryptosystems provide **key-pair generation** (private + public key), **encryption algorithms** (asymmetric key ciphers and encryption schemes like [RSA-OAEP](#) and [ECIES](#)), **digital signature algorithms** (like [DSA](#), [ECDSA](#) and [EdDSA](#)) and **key exchange algorithms** (like [DHKE](#) and [ECDH](#)).

A message **encrypted** by the **public key** is later **decrypted** by the **private key**. A message **signed** by the **private key** is later **verified** by the **public key**. The **public key** is typically shared with everyone, while the **private key** is kept secret. Calculating the private key from its corresponding public key is by design computationally infeasible.

Public-Key Cryptosystems

Well-known **public-key cryptosystems** are: [RSA](#), [ECC](#), [ElGamal](#), [DHKE](#), [ECDH](#), [DSA](#), [ECDSA](#), [EdDSA](#), [Schnorr signatures](#). Different public key cryptosystems may provide one or more of the following capabilities:

- **Key-pair generation**: generate random pairs of private key + corresponding public key.
- **Encryption / decryption**: encrypt data by public key and decrypt data by private key (often using a hybrid encryption scheme).
- **Digital signatures** (message authentication): sign messages by private key and verify signatures by public key.
- **Key-exchange algorithms**: securely exchange cryptographic key between two parties over insecure channel.

The most important and most used public-key cryptosystems are **RSA** and **ECC**. Elliptic curve cryptography (ECC) is the recommended and most preferable modern public-key cryptosystem, especially with the modern highly optimized and secure curves (like Curve25519 and Curve448), because of smaller keys, shorter signatures and better performance.

The **RSA public-key cryptosystem** is based on the mathematical concept of **modular exponentiation** (numbers raised to a power by modulus), along with some mathematical constructions and the **integer factorization problem** (which is considered to be computationally infeasible for large enough keys).

The **elliptic-curve cryptography (ECC) cryptosystem** is based on the math of the on the algebraic structure of the **elliptic curves** over finite fields and the **elliptic curve discrete logarithm problem (ECDLP)**, which is considered to be computationally infeasible for large keys. **ECC** comes together with the **ECDSA** algorithm (elliptic-curve digital signature algorithm). ECC uses smaller keys and signatures than RSA and is preferred in most modern apps. We shall discuss ECC and ECDSA later in details, along with examples.

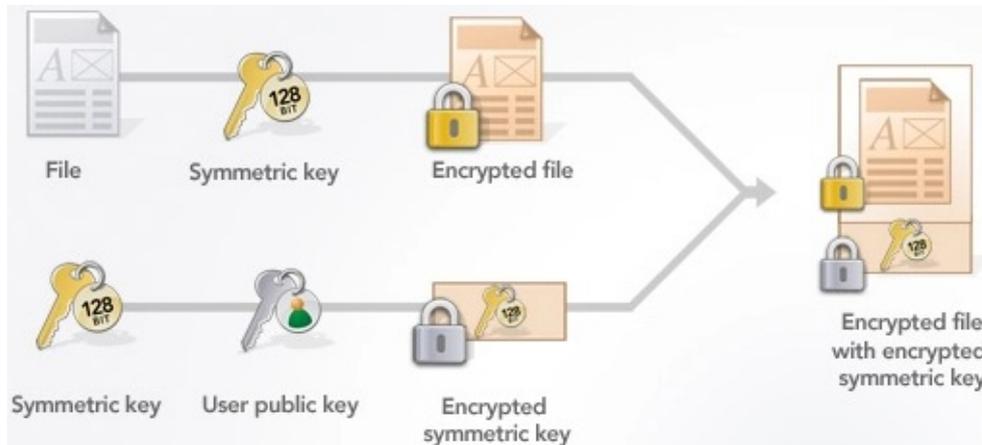
Most **public-key cryptosystems** (like RSA, ECC, DSA, ECDSA and EdDSA) are **quantum-breakable** (quantum-unsafe), which means that (at least on theory) a powerful enough quantum computer will be able to break their security and compute the private key from given public key in seconds.

Asymmetric Encryption Schemes

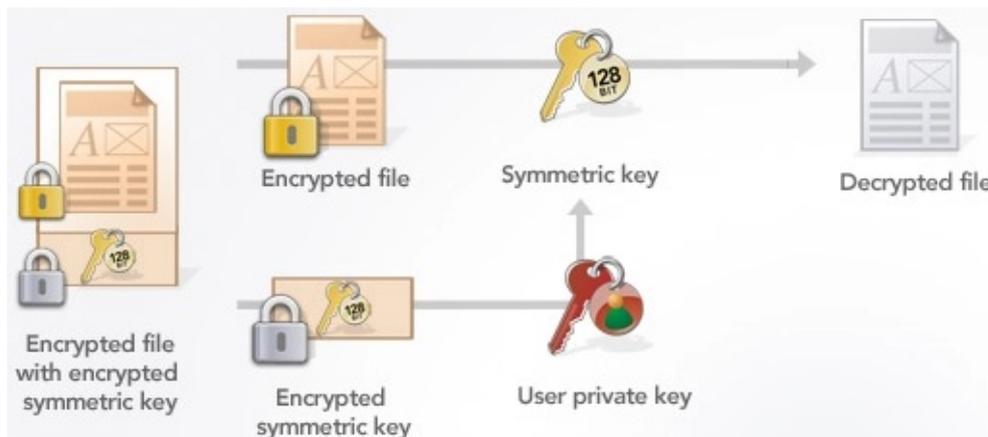
Asymmetric encryption is more complicated than symmetric encryption, not only because it uses **public** and **private keys**, but because asymmetric encryption can encrypt / decrypt only small messages, which should be mapped to the underlying math of the public-key cryptosystem. In the RSA system, the input message should be transformed to big integer (e.g. using OAEP padding), while in ECC the message should be mapped to elliptic curve point to be encrypted. Additionally, asymmetric ciphers are significantly slower than symmetric ciphers (e.g. the RSA encryption is 1000 times slower than AES).

To overcome the above limitations and to allow encrypting messages of any size, modern cryptography uses **asymmetric encryption schemes** (also known as **public key encryption schemes / asymmetric encryption constructions / hybrid encryption schemes**), like **key encapsulation mechanisms** and **integrated encrypted schemes**, which combine asymmetric encryption with symmetric key ciphers.

This is how a large document or file can be **encrypted** by combining **public-key cryptography** and **symmetric crypto algorithm**:



This is the corresponding **decryption** process (decrypt an encrypted large document using **public-key cryptography** and **symmetric crypto algorithm**):



Examples of such asymmetric encryption schemes are: [RSA-OAEP](#), [RSA-KEM](#) and [ECIES-KEM](#).

Integrated Encryption Schemes

Integrated encryption schemes (IES) are modern public key encryption schemes, which combine symmetric ciphers, asymmetric ciphers and key-derivation algorithms to provide secure **public-key based encryption**. In IES scheme asymmetric algorithms (like RSA or ECC) are used to encrypt or encapsulate a symmetric key, used later by symmetric ciphers (like AES or ChaCha20) to encrypt the input message. Some IES schemes provide also message authentication. Examples of IES schemes are [DLIES](#) (Discrete Logarithm Integrated Encryption Scheme) and [ECIES](#) (Elliptic Curve Integrated Encryption Scheme).

Key Encapsulation Mechanisms (KEMs)

A **key encapsulation mechanisms (KEM)** are asymmetric cryptographic techniques to encrypt a secret key that is used to encrypt an input message using symmetric cryptographic cipher, called a data encapsulation mechanism (DEM). **Key encapsulation mechanisms** are used in the hybrid encryption schemes and in the integrated encryption schemes, where a random element is generated in the underlying public-key cryptosystem and a symmetric key is

derived from this random element by hashing. This approach simplifies the process of combining asymmetric and symmetric encryption. Examples of modern key encapsulation mechanisms are: **RSA-KEM**, **ECIES-KEM** and **PSEC-KEM**.

Digital Signatures

In cryptography **digital signatures** provide message **authentication**, **integrity** and **non-repudiation** for digital documents. Digital signatures work in the public-key cryptosystems and use a public / private key pairs. Message **signing** is performed by the **private key** and message **verification** is performed by the corresponding **public key**.

A **message signature** mathematically guarantees that certain message was signed by certain (secret) **private key**, which corresponds to certain (non-secret) **public key**. After a message is signed, the message and **the signature cannot be modified** and thus message **authentication** and **integrity** is provided. Anyone, who knows the **public key** of the message signer, can **verify the signature**. After signing the signature author cannot reject the act of signing (this is known as **non-repudiation**).

Digital signatures are widely used today for signing digital contracts, for authorizing bank payments and signing transactions in the public blockchain systems for transferring digital assets.

Most public-key cryptosystems like **RSA** and **ECC** provide secure digital signature schemes like **DSA**, **ECDSA** and **EdDSA**. We shall discuss the digital signatures in greater detail later in this section.

Key Exchange Algorithms

In cryptography **key exchange algorithms** (**key agreement protocols** / **key negotiation schemes**) allow cryptographic keys to be exchanged between two parties, allowing the use of a cryptographic algorithm, in most cases symmetric encryption cipher. For example, when a laptop connects to the home **WiFi router**, both parties agree on a **session key**, used to symmetrically encrypt the network traffic between them.

Most key-exchange algorithms are based on public-key cryptography and the math behind this system: discrete logarithms, elliptic curves or other.

Anonymous key exchange, like Diffie–Hellman (**DHKE** and **ECDH**), does not provide authentication of the parties, and is thus vulnerable to **man-in-the-middle attacks**, but is safe from **traffic interception (sniffing) attacks**.

Authenticated key agreement schemes authenticate the identities of parties involved in the key exchange and thus prevent man-in-the-middle attacks by use of **digitally signed keys** (e.g. **PKI certificate**), **password-authenticated key agreement** or other method.

The RSA Cryptosystem - Concepts

The **RSA cryptosystem** is one of the first **public-key cryptosystems**, based on the math of the **modular exponentiations** and the computational difficulty of the **RSA problem** and the closely related **integer factorization problem (IFP)**. The RSA algorithm is named after the initial letters of its authors (**R**ivest–**S**hamir–**A**dleman) and is widely used in the early ages of computer cryptography.

Later, when **ECC** cryptography evolved, the **ECC** slowly became dominant in the asymmetric cryptosystems, because of its higher security and shorter key lengths than **RSA**.

The **RSA** algorithm provides:

- **Key-pair generation:** generate random **private key** (typically of size 1024-4096 bits) and corresponding **public key**.
- **Encryption:** **encrypt** a secret message (integer in the range $[0 \dots \text{key_length}]$) using the public key and **decrypt** it back using the secret key.
- **Digital signatures:** **sign** messages (using the private key) and **verify** message signature (using the public key).
- **Key exchange:** securely transport a secret key, used for encrypted communication later.

RSA can work with keys of different **keys of length**: 1024, 2048, 3072, 4096, 8129, 16384 or even more bits. Key length of 3072-bits and above are considered **secure**. Longer keys provide higher security but consume **more computing time**, so there is a tradeoff between security and speed. Very long RSA keys (e.g. 50000 bits or 65536 bits) may be **too slow for practical use**, e.g. key generation may take from several minutes to several hours.

RSA Key Generation

Generating an RSA public + private key pair involves the following:

Using some non-trivial **math computations from the number theory**, find three very large integers **e**, **d** and **n**, such that:

- $(m^e)^d \equiv m \pmod{n}$ for all m in the range $[0 \dots n]$

The integer number **n** is called "**modulus**" and it defines the **RSA key length**. It is typically very large prime number (e.g. 2048 bits).

The pair $\{n, e\}$ is the **public key**. It is designed to be shared with everyone. The number **e** is called "**public key exponent**". It is usually **65537** (0x010001).

The pair $\{n, d\}$ is the **private key**. It is designed to be kept in secret. It is practically infeasible to calculate the private key from the public key $\{n, e\}$. The number **d** is called "**private key exponent**" (the secret exponent).

RSA Public Key - Example

Example of 2048-bit **RSA public key** (represented as 2048-bit hexadecimal integer modulus **n** and 24-bit public exponent **e**):

```
n = 0xa709e2f84ac0e21eb0caa018cf7f697f774e96f8115fc2359e9cf60b1dd8d4048d974cdf8422bef6be3c16
2b04b916f7ea2133f0e3e4e0eee164859bd9c1e0ef0357c142f4f633b4add4aab86c8f8895cd33fbf4e024d9a3ad
6be6267570b4a72d2c34354e0139e74ada665a16a2611490debb8e131a6cffc7ef25e74240803dd71a4fcd953c98
8111b0aa9bbc4c57024fc5e8c4462ad9049c7f1abed859c63455fa6d58b5cc34a3d3206ff74b9e96c336dbacf0cd
d18ed0c66796ce00ab07f36b24cbe3342523fd8215a8e77f89e86a08db911f237459388dee642dae7cb2644a03e7
1ed5c6fa5077cf4090fafa556048b536b879a88f628698f0c7b420c4b7
e = 0x010001
```

The same RSA public key, encoded in the traditional for RSA format **PKCS#8 PEM ASN.1** looks like this:

```

-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEApwni+ErA4h6wyqAYz39p
f3d0lvgrX8I1npz2Cx3Y1ASNl0zfHCK+9r48FisEuRb36iEz80Pk407hZIwB2cHg
7wNXwUL09j00rdSquGyPiJXNM/v04CTZo61r5iZ1cLSnLSw0NU4B0edK2mZaFqJh
FJDeu44TGmz/x+8150JAgD3XGk/NlTyYgRGwqpu8TFcCT8XoxEYq2QScfxq+2FnG
NFX6bVi1zDSj0yBv90ue1sM226zwdG00MZn1s4AqwfzayTL4zQ1I/2CFajnf4no
agjbr8jdFk4je5kLa58smRKA+ce1cb6UHFPQJD6+1VgSLU2uHmoj2KGmPDHtCDE
twIDAQAB
-----END PUBLIC KEY-----

```

The above PEM ASN.1-encoded message, holding the RSA public key, can be decoded here: <https://lapo.it/asn1js>.

RSA Private Key - Example

Example of 2048-bit **RSA private key**, corresponding to the above given public key (represented as hexadecimal 2048-bit integer modulus n and 2048-bit secret exponent d):

```

n = 0xa709e2f84ac0e21eb0caa018cf7f697f774e96f8115fc2359e9cf60b1dd8d4048d974cdf8422bef6be3c16
2b04b916f7ea2133f0e3e4e0eee164859bd9c1e0ef0357c142f4f633b4add4aab86c8f8895cd33fbf4e024d9a3ad
6be6267570b4a72d2c34354e0139e74ada665a16a2611490debb8e131a6cfff7ef25e74240803dd71a4fcd953c98
8111b0aa9bbc4c57024fc5e8c4462ad9049c7f1abed859c63455fa6d58b5cc34a3d3206ff74b9e96c336dbacf0cd
d18ed0c66796ce00ab07f36b24cbe3342523fd8215a8e77f89e86a08db911f237459388dee642dae7cb2644a03e7
1ed5c6fa5077cf4090fafafa556048b536b879a88f628698f0c7b420c4b7
d = 0x10f22727e552e2c86ba06d7ed6de28326eef76d0128327cd64c5566368fcd1a9f740ad8dd221419a5550fc
8c14b33fa9f058b9fa4044775aaf5c66a999a7da4d4fdb8141c25ee5294ea6a54331d045f25c9a5f7f47960acbae
20fa27ab5669c80eaf235a1d0b1c22b8d750a191c0f0c9b3561aaa4934847101343920d84f24334d3af05fede0e3
55911c7db8b8de3bf435907c855c3d7eede4f148df830b43dd360b43692239ac10e566f138fb4b30fb1af0603cf
cf0cd8adf4349a0d0b93bf89804e7c2e24ca7615e51af66dccb71a1204e2107abbee4259f2cac917fafa3b029b
af13c4dde7923c47ee3fec248390203a384b9eb773c154540c5196bce1

```

The same RSA private key, encoded in the traditional for RSA format **PKCS#8 PEM ASN.1** looks a bit longer:

```

-----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKCAQEApwni+ErA4h6wyqAYz39pf3d0lvgrX8I1npz2Cx3Y1ASNl0zf
hCK+9r48FisEuRb36iEz80Pk407hZIwB2cHg7wNXwUL09j00rdSquGyPiJXNM/v0
4CTZo61r5iZ1cLSnLSw0NU4B0edK2mZaFqJhFJDeu44TGmz/x+8150JAgD3XGk/N
lTyYgRGwqpu8TFcCT8XoxEYq2QScfxq+2FnGNFX6bVi1zDSj0yBv90ue1sM226z
zdG00MZn1s4AqwfzayTL4zQ1I/2CFajnf4noagjbr8jdFk4je5kLa58smRKA+ce
1cb6UHFPQJD6+1VgSLU2uHmoj2KGmPDHtCDEtwIDAQABAoIBABDyJyflUuLIa6Bt
ftbeKDJu73bQEoMnzWTFVmnO/cGp90CtjdIhQZpVUPyMFLM/qfBYufpARHdar1xm
qZmn2k1P24FBw171KU6mpUMx0EXyXJpff0eWCsuuIPonq1ZpyA6vI1odCxwiuNdQ
oZHA8MmzVhqSTSEcQE0SDYTYQzTTrwX+3g41WRHH24uN479DWQfIVcPX7u3k8U
jfgwtD3TYLQ2kiOawQ5WbxOPtLMPsa8GA8/PDNit9DSaDQuTv4mATnWuJMp2FeUa
9m3M/bcaEgtiEHq77kJZ8srJF/r+0wKbrxPE3eeSPEfuP+wkg5AgOjhLnrdzwVRU
DFGwv0ECgYEAyIk7F0S0AGn2aryhw9CihDfimigCxEmtIO5q7mnItCfeQwYpSx72
1flpJNgfPc9DDfAZ2hLSsBlAPLU0a0Cuny9PCBwVuxi1WjLVaeZCV2bf11mAgw2
fjLkAXT34IX+HZ160VoetSwq9ibfkJHeCAPnh/yjdB3Vs+2wxNkU8m8CgYEA1Tzm
mjJq7M6f+zMo7DpRwFazGMmrLKFmHiGBY6sEg7EmoeH2CkAQePIGQw/Rk16gWJR6
DtUZ9666sjCh6/79rx2xg+9AB76XTFFzIx0k9cm49cIosDMk4mogSfK0Zg8nVbyW
5nEb//9JCrZ18g41D3IrT5VJof4MhfdBUjAS1jkCgYB+RDIPv3+bNx0KLGWpFwgN
Omb667B6S2ya4x227KdBPfkwD9HYosnQZDdOxvIvmUZ0bPLqJan1aaDR2Krgi1S
oNJCNPzGmwbMGvTU1Pd+Nys9NfjR0ykKix7/b9fXzman2oJdovvs0W/pf6bzD3V/
FH5HWKLOrS5u4X3JJGqVDwKBGQCd953FwW/gujld+EpqpdGGMTRAOrXqPC7QR3X5
Beo0PPon1q0UeF07m9/zsZJfCJBPM0nS8s054w7ESTA0YhpQBAPcx/2HMUsrnIj

```

```

HBxqUOQKe6l0zo6WhJQI8/+cU8GKDEm1sU1S3iWYIA9EICJoTOW08R04BjQ00jS7
1A1AUQKBgH1HrV/6S/4hjvMp+30hX5DpZviUDiwcGOGasmIYXAgwXepJUq0xN6aa
lnT+ykLGSMY/LABQiNZALZQtWk35KTshnThK6zB4e9p8JUCVrFpssJ2NCrMY3SU
qw87K1W6engeDrMunkJ/PmvSDLYeGiYWmEKQbLQchTxx1IEddXkk
-----END RSA PRIVATE KEY-----

```

It holds **the entire RSA key-pair structure**, along with several additional parameters: 2048-bit modulus n , 24-bit public exponent e , 2048-bit secret exponent d , first factor p , second factor q , and 3 other integers from the RSA internal data structure:

The above PEM ASN.1-encoded message, holding the RSA private key data, can be decoded here:

<https://lapo.it/asn1js>.

RSA Cryptography: Encrypt a Message

Encrypting a message using certain RSA **public key** $\{n, e\}$ is done by the following transformation:

- $encryptedMsg = (msg)^e \bmod n$

The msg here is a number in the range $[0 \dots n)$. Text messages should be **encoded as integers** in the range $[0 \dots n)$ before encryption (see [EAOP](#)). For larger texts, **hybrid encryption** should be used (encrypt a secret key and use it to symmetrically encrypt the text, see [RSA-KEM](#)).

The above operation **cannot be reversed**: no efficient algorithm exists to calculate msg from $encryptedMsg$, e and n (see [the RSA problem](#)), which all are **public** (non-secret) by design.

RSA Cryptography: Decrypt a Message

Decrypting the encrypted message using the corresponding RSA **private key** $\{n, d\}$ is done by the following transformation:

- $decryptedMsg = (encryptedMsg)^d \bmod n$

Why this is correct? Recall, that by definition the RSA key-pair has the following property:

- $(m^e)^d \equiv m \pmod{n}$ for any m in the range $[0 \dots n)$

From the encryption transformation we have:

- $encryptedMsg = (msg)^e \bmod n$

Hence:

- $decryptedMsg = (encryptedMsg)^d \bmod n = ((msg)^e \bmod n)^d \bmod n = (msg)^{e \cdot d} \bmod n = (msg) \bmod n = msg$

RSA Encrypt and Decrypt - Example

Let examine one **example of RSA encryption and decryption**, along with the calculations, following the above formulas. Assume we have generated the RSA public-private key pair:

- modulus $n = 143$
- public exponent $e = 7$
- private exponent $d = 103$
- public key = $\{n, e\} = \{143, 7\}$
- private key = $\{n, d\} = \{143, 103\}$

Let's **encrypt** a secret message $msg = 83$. Just follow the formula:

- $encryptedMsg = msg^e \bmod n = 83^7 \bmod 143 = 27136050989627 \bmod 143 = 8$

Now, let's **decrypt** the encrypted message back to its original value:

- $\text{decryptedMsg} = \text{encryptedMsg}^d \bmod n = 8^{103} \bmod 143 = 1042962419883256876169444192465601618458351817556959360325703910069443225478828393565899456512 \bmod 143 = \mathbf{83}$

The RSA calculations work correctly. This is because the key-pair meets the RSA property:

- $(m^e)^d \equiv m \pmod{n}$ for all m in the range $[0 \dots n)$
- $(m^7)^{103} \equiv m \pmod{143}$ for all m in the range $[0 \dots 143)$

In the real world, typically the RSA modulus n and the private exponent d are 3072-bit or 4096-bit integers and the public exponent e is 65537.

For further reading, look at this excellent explanation about **how RSA works** in detail with explanations and examples: <http://doctrina.org/How-RSA-Works-With-Examples.html>.

Because RSA encryption is a **deterministic** (has no random component) attackers can successfully launch a **chosen plaintext attack** against by encrypting likely plaintexts with the public key and test if they are equal to the ciphertext. This may not be a problem, but is a **weakness**, that should be considered when developers choose an encryption scheme.

Hybrid encryption schemes like **RSA-KEM** solve this vulnerability and allow encrypting longer texts.

RSA Encryption / Decryption - Examples in Python

Now let's demonstrate how the RSA algorithms works by a simple **example in Python**. The below code will generate **random RSA key-pair**, will **encrypt** a short message and will **decrypt** it back to its original form, using the **RSA-OAEP** padding scheme.

First, install the `pycryptodome` package, which is a powerful Python library of low-level cryptographic primitives (hashes, MAC codes, key-derivation, symmetric and asymmetric ciphers, digital signatures):

```
pip install pycryptodome
```

RSA Key Generation

Now, let's write the Python code. First, **generate the RSA keys** (1024-bit) and print them on the console (as hex numbers and in the PKCS#8 PEM ASN.1 format):

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
import binascii

keyPair = RSA.generate(3072)

pubKey = keyPair.publickey()
print(f"Public key: (n={hex(pubKey.n)}, e={hex(pubKey.e)})")
pubKeyPEM = pubKey.exportKey()
print(pubKeyPEM.decode('ascii'))

print(f"Private key: (n={hex(pubKey.n)}, d={hex(keyPair.d)})")
privKeyPEM = keyPair.exportKey()
print(privKeyPEM.decode('ascii'))
```

We use short key length to keep the sample input short, but in a real world scenario it is recommended to use 3072-bit or 4096-bit keys.

RSA Encryption

Next, **encrypt the message** using **RSA-OAEP** encryption scheme (RSA with PKCS#1 OAEP padding) with the **RSA public key**:

```
msg = b'A message for encryption'
encryptor = PKCS1_OAEP.new(pubKey)
encrypted = encryptor.encrypt(msg)
print("Encrypted:", binascii.hexlify(encrypted))
```

RSA Decryption

Finally, **decrypt the message** using using **RSA-OAEP** with the **RSA private key**:

```
decryptor = PKCS1_OAEP.new(keyPair)
decrypted = decryptor.decrypt(encrypted)
print('Decrypted:', decrypted)
```

Sample Output

A **sample output** of the code execution for the entire example is given below:

```
Public key: (n=0x9a11485bccb9569410a848fb1afdf2a81b17c1fa9f9eb546fd1deb873b49b693a4edf20eb83
62c085cd5b28ba109dbad2bd257a013f57f745402e245b0cc2d553c7b2b8dbba57ebda7f84c-fb32b7d9c254f03db
d0188e4b8e40c47b64c1bd2572834b936ffc3da9953657ef8bee80c49c2c12933c8a34804a00eb4c81248e01f, e
=0x10001)
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCaEUhbzLlWlBCoSPsa/fKoGxgFB
+p+etUb9HeuH00m2k6Tt8g64NiwIXNWyI6EJ260r0legE/V/dFQC4kWwzC1VPHsr
jbulfr2n+Ez7MrfZw1TwPb0Bi0S45AxHtkwb0lcoNLk2/8PamVN1fvi+6AxJwsEp
M8ijSASgDrTIEkjgHwIDAQAB
-----END PUBLIC KEY-----
Private key: (n=0x9a11485bccb9569410a848fb1afdf2a81b17c1fa9f9eb546fd1deb873b49b693a4edf20eb8
362c085cd5b28ba109dbad2bd257a013f57f745402e245b0cc2d553c7b2b8dbba57ebda7f84c-fb32b7d9c254f03d
bd0188e4b8e40c47b64c1bd2572834b936ffc3da9953657ef8bee80c49c2c12933c8a34804a00eb4c81248e01f,
d=0x318ab12be3cf0d4a1b7921cead454fcc42ba070462639483394d6fb9529547827e9c8d23b294a8e01f8a1019
da34e350f2307740e06a270bef1fe646e6ad213e31b528fdd5f5d03e633c07c44755ed622a629d79e822c095ebdf
9cc80e517b5566dd3d3e5b16ec737987337a0e497fdb4b5ad97af41c1c3cdd87542a4637d81)
-----BEGIN RSA PRIVATE KEY-----
MIICXAIBAAKBgQCaEUhbzLlWlBCoSPsa/fKoGxgFB+p+etUb9HeuH00m2k6Tt8g64
NiwIXNWyI6EJ260r0legE/V/dFQC4kWwzC1VPHsrjbulfr2n+Ez7MrfZw1TwPb0B
i0S45AxHtkwb0lcoNLk2/8PamVN1fvi+6AxJwsEpM8ijSASgDrTIEkjgHwIDAQAB
AoGAMYqXk+PPDUobeSHOrUVPzEK6BwRiY5SDOU1vuVKVR4J+nI0jsspSo4B+KEBna
NONQ8jB3Q0BqJwvvh+ZG5q0hPjG1KP3V9dA+YzwHxEdV7WIqYp156CLA1evfnMgO
UXtVZt09P1sw7HN5hzN6Dk1/26S1rZevQcHDzdhd1QqRjfyECQQDGUIQX10iAcGo
d5YqAGpWe0wzJ0UypeqZcZqS9MVe90kjjopCkkyntifdN/1oG7S/1KUMtLoGHqntb
c428z00/AkEAxyV0cmuJbFdfM0x2XhZ+ge/7putIx76RHDOjBpM6VQXpLEFj54kB
qGLAB7Sxr7P4AFrEjfcKJOp2YMI5BreboQJAb3EUZHT/WeDdJLutzpKPQ3x7oykM
wfQkxbxXYzVd16u96BkT6W0/gCb6hXs05zj32x1/hgfHyRvGCGjKKZdtwpwJBAJ74
y0g7h+wwoxJ0S1k4Y6yeQikxUVwCSBxLCCnjr0ohsaJPJMrz2L30YtVInFkH01L
i/Q4AWZmtDDxWkx+bYECQG8e6bGoszuX5xjvhEBs1Iws9+nMzMuYBR8HvhLo58B5
N8dk3nIsLs3UncKLiWubMAciU5jUxZoqWpRXXwECKE=
-----END RSA PRIVATE KEY-----
Encrypted: b'99b331c4e1c8f3fa227aacd57c85f38b7b7461574701b427758ee4f94b1e07d791ab70b55d672ff
55dbe133ac0bea16fc23ea84636365f605a9b645e0861ee11d68a7550be8eb35e85a4bde6d73b0b956d000866425
511c7920cdc8a3786a4f1cb1986a875373975e158d74e11ad751594de593a35de765fe329c0d3dfbbfedc '
Decrypted: b'A message for encryption'
```

Notes:

- If you run the above example, your output will be different, because it generates different **random RSA key-pair** at each execution.
- Even if you **encrypt the same message several times** with the same public key, you will get **different output**. This is because the **OAEP** padding algorithm injects some randomness with the padding.
- If you try to **encrypt larger messages**, you will get an exception, because the **1024-bit key limits** the maximum message length.

Now **play with the above code**, modify it and run it to learn how RSA works in action.

Exercises: Encrypt / Decrypt Messages using RSA

In this exercise you shall **encrypt** and **decrypt** messages using the **RSA** public-key cryptosystem.

Encrypt Message with RSA-OAEP

You are given a **text message** and a **RSA public key** (in PEM format). Write a program to **encrypt the message**, using the **RSA-OAEP** encryption scheme (RSA + PKCS#1 OAEP padding).

Input:

- First line: the input **message**
- Next few lines: the **RSA public key** (in the [PKCS#8 PEM ASN.1](#) format)
- The public key length can be 512 bits, 1024 bits, 2048 bits, 3072 bits or 4096 bits.

Output:

- The **encrypted message**, printed as **hex** string.

Write your code in programming language of choice.

Sample input:

```
Secret message
-----BEGIN PUBLIC KEY-----
MFwwDQYJKoZIhvcNAQEBBQADSwAwSAJBAMYhCcGpfoebriBbFaUMMwH3B5t7udir
ODJehnQTP1WLf9SVfQdx0v9ATJ2Rs5kQjdJ/wZYunMBVq6/FhgPZexsCAwEAAQ==
-----END PUBLIC KEY-----
```

The above input uses a **512-bit RSA public key** and a small plain text message, that can fit inside the key length (after the OAEP padding).

Sample output (for the above input):

```
218dd78c5e14b4d58efd10575b521db46c0caa5c699134abf18bbeaac170cfe446e25d0d82257082539e4ccd3e0a
a8bffc1b07d2bde9e635a7b9b7fc6cf4c266
```

Note: the above output should be **different at each execution** due to the randomness injected by the OAEP padding algorithm.

Decrypt a Message with RSA-OAEP

You are given a RSA-OAEP-encrypted ciphertext (as hex string) and a **RSA private key** (in PEM format). Write a program to **decrypt the message**, using the **RSA-OAEP** encryption scheme (RSA + PKCS#1 OAEP padding).

Input:

- First line: the **ciphertext** (the encrypted message), given as **hex** string
- Next few lines: the **RSA private key** (in the [PKCS#8 PEM ASN.1](#) format)

Output:

- Print the **decrypted message** as plain text
- Print `Decryption failed!` in case of problem

Write your code in programming language of choice.

Sample input:

```

218dd78c5e14b4d58efd10575b521db46c0caa5c699134abf18bbeaac170cfe446e25d0d82257082539e4ccd3e0a
a8bffc1b07d2bde9e635a7b9b7fc6cf4c266
-----BEGIN RSA PRIVATE KEY-----
MIIB0gIBAAJBAMyHcGpfoebriBbFaUMMwH3B5t7udirODJehnQTP1WLf9SVfQdx
0v9ATJ2Rs5kQjdJ/wZYunMBVq6/FhgPZexsCAwEAAQJAbNsZBkTzMsWqHq3JuuPz
jk3CSP7ye/i5Grnfgx0a7W0GpVrEDQNo0iihEf5pRAfaazEdfJX2Tj+auuv06392
kQIhA0eJahRw0t8cYr0LZzHHf7LWQg1RaTbtKShqmbLdBZMzAiEA2xADyA3xGXc1
txN0D0fSycwFyqkd1fsuyAwKibPteHkCIQDJ1P6UzHR1UwA434HYYeJ0U3mDN+V4
z0oI4kwTIBohAwIglRqv09EFiUudSnxf2RDqqlXcu+4W/IE/K904AL9uSECICeT
tkAnJHB7k6fvox6ErJV53w06bUF1jGw8yHuaCCHX
-----END RSA PRIVATE KEY-----

```

The above input uses a **512-bit RSA private key** and an encrypted **ciphertext** of the same length.

Sample output (for the above input):

```
Secret message
```

Another **sample input** (wrong 512-bit private key):

```

218dd78c5e14b4d58efd10575b521db46c0caa5c699134abf18bbeaac170cfe446e25d0d82257082539e4ccd3e0a
a8bffc1b07d2bde9e635a7b9b7fc6cf4c266
-----BEGIN RSA PRIVATE KEY-----
MIIB0QIBAAJBAD0kbrC4AxpqBgwVPPb8IoI/kdQkF1twrfQtoMkHgB71vpY6Sg
68CUA7Ejq/dbAHlvFdXqweK9vXH3kFpc8pcCAwEAAQJAaFr1Xm2Pun2dgWthoTOi
0YCe6LKEsf43dMJIab1mfYi1trSpGaoTXLvHR+NaAgqcr9KAH24Mi05ttUBcWRSI
QQIhAOLTSyeDznq5rqdwBLU8p6USpeImRhWRNcCHA/QLxcaPAiEAqu+01p1YB3Mp
GKgB9PvZE3TZqmlgtEFfSMYinF3g13kCIF9FjpCXMYkkysZLWG2e32+HaK0XneJb
Lq+iRjfQZg7jAiBcm6D1YRV6I8gWFZ/JzFBVHC95BdJgljYGI2JJI+QuBcQIGLJjH
IPctSCUtukz+7fde0dw/0FINcUGvnQyuEK34UxE=
-----END RSA PRIVATE KEY-----

```

The corresponding **output** should be:

```
Decryption failed!
```

Note that the **RSA-OAEP** padding algorithm has built-in checksum, which allows to detect incorrect decryption attempts, but it is not an authenticated encryption scheme.

* Implement Hybrid Encryption / Decryption with RSA-KEM

Write a program to **encrypt** a large message (bigger than the RSA key length, e.g. a PDF document) using the **RSA-KEM** hybrid encryption scheme with **AES** symmetric encryption (use block mode of choice, e.g. **GCM** or **CTR**).

Hint:

- Check this example first: <https://github.com/digitalbazaar/forge#rsakem>.
- Note that in some languages it is hard to find and **RSA-KEM** implementation, so you can skip this exercise or use another **hybrid encryption scheme** (e.g. RSA + AES + HMAC).

Input:

- The **message** for encryption
- **RSA public key** (in PEM format)

Output:

- The encrypted **ciphertext** (hex string)
- The random **IV** salt for the AES cipher (hex string)
- The authentication **tag / MAC** for the encrypted message (hex string)
- The **encapsulated secret key** for the AES algorithm (hex string)

Write a program to **decrypt** given encrypted message, produced by the previous exercise, using the **RSA-KEM** hybrid encryption scheme with **AES** symmetric encryption (use block mode of choice, e.g. **GCM** or **CTR**).

Input:

- The encrypted **ciphertext** (hex string)
- The random **IV** salt for the AES cipher (hex string)
- The authentication **tag / MAC** for the encrypted message (hex string)
- The **encapsulated secret key** for the AES algorithm (hex string)

Output:

- The decrypted original plaintext **message**
- Print `Decryption failed!` if the message decryption is not successful (e.g. wrong password)

Elliptic Curve Cryptography (ECC) - Concepts

The **Elliptic Curve Cryptography (ECC)** is modern **family of public-key cryptosystems**, which is based on the algebraic structures of the **elliptic curves over finite fields** and on the difficulty of the **Elliptic Curve Discrete Logarithm Problem (ECDLP)**.

ECC implements all major capabilities of the asymmetric cryptosystems: **encryption, signatures and key exchange**.

The **ECC cryptography** is considered a natural modern **successor of the RSA** cryptosystem, because ECC uses **smaller keys** and signatures than RSA for the same level of security and provides very **fast key generation, fast key agreement and fast signatures**.

ECC Keys

The **private keys** in the ECC are integers (in the range of the curve's field size, typically **256-bit** integers). Example of 256-bit ECC private key (hex encoded, 32 bytes, 64 hex digits) is:

```
0x51897b64e85c3f714bba707e867914295a1377a7463a9dae8ea6a8b914246319 .
```

The **key generation** in the ECC cryptography is as simple as securely generating a **random integer** in certain range, so it is extremely fast. Any number within the range is valid ECC private key.

The **public keys** in the ECC are **EC points** - pairs of integer coordinates $\{x, y\}$, laying on the curve. Due to their special properties, **EC points** can be **compressed** to just one coordinate + 1 bit (odd or even). Thus the **compressed public key**, corresponding to a 256-bit ECC private key, is a **257-bit** integer. Example of ECC public key

(corresponding to the above private key, encoded in the Ethereum format, as hex with prefix `02` or `03`) is:

```
0x02f54ba86dc1ccb5bed0224d23f01ed87e4a443c47fc690d7797a13d41d2340e1a .
```

In this format the public key actually takes 33 bytes (66 hex digits), which can be optimized to exactly 257 bits.

Curves and Key Length

ECC crypto algorithms can use different underlying **elliptic curves**. Different curves provide different level of **security** (cryptographic strength), different **performance** (speed) and different **key length**, and also may involve different algorithms.

ECC curves, adopted in the popular cryptographic libraries and security standards, have **name** (named curves, e.g. `secp256k1` or `Curve25519`), **field size** (which defines the key length, e.g. **256-bit**), security **strength** (usually the field size / 2 or less), **performance** (operations/sec) and many other parameters.

ECC keys have **length**, which directly depends on the underlying curve. In most applications (like OpenSSL, OpenSSH and Bitcoin) the default **key length** for the ECC private keys is **256 bits**, but depending on the curve many different ECC key sizes are possible: 192-bit (curve `secp192r1`), 233-bit (curve `sect233k1`), 224-bit (curve `secp224k1`), 256-bit (curves `secp256k1` and `Curve25519`), 283-bit (curve `sect283k1`), 384-bit (curves `p384` and `secp384r1`), 409-bit (curve `sect409r1`), 414-bit (curve `Curve41417`), 448-bit (curve `Curve448-Goldilocks`), 511-bit (curve `M-511`), 521-bit (curve `P-521`), 571-bit (curve `sect571k1`) and many others.

ECC Algorithms

Elliptic-curve cryptography (ECC) provides several groups of algorithms, based on the math of the elliptic curves over finite fields:

- ECC **digital signature** algorithms like **ECDSA** (for classical curves) and **EdDSA** (for twisted Edwards curves).
- ECC **encryption** algorithms and hybrid encryption schemes like the **ECIES** integrated encryption scheme and **EEEC** (EC-based ElGamal).
- ECC **key agreement** algorithms like **ECDH**, **X25519** and **FHMQV**.

All these algorithms use a **curve** behind (like `secp256k1`, `curve25519` or `p521`) for the calculations and rely on the difficulty of the **ECDLP** (elliptic curve discrete logarithm problem). All these algorithms use public / private key pairs, where the **private key** is an integer and the **public key** is a point on the elliptic curve (EC point). Let's get into details about the elliptic curves over finite fields.

Elliptic Curves

In mathematics **elliptic curves** are plane algebraic curves, consisting of all points $\{x, y\}$, described by the equation:

$$Ax^3 + Bx^2y + Cxy^2 + Dy^3 + Ex^2 + Fxy + Gy^2 + Hx + Iy + J = 0,$$

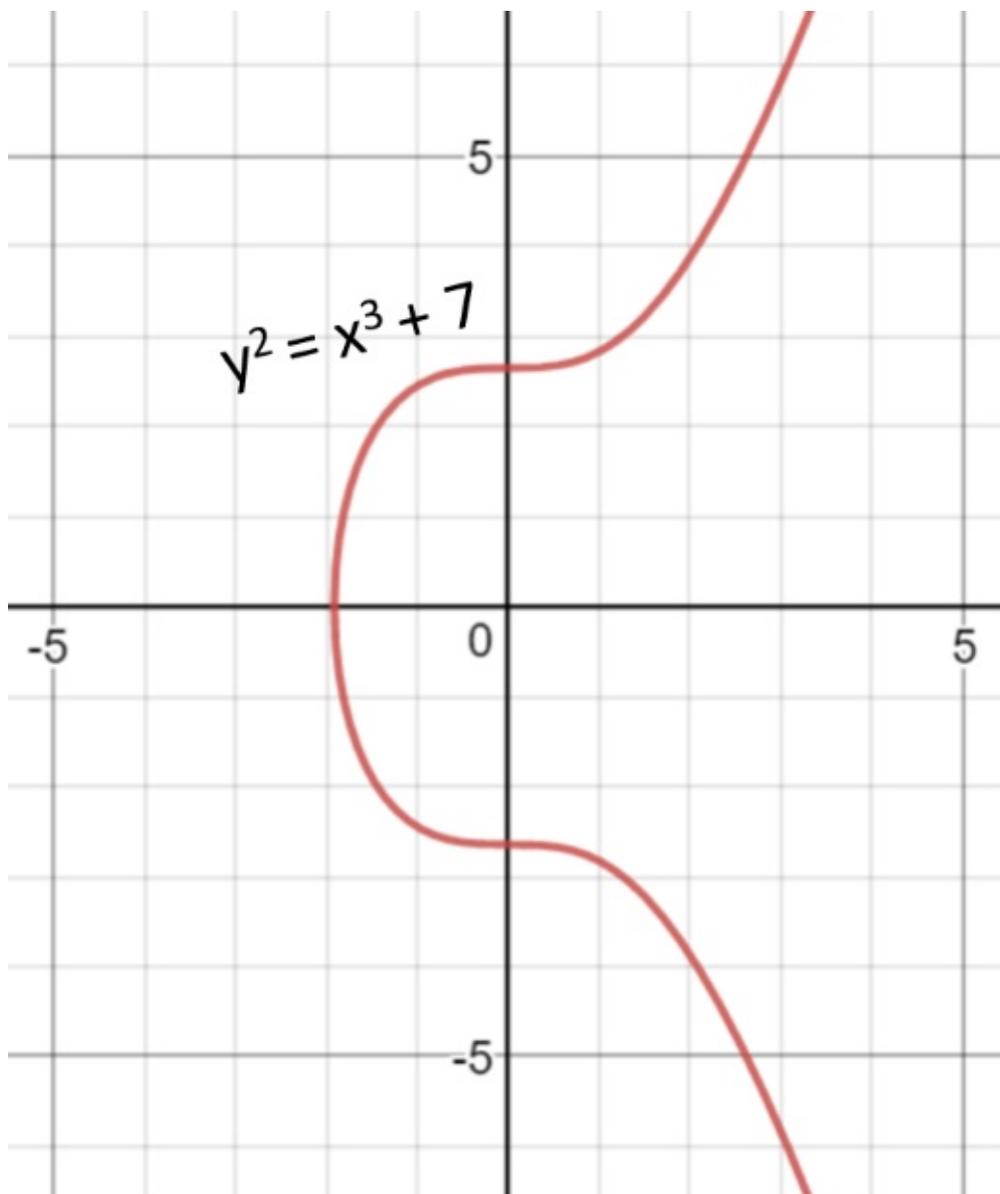
Cryptography uses **elliptic curves** in a simplified form (Weierstrass form), which is defined as:

- $y^2 = x^3 + ax + b$

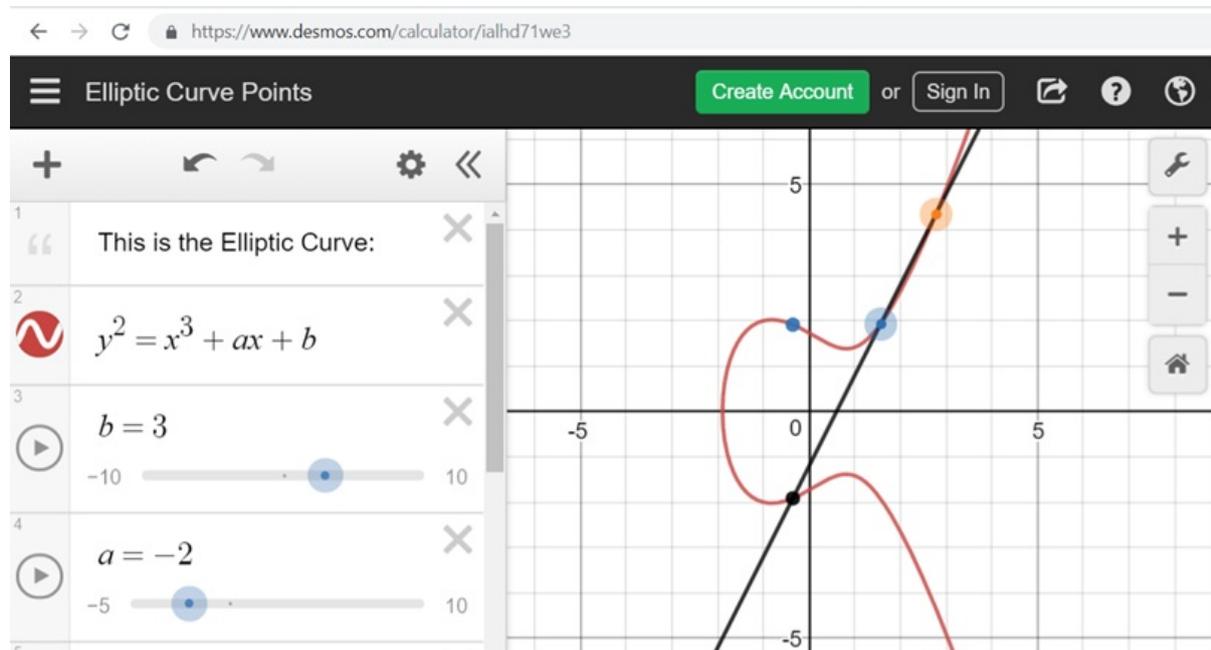
For example, the **NIST curve** `secp256k1` (used in Bitcoin) is based on an elliptic curve in the form:

- $y^2 = x^3 + 7$ (the above elliptic curve equation, where $a = 0$ and $b = 7$)

This is a visualization of the above elliptic curve:



To learn more about the equations of the elliptic curves and how they look like, play a bit with this **online elliptic curve visualization tool**: <https://www.desmos.com/calculator/ialhd71we3>.



Elliptic Curves over Finite Fields

The **elliptic curve cryptography (ECC)** uses **elliptic curves over the finite field \mathbf{p}** (where \mathbf{p} is prime and $\mathbf{p} > 3$) or $\mathbf{2}^m$ (where the fields size $\mathbf{p} = 2^m$). This means that the field is a **square matrix** of size $\mathbf{p} \times \mathbf{p}$ and the points on the curve are limited to **integer coordinates** within the field only. All algebraic operations within the field (like point addition and multiplication) result in another point within the field. The elliptic curve equation over the finite field \mathbf{p} takes the following modular form:

- $y^2 \equiv x^3 + ax + b \pmod{\mathbf{p}}$

Respectively, the "Bitcoin curve" `secp256k1` takes the form:

- $y^2 \equiv x^3 + 7 \pmod{\mathbf{p}}$

Unlike **RSA**, which uses for its key space the **integers** in the range $[0 \dots \mathbf{p}-1]$ (the field \mathbb{Z}_p), the **ECC** uses the **points** $\{x, y\}$ within the Galois field \mathbf{p} (where x and y are integers in the range $[0 \dots \mathbf{p}-1]$).

An **elliptic curve over the finite field \mathbf{p}** consists of:

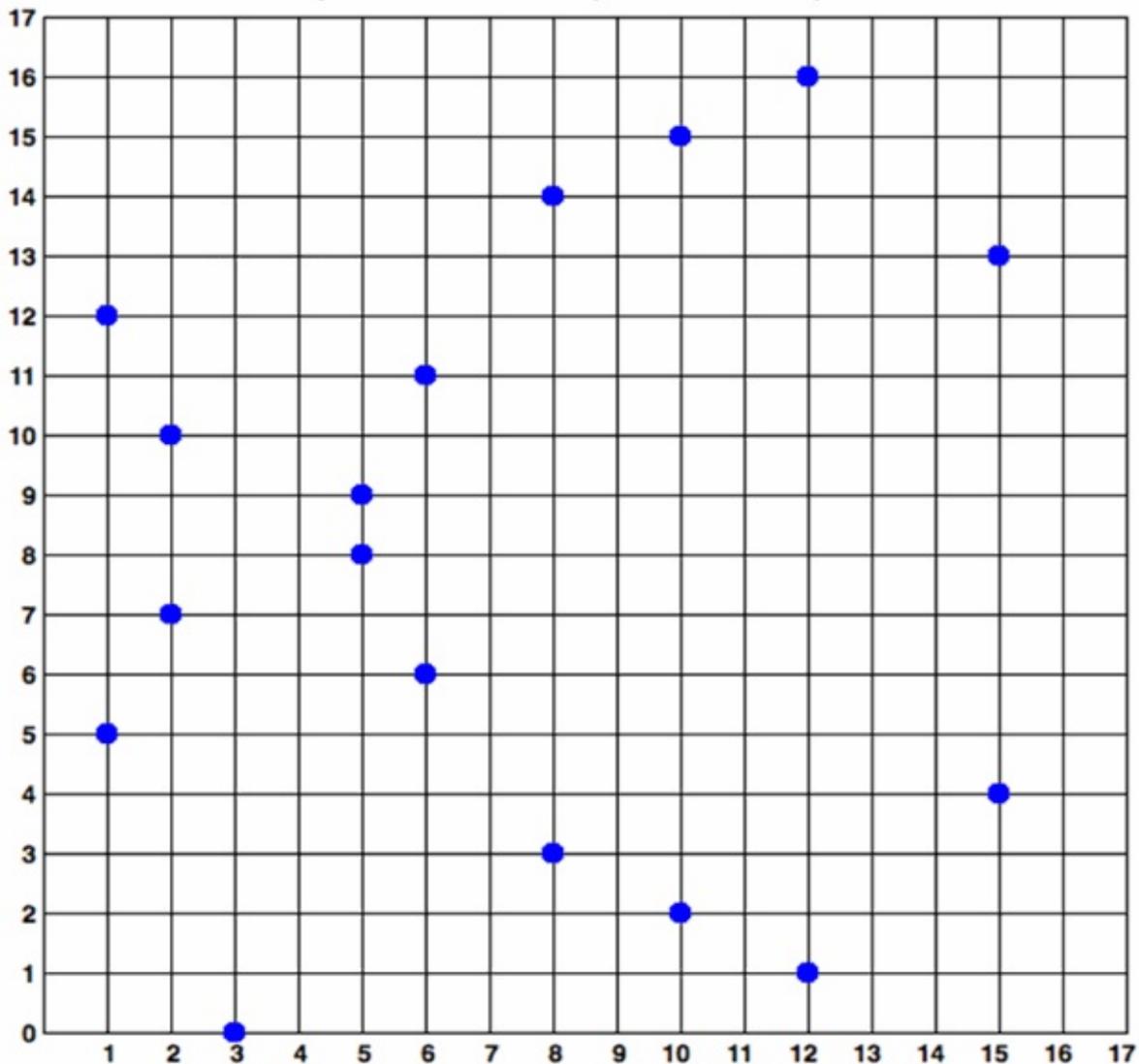
- a set of integer coordinates $\{x, y\}$, such that $0 \leq x, y < \mathbf{p}$
- staying on the elliptic curve: $y^2 \equiv x^3 + ax + b \pmod{\mathbf{p}}$

Example of elliptic curve over the finite field **17**:

- $y^2 \equiv x^3 + 7 \pmod{17}$

This elliptic curve over **17** looks like this:

$$y^2 \equiv x^3 + 7 \pmod{17}$$



Note that the elliptic curve over finite field $y^2 \equiv x^3 + 7 \pmod{17}$ consists of the **blue points** at the above figure, i.e. in practice the "elliptic curves" used in cryptography are "sets of points in square matrix", not classical "curves".

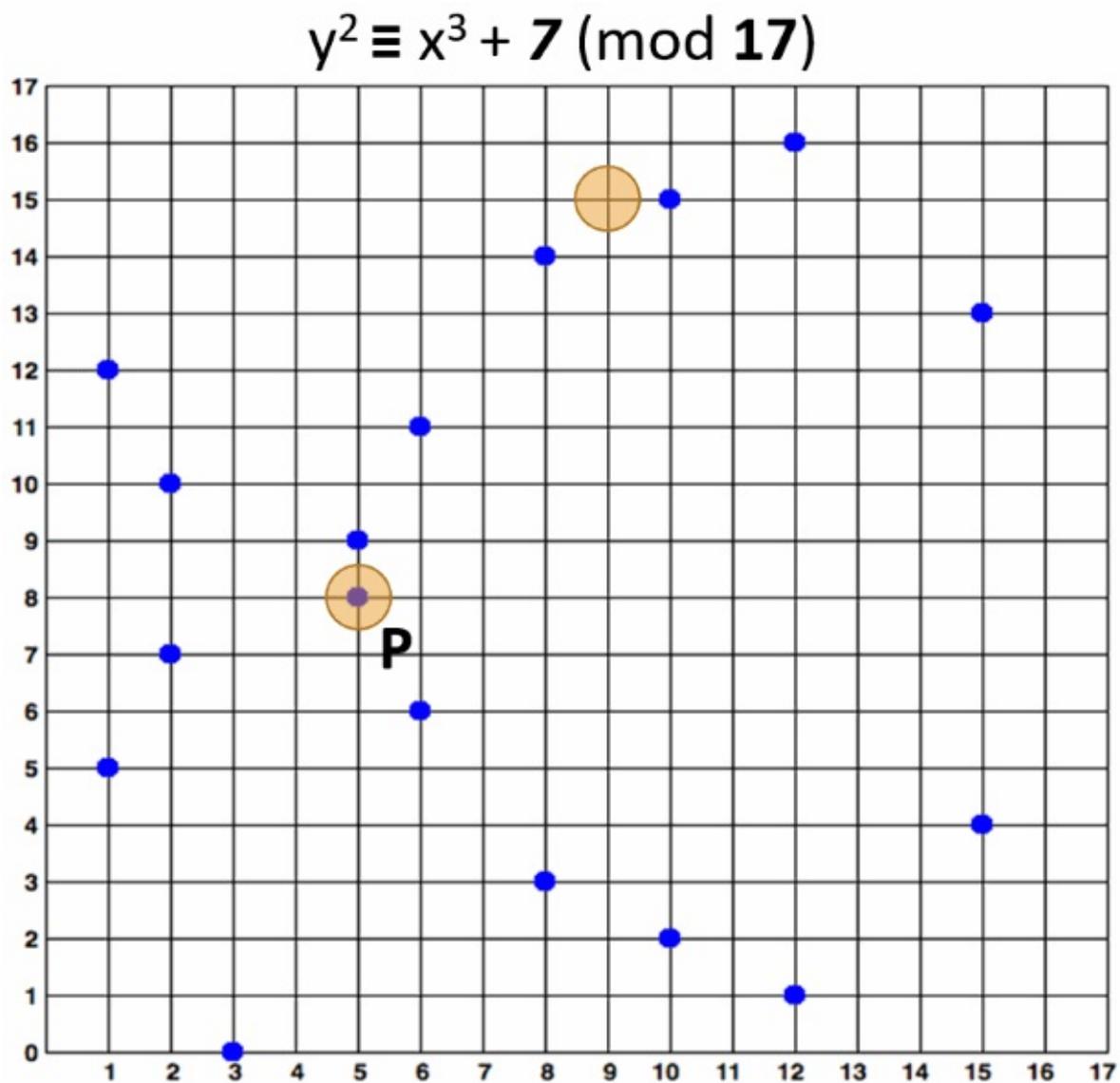
The above curve is "educational". It provides **very small key length** (4-5 bits). In the real world developers typically use curves of 256-bits or more.

Elliptic Curves over Finite Fields: Calculations

It is pretty easy to calculate whether **certain point belongs to certain elliptic curve** over a finite field. For example, a point $\{x, y\}$ belongs to the curve $y^2 \equiv x^3 + 7 \pmod{17}$ when and only when:

- $x^3 + 7 - y^2 \equiv 0 \pmod{17}$

The point **P {5, 8} belongs** to the curve, because `(5**3 + 7 - 8**2) % 17 == 0`. The point **{9, 15} does not belong** to the curve, because `(9**3 + 7 - 15**2) % 17 != 0`. These calculations are in Python style. The above mentioned elliptic curve and the points **{5, 8}** and **{9, 15}** are visualized below:



Multiplying ECC Point by Integer

Two points over an elliptic curve (EC points) can be **added** and the result is another point. This operation is known as **EC point addition**. If we add a point **G** to itself, the result is $G + G = 2 * G$. If we add **G** again to the result, we will obtain $3 * G$ and so on. This is how **EC point multiplication** is defined.

A point **G** over an elliptic curve over finite field (EC point) can be **multiplied by an integer k** and the result is another EC point **P** on the same curve and this operation is **fast**:

- $P = k * G$

The above operation involves some formulas and transformations, but for simplicity, we shall skip them. The important thing to know is that **multiplying EC point by integer returns another EC point** on the same curve and this operation is **fast**. Multiplying an EC point by 0 returns a special EC point called "*infinity*".

Everyone is free to [read more about EC point multiplication](#) in Wikipedia.

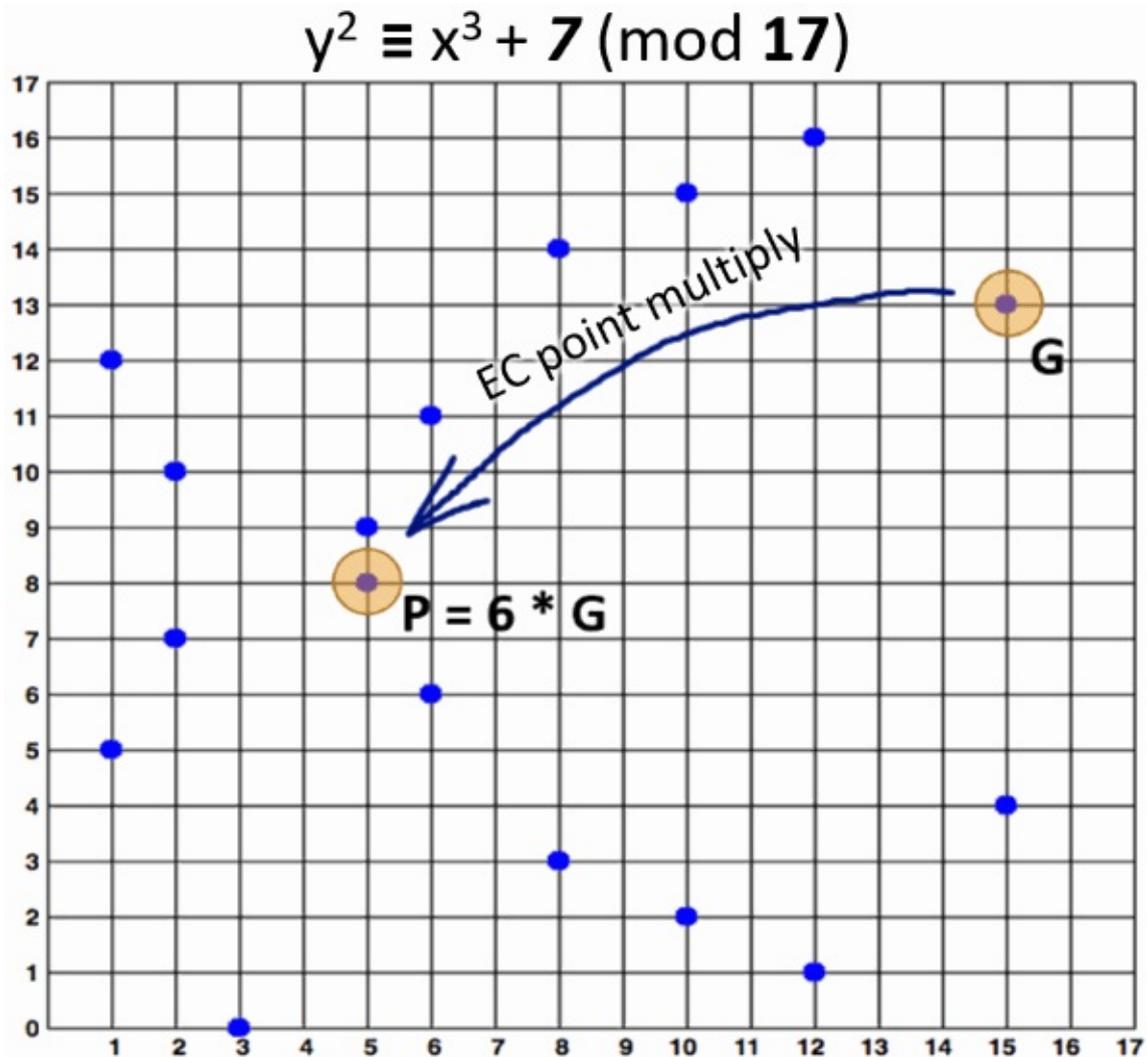
Example: Multiply EC Point by Integer

The formulas for EC multiplication differ for the different forms of **representation of the curve**. In this example, we shall use an elliptic curve in the classical **Weierstrass form**.

For **example** let's take the EC point $G = \{15, 13\}$ on the elliptic curve over finite field $y^2 \equiv x^3 + 7 \pmod{17}$ and multiply it by $k = 6$. We shall obtain an EC point $P = \{5, 8\}$:

- $P = k * G = 6 * \{15, 13\} = \{5, 8\}$

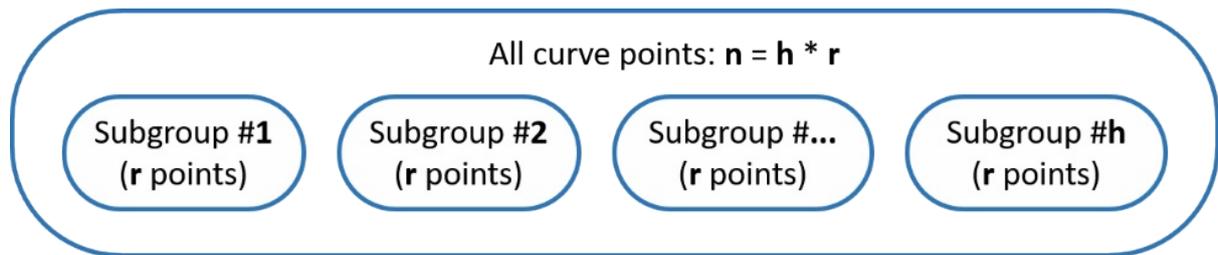
The below figure visualizes this example of EC point multiplication:



Order and Cofactor of Elliptic Curve

An elliptic curve over a finite field can form a finite **cyclic algebraic group**, which consists of all the points on the curve. In a cyclic group, if two EC points are added or an EC point is multiplied to an integer, the result is another EC point from the same cyclic group (and on the same curve). The **order of the curve** is the **total number of all EC points** on the curve. This total number of points includes also the special point called "**point at infinity**", which is obtained when a point is multiplied by 0.

Some curves form a single **cyclic group** (holding all their EC points), while others form several non-overlapping **cyclic subgroups** (each holding a subset of the curve's EC points). In the second scenario the points on the curve are split into h cyclic subgroups (partitions), each of order r (each subgroup holds equal number of points). The **order** of entire group is $n = h * r$ (the number of subgroups, multiplied by the number of points in each subgroup). The number of subgroups h holding the EC points is called **cofactor**.



The **cofactor** is typically expressed by the following formula:

- $h = n / r$

where

- **n** is the **order of the curve** (the number of all its points)
- **h** is the curve **cofactor** (the number of non-overlapping **subgroups** of points, which together hold all curve points)
- **r** is the **order of the subgroups** (the number of points in each subgroup, including the **infinity** point for each subgroup)

In other words, the points over an elliptic curve stay in one or several non-overlapping subsets, called **cyclic subgroups**. The number of subgroups is called "**cofactor**". The total number of points in all subgroups is called "**order**" of the curve and is usually denoted by **n**. If the curve consists of **only one cyclic subgroup**, its **cofactor h = 1**. If the curve consists of several subgroups, its **cofactor > 1**.

- Example of elliptic curve having **cofactor = 1** is `secp256k1` .
- Example of elliptic curve having **cofactor = 8** is `Curve25519` .
- Example of elliptic curve having **cofactor = 4** is `Curve448` .

The "Generator" Point in ECC

For the elliptic curves over finite fields, the ECC cryptosystems define a special pre-defined (constant) EC point called **generator point G** (**base point**), which can **generate any other point in its subgroup** over the elliptic curve by **multiplying G by some integer** in the range $[0...r]$. The number **r** is called "**order**" of the cyclic subgroup (the total number of all points in the subgroup).

For curves with **cofactor = 1** there is **only one subgroup** and the order **n** of the curve (the total number of different points over the curve, including the **infinity**) is equal to the number **r**.

When **G** and **n** are carefully selected, and the **cofactor = 1**, all possible EC points on the curve (including the special point **infinity**) can be generated from the generator **G** by multiplying it by integer in the range $[1...n]$. This integer **n** is known as "**order of the curve**".

It is important to know that the **order r of the subgroup**, obtained from certain EC generator point **G** (which may be different from the order of the curve) defines the **total number of all possible private keys** for this curve: $r = n / h$ (curve order, divided by the curve cofactor). Cryptographers select carefully the elliptic curve domain parameters (curve equation, generator point, cofactor, etc.) to ensure that **the key space is large enough** for certain cryptographic strength.

To summarize, in the ECC cryptography the EC points, together with the generator point **G** form **cyclic groups** (or **cyclic subgroups**), which means that a number **r** exists ($r > 1$), such that $r * G = 0 * G = \text{infinity}$ and all points in the subgroup can be obtained by multiplying **G** by integer in the range $[1...r]$. The number **r** is called **order of the group** (or **subgroup**).

Elliptic curve subgroups usually have **many generator points**, but cryptographers carefully select one of them, which generates the entire group (or subgroup) and is suitable for performance optimizations in the computations. This is the generator known as "**G**".

It is known that for some curves different generator points generate subgroups of different order. More precisely, if the group order is n , for each prime d dividing n , there is a point Q such that $d * Q = \textit{infinity}$. This means that some points used as generators for the same curve will generate smaller subgroups than others. If the group is small, the security is weak. This is known as "**small-subgroup attacks**". This is the reason why cryptographers usually choose the **subgroup order r** to be a **prime number**.

For elliptic curves with cofactor $h > 1$, different **base points** can generate different **subgroups** of EC points on the curve. By choosing a certain **generator point**, we choose to operate over a certain **subgroup of points** on the curve and most EC point operations and ECC crypto algorithms will work well. Still in some cases, special attention should be given, so it is recommended to use only proven ECC implementations, algorithms and software packages.

Generator Point - Example

At the above example (the EC over finite field $y^2 \equiv x^3 + 7 \pmod{17}$), if we take the point $G = \{15, 13\}$ as **generator**, any other point from the curve can be obtained by multiplying G by some integer in the range $[1..18]$. Thus the **order** of this EC is $n = 18$ and its cofactor $h = 1$.

Note that the curve has 17 normal EC points (shown at the above figures) + one special "**point at infinity**", all staying in a single subgroup, and the curve order is **18** (not 17).

Note also, that if we take the **point $\{5, 9\}$** as **generator**, it will generate **just 3 EC points**: $\{5, 8\}$, $\{5, 9\}$ and **infinity**. Because the curve order is not prime number, different generators may generate subgroups of different order. This is a good example why we should not "invent" our own elliptic curves for cryptographic purposes and we should use proven curves.

Private Key, Public Key and the Generator Point in ECC

In the **ECC**, when we multiply a fixed EC point G (the **generator point**) by certain **integer k** (k can be considered as **private key**), we obtain an EC point P (its corresponding **public key**).

Consequently, in ECC we have:

- **Elliptic curve (EC)** over finite field p
- G == **generator point** (fixed constant, a base point on the EC)
- k == **private key** (integer)
- P == **public key** (point)

It is **very fast** to calculate $P = k * G$, using the well-known **ECC multiplication algorithms** in time $\log_2(k)$, e.g. the "**double-and-add algorithm**". For 256-bit curves, it will take just a few hundreds simple EC operations.

It is **extremely slow** (considered infeasible for large k) to calculate $k = P / G$.

This asymmetry (fast multiplication and infeasible slow opposite operation) is the basis of the security strength behind the ECC cryptography, also known as the **ECDLP problem**.

Elliptic-Curve Discrete Logarithm Problem (ECDLP)

The **Elliptic Curve Discrete Logarithm Problem (ECDLP)** in computer science is defined as follows:

- By given elliptic curve over finite field p and generator point G on the curve and point P on the curve, find the integer k (if it exists), such that $P = k * G$

For carefully chosen (by cryptographers) finite fields and elliptic curves, **the ECDLP problem has no efficient solution**.

The **multiplication** of elliptic curve points in the group p is similar to **exponentiation** of integers in the group Z_p (this is known as *multiplicative notation*) and this is how the **ECDLP problem** is similar to the **DLP problem** (discrete logarithm problem).

In the ECC cryptography, many algorithms rely on the **computational difficulty of the ECDLP problem** over carefully chosen field \mathbf{p} and elliptic curve, for which **no efficient algorithm exists**.

ECC and Curve Security Strength

Because the fastest known algorithm to solve the **ECDLP** for key of size \mathbf{k} needs \sqrt{k} steps, this means that to achieve a \mathbf{k} -bit **security strength**, at least **$2 \cdot k$ -bit curve** is needed. Thus **256-bit elliptic curves** (where the field size \mathbf{p} is 256-bit number) typically provide nearly **128-bit security strength**.

In fact, the strength is **slightly less**, because the **order** of the curve (\mathbf{n}) is typically less than the fields size (\mathbf{p}) and because the curve may have cofactor $\mathbf{h} > 1$ (and subgroup order $\mathbf{r} = \mathbf{n} / \mathbf{h}$, smaller than \mathbf{n}) and because the number of steps is not exactly \sqrt{k} , but is $0.886 * \sqrt{k}$. A precise **security strength** estimation for the most popular **standard elliptic curves** is given here: <http://safecurves.cr.yt/rho.html>.

For example, the `secp256k1` ($\mathbf{p} = 256$) curve provides ~ 128 -bit security (127.8 bits to be precise) and the `Curve448` ($\mathbf{p} = 448$) provides ~ 224 -bit security (222.8 bits to be precise).

Multiplication of EC Points - Example in Python

Now, after all the concepts, let's **write some code**. We shall use the **Python library `tinyec`**, which provides **ECC** primitives, such as **cyclic groups** (the `SubGroup` class), **elliptic curves** over finite fields (the `Curve` class) and **EC points** (the `Point` class). First, install the package `tinyec`:

```
pip install tinyec
```

We shall play with the educational curve from our previous examples $y^2 \equiv x^3 + 7 \pmod{17}$, with the generator point $\mathbf{G} = \{15, 13\}$, which has order of $\mathbf{n} = 18$. We shall name it `p1707`.

```
from tinyec.ec import SubGroup, Curve

field = SubGroup(p=17, g=(15, 13), n=18, h=1)
curve = Curve(a=0, b=7, field=field, name='p1707')
print('curve:', curve)

for k in range(0, 25):
    p = k * curve.g
    print(f"{k} * G = ({p.x}, {p.y})")
```

The above code demonstrates the **EC multiplication**. It multiplies the generator point \mathbf{G} by 0, 1, 2, ..., 24. The output from the above program is as follows:

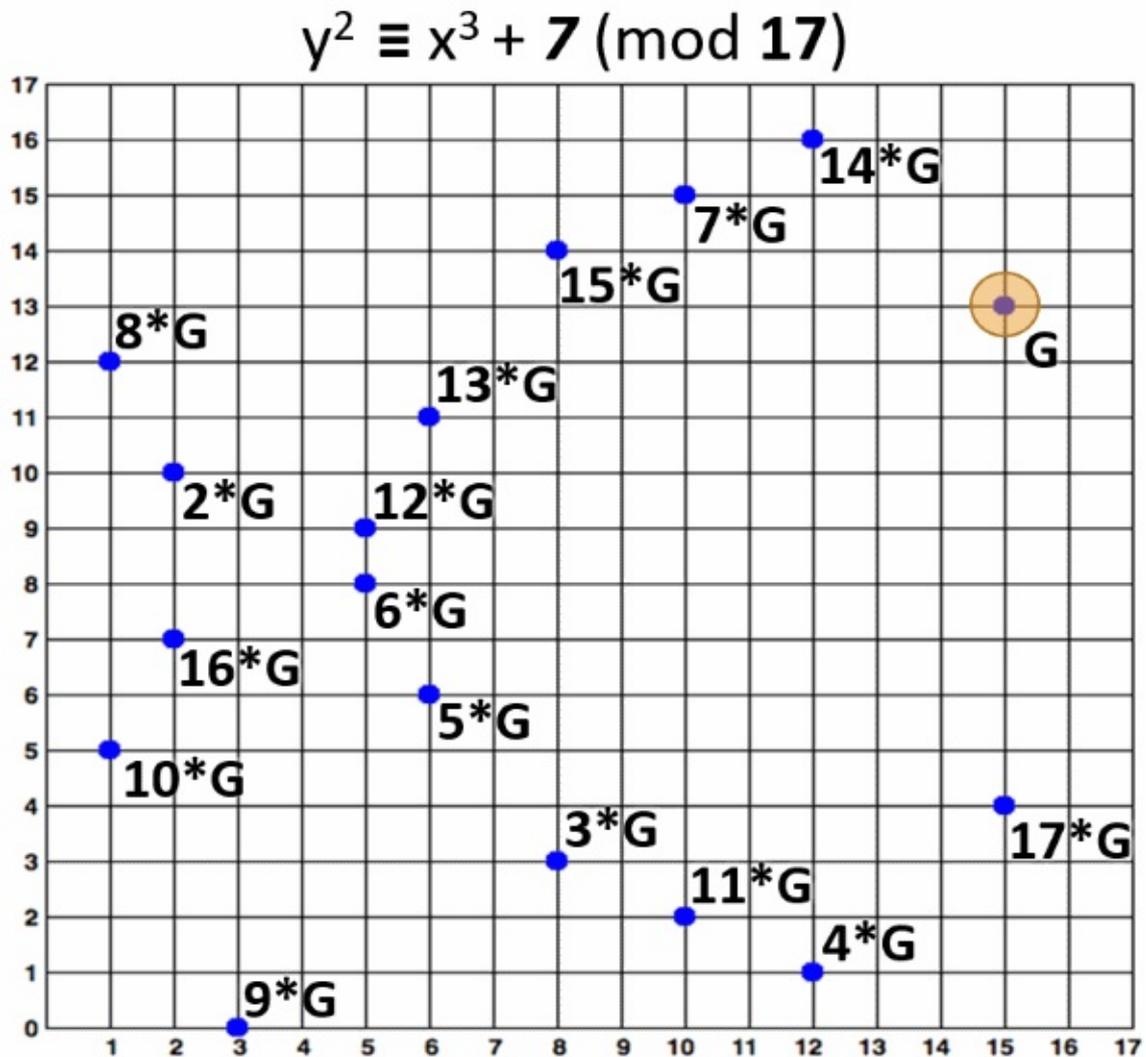
```
curve: "p1707" => y^2 = x^3 + 0x + 7 (mod 17)
0 * G = (None, None)
1 * G = (15, 13)
2 * G = (2, 10)
3 * G = (8, 3)
4 * G = (12, 1)
5 * G = (6, 6)
6 * G = (5, 8)
7 * G = (10, 15)
8 * G = (1, 12)
9 * G = (3, 0)
10 * G = (1, 5)
11 * G = (10, 2)
12 * G = (5, 9)
```

```
13 * G = (6, 11)
14 * G = (12, 16)
15 * G = (8, 14)
16 * G = (2, 7)
17 * G = (15, 4)
18 * G = (None, None)
19 * G = (15, 13)
20 * G = (2, 10)
21 * G = (8, 3)
22 * G = (12, 1)
23 * G = (6, 6)
24 * G = (5, 8)
```

It is visible that $0 * G = \textit{infinity}$. It is also clearly visible, that the EC group is **cyclic** and the order of the EC group is **n = 18**, because starting from **k = 18**, the next points repeat the first ones:

- $18 * G = 0 * G = \textit{infinity}$
- $19 * G = 1 * G = \{15, 13\}$
- $20 * G = 2 * G = \{2, 10\}$
- $21 * G = 3 * G = \{8, 3\}$
- etc.

The EC points, generated by multiplying the generator point **G** by 2, 3, 4, ..., 17 are shown on the figure below:



Let's modify a bit the above example and change the generator point to be $G' = \{5, 9\}$. This will change significantly the output:

```
from tinyec.ec import SubGroup, Curve

field = SubGroup(p=17, g=(5, 9), n=18, h=1)
curve = Curve(a=0, b=7, field=field, name='p1707')
print('curve:', curve)

for k in range(0, 25):
    p = k * curve.g
    print(f"{k} * G' = ({p.x}, {p.y})")
```

The output shows that the subgroup order of the new generator point is not 18, but is 3. This is possible, because 18 is **not prime**. It is clear from the output, that $3 * G' = \textit{infinity}$ and the obtained subgroup order is 3:

```
curve: "p1707" => y^2 = x^3 + 0x + 7 (mod 17)
0 * G' = (None, None)
1 * G' = (5, 9)
2 * G' = (5, 8)
3 * G' = (None, None)
4 * G' = (5, 9)
```

```

5 * G' = (5, 8)
6 * G' = (None, None)
...

```

The above example again confirms that designing an elliptic curve for cryptography should be done by cryptographers, not by developers. Developers should rely on well established crypto-standards and proven crypto-libraries.

Multiplication of EC Points - Real-World Example in Python

Now, let's write a **real-world example**. Instead of using our educational curve `p1707` (4-5-bit curve, $p = 17$), we shall use the 192-bit cryptographic curve `secp192r1` (192-bit, $p = 6277101735386680763835789423207666416083908700390324961279$). The below example is similar to the previous:

```

from tinyec import registry

curve = registry.get_curve('secp192r1')
print('curve:', curve)

for k in range(0, 10):
    p = k * curve.g
    print(f"{k} * G = ({p.x}, {p.y})")

print("Cofactor =", curve.field.h)

print('Cyclic group order =', curve.field.n)

nG = curve.field.n * curve.g
print(f"n * G = ({nG.x}, {nG.y})")

```

The output is also similar to the previous example:

```

curve: "secp192r1" => y^2 = x^3 + 6277101735386680763835789423207666416083908700390324961276
x + 2455155546008943817740293915197451784769108058161191238065 (mod 627710173538668076383578
9423207666416083908700390324961279)
0 * G = (None, None)
1 * G = (602046282375688656758213480587526111916698976636884684818, 174050332293622031404857
552280219410364023488927386650641)
2 * G = (5369744403678710563432458361254544170966096384586764429448, 54292343797890710397506
54906915254128254326554272718558123)
3 * G = (2915109630280678890720206779706963455590627465886103135194, 29466267115587920039806
54088990112021985937607003425539581)
4 * G = (1305994880430903997305943738697779408316929565234787837114, 39818639774511503421169
87835776121688410789618551673306674)
5 * G = (410283251116784874018993562136566870110676706936762660240, 120665467489982524668820
5669651974202006189255452737318561)
6 * G = (4008504146453526025173196900303594155799995627910231899946, 32637593013051769069908
06636587838100022690095020155627760)
7 * G = (3473339081378406123852871299395262476289672479707038350589, 21527131769066036042008
42901176476029776544337891569565621)
8 * G = (1167950611014894512313033362696697441497340081390841490910, 40021779061112151271484
83369584652296488769677804145538752)
9 * G = (3176317450453705650283775811228493626776489433309636475023, 44601893774669384766793

```

```

803854980115179612118075017062201)
Cofactor = 1
Cyclic group order = 6277101735386680763835789423176059013767194773182842284081
n * G = (None, None)

```

The curve `secp192r1` uses a **cyclic group** of very large order $n = 6277101735386680763835789423176059013767194773182842284081$ (prime number) with **cofactor** $h = 1$, and as we can expect, $n * G = \textit{infinity}$, just like at the previous example with our educational curve.

Now, let's generate a random **private key** `privKey` (integer in the range $[0..n-1]$) and its corresponding **public key** `pubKey = privKey * G`:

```

from tinyec import registry
import secrets

curve = registry.get_curve('secp192r1')

privKey = secrets.randbelow(curve.field.n)
pubKey = privKey * curve.g
print("private key:", privKey)
print("public key:", pubKey)

```

The above code will produce output like this:

```

private key: 4225655318977962031264230130242180748818603147467615868902
public key: (5396030834456770190396776530938374882273836179487834152291, 3422160588166914010
077732710830109086004758012634997793937) on "secp192r1" => y^2 = x^3 + 627710173538668076383
5789423207666416083908700390324961276x + 245515554600894381774029391519745178476910805816119
1238065 (mod 6277101735386680763835789423207666416083908700390324961279)

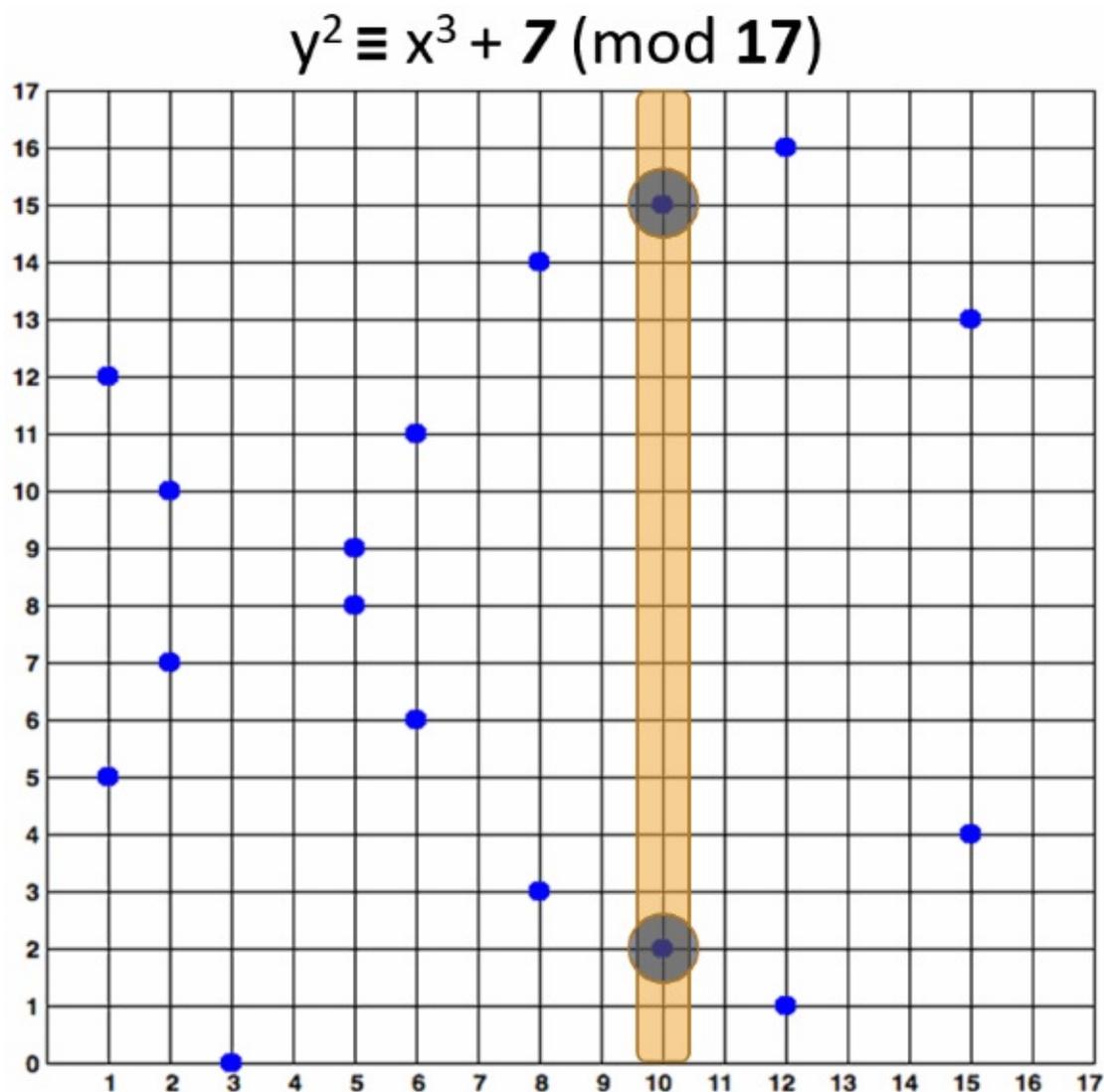
```

Later we shall use such **pairs of ECC keys** {private key, public key} to encrypt data, sign messages and verify signatures.

Note that in real projects, **192-bit curves are considered weak**, so 256-bit curves are recommended (or more bits), where the keys are also 256-bits (or respectively more). We use 192-bit curve in the above example just to make the sample output smaller.

Public Key Compression in the Elliptic Key Cryptosystems

Elliptic curves over finite fields p (in the Weierstrass form) have **at most 2 points per y coordinate** (odd x and even x). This property comes from the nature of the elliptic curve equation and is illustrated at the below graph:



Due to this property, an elliptic curve point (and respectively an ECC public key) $P \{x, y\}$ can be **compressed** as $C \{x, \text{odd/even}\}$. This means to erase the y coordinate from the point and represent it as 1 bit (odd y or even y).

Compressed EC point is an EC point $\{x, y\}$ represented in its shorter form $\{x, \text{odd / even}\}$. ECC **public keys** are EC points, so they can also be compressed in the same way.

To **decompress a point**, we can calculate its two possible y coordinates by the formulas:

- $y_1 = \text{mod_sqrt}(x^3 + ax + b, p)$
- $y_2 = p - \text{mod_sqrt}(x^3 + ax + b, p)$

Then we take the **odd** or **even** from the above coordinates (according to the additional parity bit in the compressed representation).

The **modular square root** (`mod_sqrt`) can be calculated using the [Tonelli–Shanks algorithm](#).

Let's take an **example**: at the elliptic curve $y^2 \equiv x^3 + 7 \pmod{17}$ the point $P \{10, 15\}$ can be **compressed** as $C \{10, \text{odd}\}$. For **decompression**, we first calculate the two possible y coordinates for $x = 10$ using the above formulas: $y_1 = 2$ and $y_2 = 15$. Then we choose the **odd** one: $y = 15$. The decompressed point is $\{10, 15\}$.

Compressing a EC Point / Public Key - Example in Python

The code below implements **public key compression** and **decompression** in Python. It uses a library called `nummaster` for the "**modular square root**" function, which is unavailable in Python. First install the `nummaster` package:

```
pip install nummaster
```

Now implement the EC point **compression** and **decompression** functions in Python:

```
from nummaster.basic import sqrtmod

def compress_point(point):
    return (point[0], point[1] % 2)

def uncompress_point(compressed_point, p, a, b):
    x, is_odd = compressed_point
    y = sqrtmod(pow(x, 3, p) + a * x + b, p)
    if bool(is_odd) == bool(y & 1):
        return (x, y)
    return (x, p - y)
```

Finally, compress and decompress the point $\{10, 15\}$ on the curve $y^2 \equiv x^3 + 7 \pmod{17}$, just as an example:

```
p, a, b = 17, 0, 7
point = (10, 15)
print(f"original point = {point}")
compressed_p = compress_point(point)
print(f"compressed = {compressed_p}")
restored_p = uncompress_point(compressed_p, p, a, b)
print(f"uncompressed = {restored_p}")
```

The output of the above code is:

```
original point = (10, 15)
compressed = (10, 1)
uncompressed = (10, 15)
```

Elliptic Curve Domain Parameters for ECC

ECC elliptic curves are described by a set of elliptic curve **domain parameters**, such as the curve equation parameters, the field parameters and the generator point coordinates. These parameters are specified in **cryptography standards**, such as:

- [SEC 2: Recommended Elliptic Curve Domain Parameters](#)
- [NIST FIPS PUB 186-4 Digital Signature Standard \(DSS\)](#)
- [Brainpool ECC Standard \(RFC-5639\)](#)

These standards define the parameters for a set of **named curves**, such as `secp256k1`, `P-521` and `brainpool1P512t1`. The elliptic curves over finite fields, described in these crypto standards are well researched and analysed by cryptographers and are considered to have certain **security strength**, also described in these standards.

Some cryptographers (like [Daniel Bernstein](#)) believe that most of the curves, described in the official crypto-standards are "**unsafe**" and define their own **crypto-standards**, which consider the **ECC security** in much broader level.

The Bernstein's [SafeCurves](#) standard lists the curves, which are **safe** according to a set of ECC security requirements. The standard is available at <https://safecurves.cr.yp.to>.

Choosing an Elliptic Curve for ECC

To use ECC all communicating parties should agree on the EC domain parameters (all the elements defining the elliptic curve). It is highly **recommended** to use a **named curve** from the above standards with **at least 256-bit** modulus. Standard curves are well studied by cryptographers to guarantee their security strength.

Don't use own elliptic curve (with non-standard domain parameters), unless you are experienced cryptographer and you know very well what are you doing! Many curves have **weaknesses**, which make the ECDLP problem not so difficult and compromise the security. If you are afraid of backdoored curves, use a standard safe curve from the [SafeCurves list](#).

Named Curves - Example

In ECC cryptography, **elliptic curves over the finite fields** are used, where the modulus p and the order n are very large integers (n is usually prime number), e.g. **256-bit** number. The **finite field** of the curve is of square form of size $p \times p$, which is incredibly large, and all possible EC points on the curve (the order of the curve n) is also a very big integer, e.g. 256-bit. For example, the **domain parameters** for the curve **secp256k1** (the Bitcoin curve) are defined as follows:

- p (modulus) = `0xFFC2F`
- n (order; size; the count of all possible EC points) =
`0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141`
- a (the constant "a" in $y^2 \equiv x^3 + a*x + b \pmod{p}$) =
`0x00`
- b (the constant "b" in $y^2 \equiv x^3 + a*x + b \pmod{p}$) =
`0x0007`
- g (the curve generator point $G \{x, y\}$) =
(`0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798` ,
`0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8`)
- h (cofactor, typically 1) = 01

We already know that a 256-bit curve (which means that p and n are 256-bit numbers) provides 128-bit **security strength**, which means that to find the private key from the public key or signature, the best known non-quantum algorithm will take approximately 2^{128} operations. The above-defined ECC curve **secp256k1** has 128-bit strength.

Python Examples with the "secp256k1" Curve

Now let's put in action the above domain parameters for the **secp256k1** curve. Let's define the EC and calculate the public key for certain private key:

```
from tinyec.ec import SubGroup, Curve

# Domain parameters for the `secp256k1` curve
# (as defined in http://www.secg.org/sec2-v2.pdf)
name = 'secp256k1'
p = 0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f
n = 0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141
a = 0x0000000000000000000000000000000000000000000000000000000000000000
b = 0x0000000000000000000000000000000000000000000000000000000000000007
g = (0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798,
     0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8)
h = 1
curve = Curve(a, b, SubGroup(p, g, n, h), name)
print('curve:', curve)
```

```

privKey = int('0x51897b64e85c3f714bba707e867914295a1377a7463a9dae8ea6a8b914246319', 16)
print('privKey:', hex(privKey)[2:])

pubKey = curve.g * privKey
pubKeyCompressed = '0' + str(2 + pubKey.y % 2) + str(hex(pubKey.x)[2:])
print('pubKey:', pubKeyCompressed)

```

The above code defines the `secp256k1` curve through its domain parameters and calculates a **public key** by given **private key**. This is done by multiplying the curve generator **G** by the private key. The result is correct, like it is visible from the program output:

```

curve: "secp256k1" => y^2 = x^3 + 0x + 7 (mod 1157920892373161954235709850086879078532699846
65640564039457584007908834671663)
privKey: 51897b64e85c3f714bba707e867914295a1377a7463a9dae8ea6a8b914246319
pubKey: 02f54ba86dc1ccb5bed0224d23f01ed87e4a443c47fc690d7797a13d41d2340e1a

```

The public key is compressed and encoded in the standard format (encode the **y** coordinate as prefix `02` or `03`).

Edwards Curves

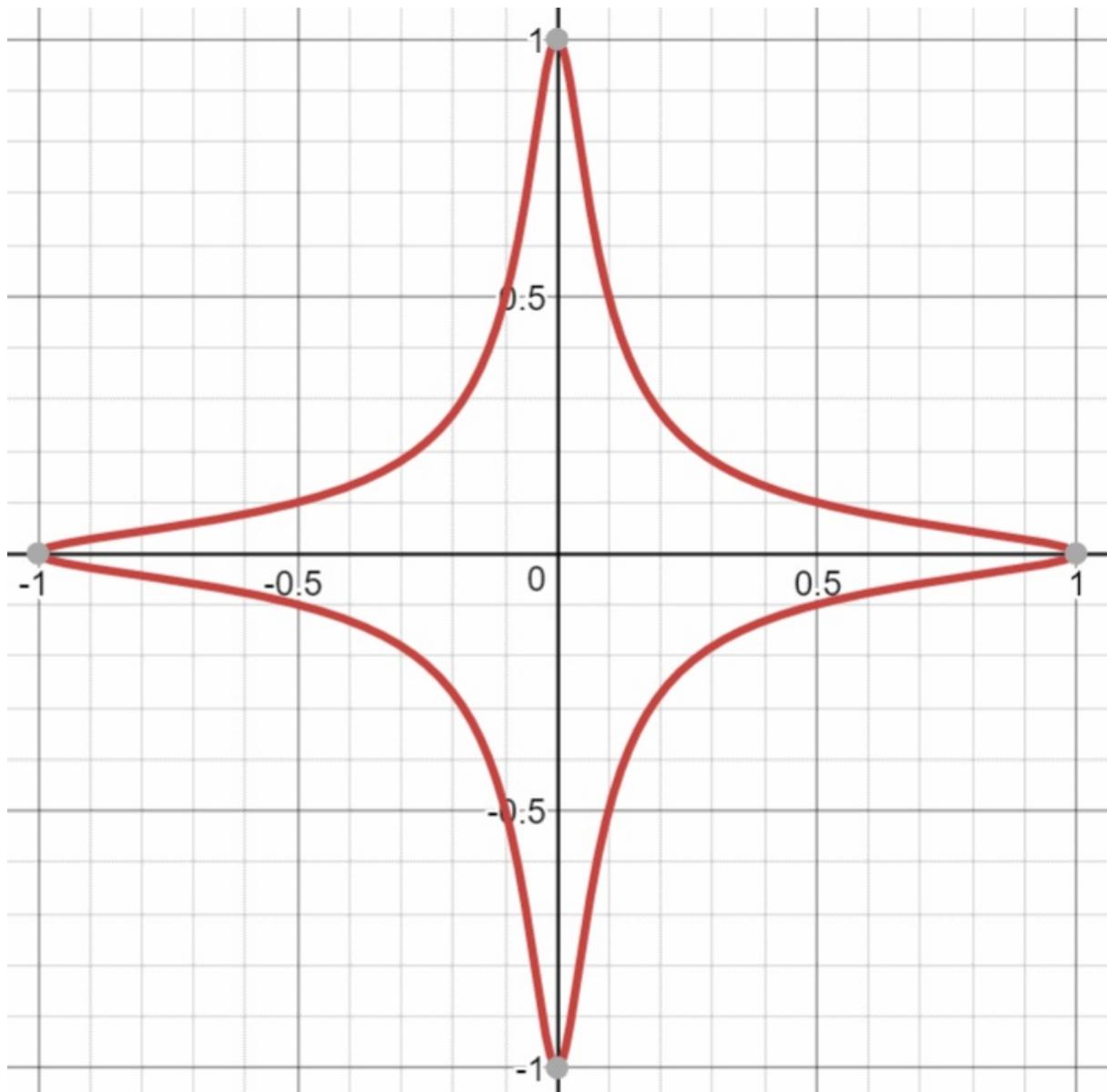
Elliptic curves in the elliptic curve cryptography (ECC) may be presented in several **forms** (representations), which are proven to be birationally equivalent (isomorphic):

- **Weierstrass form** of elliptic curve:
 - $y^2 = x^3 + ax + b$
 - Example Weierstrass curve used in ECC is `secp256k1`, which has the form $y^2 = x^3 + 7$
- **Montgomery form** of elliptic curve:
 - $By^2 = x^3 + Ax^2 + x$
 - Example Montgomery curve used in ECC is `Curve25519`, which has the form $y^2 = x^3 + 486662x^2 + x$
- **Edwards form** of elliptic curve:
 - $x^2 + y^2 = 1 + dx^2y^2$
 - Example Edwards curve used in ECC is `Curve448`, which has the form $x^2 + y^2 = 1 - 39081x^2y^2$

For performance reasons elliptic curve cryptography (ECC) sometimes uses **Edwards curves**, which are elliptic curves in the following form:

- $x^2 + y^2 = 1 + dx^2y^2$

For example, if $d = 300$, the Edwards curve $x^2 + y^2 = 1 + 300x^2y^2$ looks like this:



Every **Edwards curve** is birationally equivalent to an **elliptic curve in Weierstrass form** ($y^2 = x^3 + ax + b$) and thus has the same properties like the classical elliptic curves.

Edwards curves over a finite prime field \mathbf{p} (where \mathbf{p} is large prime number) provide fast integer to EC point multiplication, which has similar cryptographic properties like the classical elliptic curves, and the **ECDLP** problem has the same computational difficulty, suitable for cryptographic purposes.

Examples of well-known cryptographic elliptic **Edwards curves** over finite prime fields are: **Curve1174** (251-bit), **Curve25519** (255-bit), **Curve383187** (383-bit), **Curve41417** (414-bit), **Curve448** (448-bit), **E-521** (521-bit) and others.

Curve25519, X25519 and Ed25519

With carefully selected curve parameters, the **Edwards curves over finite fields** can implement ECC cryptosystems capable to provide ECDH **key agreement** schemes, **digital signatures** and **hybrid encryption** schemes, with very **high performance**.

For example, the [Curve25519](#) is the **Edwards curve**, defined by the following elliptic curve equation in **Montgomery form**:

- $y^2 = x^3 + 486662x^2 + x$

over the finite prime field \mathbf{p} , where $\mathbf{p} = 2^{255} - 19$ (the curve is 255-bit).

In fact, the above equation does not match directly the Edwards curve equation, but it is proven to be birationally equivalent to the following **twisted Edwards curve** (known as **edwards25519**):

- $-x^2 + y^2 = 1 + 3709570593466943934313808350875456518954211387984321901638878553308594028355x^2y^2$

The elliptic curve **Curve25519** consists of all points $\{\mathbf{x}, \mathbf{y}\}$ with integer coordinates, defined by the modular equation:

- $y^2 \equiv x^3 + 486662x^2 + x \pmod{2^{255} - 19}$

The above equation has its equivalent in the classical **Weierstrass form** for the elliptic curves ($y^2 = x^3 + ax + b$), but the above form is designed especially for speed optimizations.

The **Curve25519** is carefully engineered, by a team of cryptographers, led by Daniel Bernstein, at several levels of design and implementation to achieve **very high speeds** without compromising security.

The **Curve25519** has **order** (in its underlying cyclic group) $\mathbf{n} = 2^{252} + 0x14def9dea2f79cd65812631a5cf5d3ed$ and **cofactor** $\mathbf{h} = 8$ and provides **125.8-bit security strength** (it is sometimes referred as ~ 128 -bit security). The **private keys** for the Curve25519 are 251 bits and are usually encoded as **256-bit integers** (32 bytes, 64 hex digits). The **public keys** are typically encoded also as **256-bit integers** (255-bit y-coordinate + 1-bit x-coordinate) and this is very convenient for developers.

Based on the **Curve25519** an **ECDH function** is derived, called **X25519** (used for elliptic-key Diffie–Hellman key agreement schemes) and fast **digital signature scheme** is derived, called **Ed25519**, based on the the **EdDSA** algorithm. These schemes are **very fast**, because they involve multiplications and other simple operations with small integers (mostly 32-bit arithmetic), which can be efficiently implemented in the modern microprocessors (CPUs). Note that X25519 and Ed25519 use **different encodings for the EC points**, so they are not directly compatible and require conversion if you want to use the same public-private key pairs.

Curve448, X448 and Ed448

The **Curve448** (**Curve448-Goldilocks**) is an untwisted **Edwards curve**, defined by the equation:

- $x^2 + y^2 = 1 - 39081x^2y^2$

over the finite prime field \mathbf{p} , where $\mathbf{p} = 2^{448} - 2^{224} - 1$. It has **order** of $\mathbf{n} = 2^{446} -$

$0x8335dc163bb124b65129c96fde933d8d723a70aac873d6d54a7bb0d$ and **cofactor** $\mathbf{h} = 4$. Like any other Edwards curve, the **Curve448** has equivalent in the **Weierstrass form** ($y^2 = x^3 + ax + b$), but the above Edwards form provides significant optimizations in the EC point calculations and improved performance.

The **Curve448** provides ~ 224 -bit security level (more precisely **222.8-bits**). The **private keys** for the Curve448 are 446 bits and are typically encoded as **448-bit integers** (56 bytes, 112 hex digits). The **public keys** are also encoded as **448-bit integers**.

The **Curve448** is suitable for **ECDH key agreement** (ECDH function, known as **X448**) and for fast **digital signatures** (EdDSA algorithm, known as **Ed448** or **edwards448**). Note that X448 and Ed448 use **different encodings for the EC points**, so they are not directly compatible and require conversion if you want to use the same public-private key pairs.

Curve25519 or Curve448?

Prefer **Curve448** to **Curve25519** when your application needs a **higher level of security**, but have in mind that **Curve448** is about 3 times **slower** than **Curve25519** and uses longer key length and signature length.

Prefer **Curve25519** to **Curve448** when you need better performance and smaller keys and signatures.

Learn more about the **Curve25519** and **Curve448** from the technical perspective from:

- [RFC 7748 - Elliptic Curves for Security](#) - the Internet technical standard for implementing the **X25519** and **X448** key exchange protocols.
- [RFC 8032 - Edwards-Curve Digital Signature Algorithm \(EdDSA\)](#) - the Internet technical standard for implementing the the **Ed25519** and **EdDSA-Ed448** signature schemes.

In the general case, have in mind that **Curve25519 is faster** than **secp256k1** and the other 256-bit standard NIST curves and is considered more secure, so it is the recommended choice for ~ 128-bit security. Similarly, the **Curve448** has better performance than the classical curves with similar key length, so it is the recommended curve for ~ 224-bit security.

Curve25519 - Example in Python

To demonstrate the elliptic curve **Curve25519** in practice, we shall first install the [pynacl](#) crypto library for Python:

```
pip install pynacl
```

The [Python binding to the Networking and Cryptography \(NaCl\) library \(PyNaCl\)](#) implements many modern cryptographic algorithms, including the EC point arithmetic over the **Curve25519** and **Ed25519** signatures.

Next, generate a random 252-bit **private key** and its corresponding **public key** (EC point) on the **Curve25519** (both keys will be encoded internally as **256-bit integers**):

```
from nacl.public import PrivateKey
import binascii

privKey = PrivateKey.generate()
pubKey = privKey.public_key

print("privKey:", binascii.hexlify(bytes(privKey)))
print("pubKey: ", binascii.hexlify(bytes(pubKey)))
```

The sample output from the above code shows that both the **public** and the **private** (secret) keys on the **Curve25519** are encoded as **256-bit integers** (64 hex digits, 32 bytes) and this simplifies the developers:

```
privKey: b'8175f7cd524a59b6efbd447985ce5d97c546b319521ff236203970e50052c641'
pubKey:  b'cf97a96568fee4ddb232f617fd5b9df2d2e5b90e68ba7f6d5129ea92d7d8f95e'
```

In fact, different crypto libraries may use different **key encodings** and typically X25519 ECDH keys are encoded differently than Ed25519 keys (Montgomery curve coordinates vs. twisted Edwards curve coordinates).

ECDH Key Exchange (Elliptic Curve Diffie–Hellman Key Exchange)

The **ECDH** (Elliptic Curve Diffie–Hellman Key Exchange) is an **anonymous key agreement scheme**, which allows two parties, each having an elliptic-curve public–private key pair, to establish a **shared secret** over an insecure channel. **ECDH** is very similar to the classical **DHKE** (Diffie–Hellman Key Exchange) algorithm, but it uses **ECC point multiplication** instead of **modular exponentiations**. ECDH is based on the following property of EC points:

- $(a * G) * b = (b * G) * a$

If we have two **secret numbers** **a** and **b** (two **private keys**, belonging to Alice and Bob) and an ECC elliptic curve with generator point **G**, we can exchange over an insecure channel the values **(a * G)** and **(b * G)** (the **public keys** of Alice and Bob) and then we can derive a shared secret: **secret = (a * G) * b = (b * G) * a**. Pretty simple. The above equation takes the following form:

- $alicePubKey * bobPrivKey = bobPubKey * alicePrivKey = secret$

The **ECDH** algorithm (Elliptic Curve Diffie–Hellman Key Exchange) is trivial:

1. **Alice** generates a **random** ECC key pair: {**alicePrivKey**, **alicePubKey = alicePrivKey * G**}
2. **Bob** generates a **random** ECC key pair: {**bobPrivKey**, **bobPubKey = bobPrivKey * G**}
3. Alice and Bob **exchange their public keys** through the insecure channel (e.g. over Internet)
4. **Alice** calculates **sharedKey = bobPubKey * alicePrivKey**
5. **Bob** calculates **sharedKey = alicePubKey * bobPrivKey**
6. Now both **Alice** and **Bob** have the same **sharedKey == bobPubKey * alicePrivKey == alicePubKey * bobPrivKey**

In the next section, we shall implement the ECDH algorithm and demonstrate it with code example.

ECDH Key Exchange - Examples in Python

Now let's implement the **ECDH** algorithm (Elliptic Curve Diffie–Hellman Key Exchange) in Python.

We shall use the `tinyec` library for ECC in Python:

```
pip install tinyec
```

Now, let's generate two public-private **key pairs**, exchange the **public keys** and calculate the **shared secret**:

```
from tinyec import registry
import secrets

def compress(pubKey):
    return hex(pubKey.x) + hex(pubKey.y % 2)[2:]

curve = registry.get_curve('brainpoolP256r1')

alicePrivKey = secrets.randbelow(curve.field.n)
alicePubKey = alicePrivKey * curve.g
print("Alice public key:", compress(alicePubKey))

bobPrivKey = secrets.randbelow(curve.field.n)
bobPubKey = bobPrivKey * curve.g
print("Bob public key:", compress(bobPubKey))

print("Now exchange the public keys (e.g. through Internet)")

aliceSharedKey = alicePrivKey * bobPubKey
print("Alice shared key:", compress(aliceSharedKey))

bobSharedKey = bobPrivKey * alicePubKey
print("Bob shared key:", compress(bobSharedKey))

print("Equal shared keys:", aliceSharedKey == bobSharedKey)
```

The **elliptic curve** used for the ECDH calculations is **256-bit** named curve `brainpoolP256r1`. The **private keys** are **256-bit** (64 hex digits) and are generated randomly. The **public keys** will be **257 bits** (65 hex digits), due to **key compression**.

The **output** of the above code looks like this:

```
Alice public key: 0x66c808e6b5be6d6620934bc6ffa2b8b47f9786c002bfb06d53a0c27535641a5d1
Bob public key: 0xd15195432d1ac7f38aeb054d07d9b2e1faa913b78ad04d5efdd4a1ee8d9a3191
Now exchange the public keys (e.g. through Internet)
Alice shared key: 0x90f5a1cf2ed1dbb0322178df6bb0dd72c541884618b2989a3e5e663198667a621
Bob shared key: 0x90f5a1cf2ed1dbb0322178df6bb0dd72c541884618b2989a3e5e663198667a621
Equal shared keys: True
```

Due to randomization, if you run the above code, the **keys will be different**, but the calculated **shared secret** for Alice and Bob at the end will always be **the same**. The generated **shared secret** is a **257-bit** integer (compressed EC point for 256-bit curve, encoded as 65 hex digits).

Exercises: ECDH Key Exchange

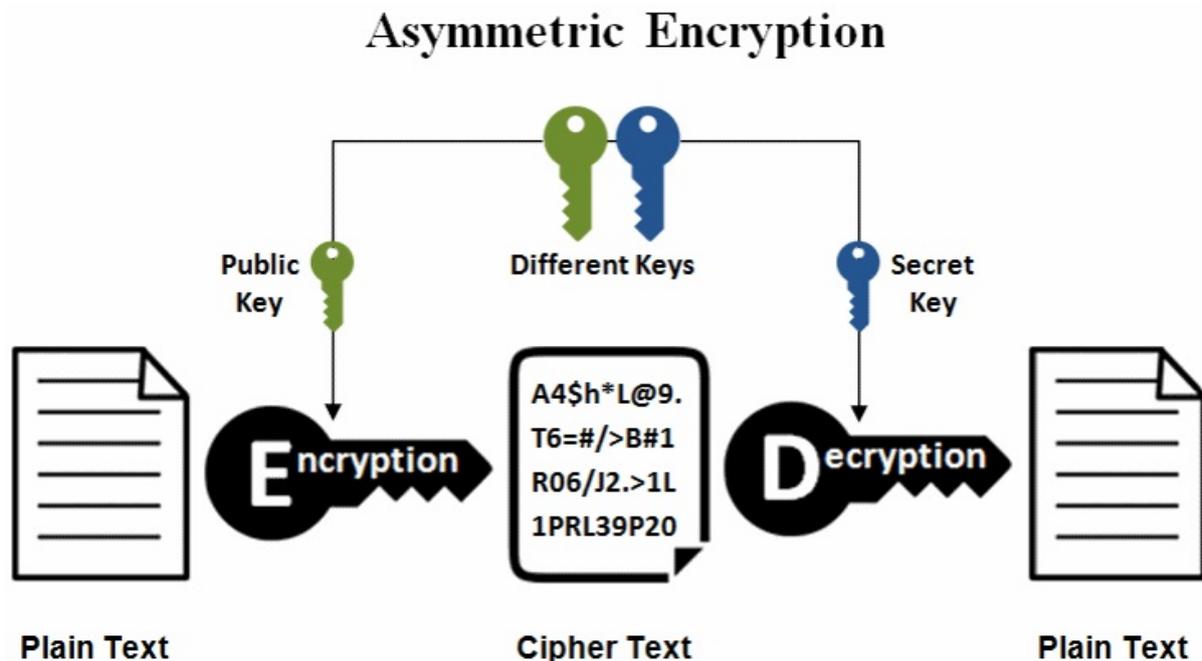
...

TODO

...

ECC-Based Encryption / Decryption

Assume we have a ECC **private-public key pair**. We want to encrypt and decrypt data using these keys. By definition, **asymmetric encryption** works as follows: if we **encrypt data by a private key**, we will be able to **decrypt** the ciphertext later by the corresponding **public key**:



The above process can be directly applied for the **RSA** cryptosystem, but not for the **ECC**. The elliptic curve cryptography (ECC) **does not directly provide encryption** method. Instead, we can design a **hybrid encryption scheme** by using the **ECDH** (Elliptic Curve Diffie–Hellman) key exchange scheme to derive a **shared secret key** for symmetric data encryption and decryption. Let's get into details how to do this.

ECC-Based Secret Key Derivation (using ECDH)

Assume we have a **cryptographic elliptic curve** over finite field, along with its generator point **G**. We can use the following two functions to calculate a **shared a secret key** for **encryption** and **decryption** (derived from the ECDH scheme):

- **calculateEncryptionKey**(pubKey) --> (sharedECCKey, ciphertextPubKey)
 1. Generate **ciphertextPrivKey** = *new random private key*.
 2. Calculate **ciphertextPubKey** = ciphertextPrivKey * G.
 3. Calculate the ECDH shared secret: **sharedECCKey** = pubKey * ciphertextPrivKey.
 4. Return both the **sharedECCKey** + **ciphertextPubKey**. Use the **sharedECCKey** for symmetric encryption. Use the randomly generated **ciphertextPubKey** to calculate the decryption key later.
- **calculateDecryptionKey**(privKey, ciphertextPubKey) --> sharedECCKey
 1. Calculate the the ECDH shared secret: **sharedECCKey** = ciphertextPubKey * privKey.
 2. Return the **sharedECCKey** and use it for the decryption.

The above calculations use the same math, like the **ECDH** algorithm (see the [previous section](#)). Recall that EC points have the following property:

- $(a * G) * b = (b * G) * a$

Now, assume that **a** = privKey, **a * G** = pubKey, **b** = ciphertextPrivKey, **b * G** = ciphertextPubKey.

The above equation takes the following form:

- $\text{pubKey} * \text{ciphertextPrivKey} = \text{ciphertextPubKey} * \text{privKey} = \text{sharedECCKey}$

This is what exactly the above two functions calculate, directly following the **ECDH key agreement** scheme.

ECC-Based Secret Key Derivation - Example in Python

The below Python code uses the `tinyec` library to generate a **ECC private-public key pair** (based on the `brainpoolP256r1` curve) and then derive a **secret key** (for encryption) from the ECC **public key** and later derive the same **secret key** (for decryption) from the **private key** and the generated earlier **ciphertext public key**:

```
from tinyec import registry
import secrets

curve = registry.get_curve('brainpoolP256r1')

def compress_point(point):
    return hex(point.x) + hex(point.y % 2)[2:]

def ecc_calc_encryption_keys(pubKey):
    ciphertextPrivKey = secrets.randbelow(curve.field.n)
    ciphertextPubKey = ciphertextPrivKey * curve.g
    sharedECCKey = pubKey * ciphertextPrivKey
    return (sharedECCKey, ciphertextPubKey)

def ecc_calc_decryption_key(privKey, ciphertextPubKey):
    sharedECCKey = ciphertextPubKey * privKey
    return sharedECCKey

privKey = secrets.randbelow(curve.field.n)
pubKey = privKey * curve.g
print("private key:", hex(privKey))
print("public key:", compress_point(pubKey))

(encryptKey, ciphertextPubKey) = ecc_calc_encryption_keys(pubKey)
print("ciphertext pubKey:", compress_point(ciphertextPubKey))
print("encryption key:", compress_point(encryptKey))

decryptKey = ecc_calc_decryption_key(privKey, ciphertextPubKey)
print("decryption key:", compress_point(decryptKey))
```

The code is pretty simple and demonstrates that we can generate a pair { **secret key + ciphertext public key** } from given **public key** and later we can recover the **secret key** from the pair { **ciphertext public key + private key** }. The above code produces output like this:

```
private key: 0x2e2921b4cde59cdf01e7a014a322abd530b3015085c31cb6e59502da761d29e9
public key: 0x850d3873cf4ac50ddb54ddb27f8225fc43bd3f4c2cc0a4f9d1f9ce15fc4eb711
ciphertext pubKey: 0x71586f9999d3ee050005054bc681c1d96c5eb054ca15b080ba245e495627003b0
encryption key: 0x9d13d3f8f9747669432f575731926b5ed99a6883f00146cbd3203ffa7ff8b1ae1
decryption key: 0x9d13d3f8f9747669432f575731926b5ed99a6883f00146cbd3203ffa7ff8b1ae1
```

It is clear that the **encryption key** (derived from the public key) and the **decryption key** (derived from the corresponding private key) **are the same**. This is due to the above discussed property of the ECC: $\text{pubKey} * \text{ciphertextPrivKey} = \text{ciphertextPubKey} * \text{privKey}$. These keys will be used for encryption and decryption in

an integrated encryption scheme. The above output will be different if you run the code (due to the randomness used to generate `ciphertextPrivKey`), but the encryption and decryption keys will always be the same (the ECDH shared secret).

ECC-Based Hybrid Encryption / Decryption - Example in Python

Once we have the **secret key**, we can use it for **symmetric data encryption**, using a symmetric encryption scheme like AES-GCM or ChaCha20-Poly1305. Let's implement a fully-functional **asymmetric ECC encryption and decryption** hybrid scheme. It will be based on the `brainpoolP256r1` curve and the **AES-256-GCM** authenticated symmetric cipher.

We shall use the `tinyec` and `pycryptodome` Python libraries respectively for ECC calculations and for the AES cipher:

```
pip install tinyec
pip install pycryptodome
```

Let's examine this full **ECC + AES hybrid encryption** example:

```
from tinyec import registry
from Crypto.Cipher import AES
import hashlib, secrets, binascii

def encrypt_AES_GCM(msg, secretKey):
    aesCipher = AES.new(secretKey, AES.MODE_GCM)
    ciphertext, authTag = aesCipher.encrypt_and_digest(msg)
    return (ciphertext, aesCipher.nonce, authTag)

def decrypt_AES_GCM(ciphertext, nonce, authTag, secretKey):
    aesCipher = AES.new(secretKey, AES.MODE_GCM, nonce)
    plaintext = aesCipher.decrypt_and_verify(ciphertext, authTag)
    return plaintext

def ecc_point_to_256_bit_key(point):
    sha = hashlib.sha256(int.to_bytes(point.x, 32, 'big'))
    sha.update(int.to_bytes(point.y, 32, 'big'))
    return sha.digest()

curve = registry.get_curve('brainpoolP256r1')

def encrypt_ECC(msg, pubKey):
    ciphertextPrivKey = secrets.randbelow(curve.field.n)
    sharedECKey = ciphertextPrivKey * pubKey
    secretKey = ecc_point_to_256_bit_key(sharedECKey)
    ciphertext, nonce, authTag = encrypt_AES_GCM(msg, secretKey)
    ciphertextPubKey = ciphertextPrivKey * curve.g
    return (ciphertext, nonce, authTag, ciphertextPubKey)

def decrypt_ECC(encryptedMsg, privKey):
    (ciphertext, nonce, authTag, ciphertextPubKey) = encryptedMsg
    sharedECKey = privKey * ciphertextPubKey
    secretKey = ecc_point_to_256_bit_key(sharedECKey)
```

```

    plaintext = decrypt_AES_GCM(ciphertext, nonce, authTag, secretKey)
    return plaintext

msg = b'Text to be encrypted by ECC public key and ' \
      b'decrypted by its corresponding ECC private key'
print("original msg:", msg)
privKey = secrets.randbelow(curve.field.n)
pubKey = privKey * curve.g

encryptedMsg = encrypt_ECC(msg, pubKey)
encryptedMsgObj = {
    'ciphertext': binascii.hexlify(encryptedMsg[0]),
    'nonce': binascii.hexlify(encryptedMsg[1]),
    'authTag': binascii.hexlify(encryptedMsg[2]),
    'ciphertextPubKey': hex(encryptedMsg[3].x) + hex(encryptedMsg[3].y % 2)[2:]
}
print("encrypted msg:", encryptedMsgObj)

decryptedMsg = decrypt_ECC(encryptedMsg, privKey)
print("decrypted msg:", decryptedMsg)

```

The above example starts from generating an ECC public and private key **key pair**: `pubKey` + `privKey`, using the `tinyec` library. These keys will be used to **encrypt** the message `msg` through the hybrid encryption scheme (asymmetric ECC + symmetric AES) and to **decrypt** is later back to its original form.

Next, we **encrypt** `msg` by using the `pubKey` and we obtain as a result the following set of output: { `ciphertext`, `nonce`, `authTag`, `ciphertextPubKey` }. The `ciphertext` is obtained by the symmetric AES-GCM encryption, along with the `nonce` (random AES initialization vector) and `authTag` (the MAC code of the encrypted text, obtained by the GCM block mode). Additionally, we obtain a randomly generated `ciphertextPubKey`, which will be used to recover the AES symmetric key during the decryption (using the ECDH key agreement scheme, as it was show before).

To **decrypt** the encrypted message, we use the data produced during the encryption { `ciphertext`, `nonce`, `authTag`, `ciphertextPubKey` }, along with the decryption `privateKey`. The result is the decrypted plaintext message. We use authenticated encryption (GCM block mode), so if the decryption key or some other parameter is incorrect, the decryption will fail with an **exception**.

Internally, the `encrypt_ECC(msg, pubKey)` function first generates an **ECC key-pair** for the ciphertext and calculates the symmetric encryption shared ECC key `sharedECCKey = ciphertextPrivKey * pubKey`. This key is an EC point, so it is then transformed to **256-bit AES secret key** (integer) though hashing the point's `x` and `y` coordinates. Finally, the **AES-256-GCM** cipher (from `pycryptodome`) **encrypts** the message by the 256-bit shared secret key `secretKey` and produces as **output** `ciphertext + nonce + authTag`.

The `decrypt_ECC(encryptedMsg{ciphertext, nonce, authTag, ciphertextPubKey}, privateKey)` function internally first calculates the symmetric encryption shared ECC key `sharedECCKey = privateKey * ciphertextPubKey`. It is an EC point, so it should be first transformed to **256-bit AES secret key** though hashing the point's `x` and `y` coordinates. Then the **AES-256-GCM cipher** is used to **decrypt** the `ciphertext + nonce + authTag` by the 256-bit shared secret key `secretKey`. The produced output is the original plaintext message (or an exception in case of incorrect decryption key or unmatching `authTag`).

The output from the above code looks like this:

```
original msg: b'Text to be encrypted by ECC public key and decrypted by its corresponding EC
```

```
C private key'
```

```
encrypted msg: {'ciphertext': b'b5953b3082fcefdbde91dd3c03cf83dde0822c19be6ae906a634db651152  
95e7cbcd7a1a492d69ba5be91990c70d8df9dc84360cf554f155ef81ce1f0ad44bd9fdabbc5f960517089262b339  
0e61b37610012bee4e6bcae335', 'nonce': b'9d55f4b5c87fff773d0457f3b23a953e', 'authTag': b'5c9d  
339778925aa4e44f43252a28681d', 'ciphertextPubKey': '0x21dbc985b625f2a42d0f86fc234b49b5547792  
8bae73dfac73bafd9bed50abe70'}
```

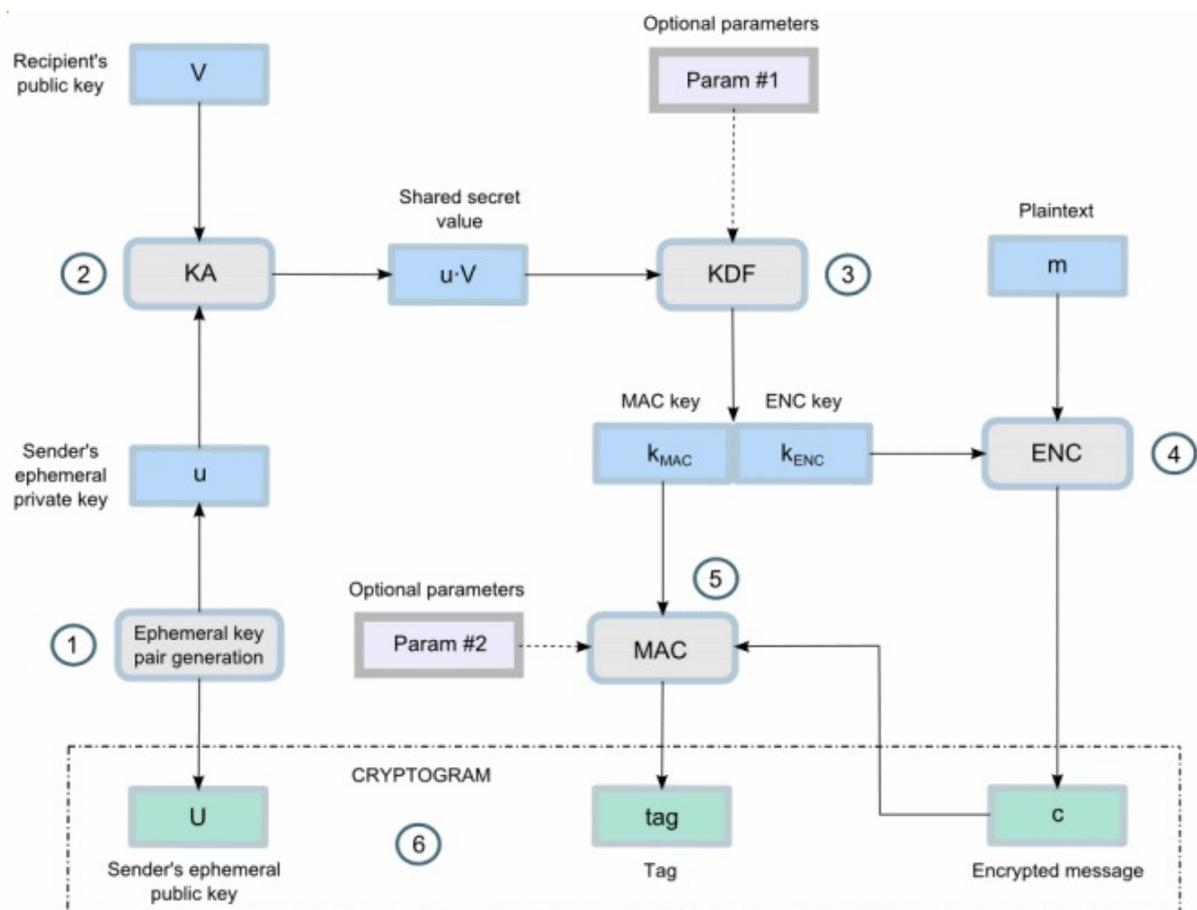
```
decrypted msg: b'Text to be encrypted by ECC public key and decrypted by its corresponding E  
CC private key'
```

Enjoy the above example, **play with it**, try to understand how exactly it works, try to change the underlying ECC curve, try to change the symmetric encryption algorithm, try to decrypt the ciphertext with wrong private key.

ECIES (Elliptic Curve Integrated Encryption Scheme)

A hybrid encryption scheme similar to the previously demonstrated code is standardized under the name **Elliptic Curve Integrated Encryption Scheme (ECIES)** in many crypto standards like [SECG SEC-1](#), [ISO/IEC 18033-2](#), [IEEE 1363a](#) and [ANSI X9.63](#). **ECIES** is a public-key authenticated encryption scheme, which works similarly to the above code examples, but uses a **KDF** (key-derivation function) for deriving separate **MAC key** and symmetric **encryption key** from the ECDH shared secret. It has many variants.

The **ECIES standard** combines ECC-based **asymmetric cryptography** with **symmetric ciphers** to provide data encryption by EC private key and decryption by the corresponding EC public key. The **ECIES** encryption scheme uses **ECC** cryptography (public key cryptosystem) + key-derivation function (**KDF**) + **symmetric encryption** algorithm + **MAC** algorithm, combined together like it is shown on the figure below:



The input of the **ECIES encryption** consists of recipient's **public key** + **plain text message**. The output consists of sender's ephemeral public key (**ciphertext public key**) + **encrypted message** (ciphertext + symmetric algorithm parameters) + **authentication tag** (MAC code):

- `ECIES-encrypt(recipientPublicKey, plaintextMessage) → { cipherTextPublicKey, encryptedMessage, authTag }`

The **ECIES decryption** takes the output from the encryption + the **recipient's private key** and produces the original plaintext message or detects a problem (e.g. integrity / authentication error):

- `ECIES-decrypt(cipherTextPublicKey, encryptedMessage, authTag, recipientPrivateKey,) → plaintextMessage`

The ECIES encryption scheme is a **framework**, not a concrete algorithm. It can be implemented by plugging different algorithms, e.g. the **secp256k1** or **P-521** elliptic curve for the public-key calculations + **PBKDF2** or **Scrypt** for KDF function + **AES-CTR** or **AES-GCM** or **ChaCha20-Poly1305** for symmetric cipher and authentication tag + **HMAC-SHA512** for MAC algorithm (in case of unauthenticated encryption).

In the next section we shall demonstrate through a **code example** how to use ECIES in practice.

ECIES (Elliptic Curve Integrated Encryption Scheme) - Example

Now, let's demonstrate how the **ECIES encryption scheme** works in practice in **Python**. We shall use a Python library `eciespy`:

```
pip install eciespy
```

A sample Python code to generate public / private **key pair** and **encrypt** and **decrypt** a message using ECIES is:

```
from ecies.utils import generate_eth_key
from ecies import encrypt, decrypt
import binascii

privKey = generate_eth_key()
privKeyHex = privKey.to_hex()
pubKeyHex = privKey.public_key.to_hex()
print("Encryption public key:", pubKeyHex)
print("Decryption private key:", privKeyHex)

plaintext = b'Some plaintext for encryption'
print("Plaintext:", plaintext)

encrypted = encrypt(pubKeyHex, plaintext)
print("Encrypted:", binascii.hexlify(encrypted))

decrypted = decrypt(privKeyHex, encrypted)
print("Decrypted:", decrypted)
```

The above code is pretty simple: just generate ECC **public + private key pair** using

`ecies.utils.generate_eth_key()` and call the `ecies.encrypt(pubKey, msg)` and `decrypt(privKey, encryptedMsg)` functions from the `eciespy` library.

The **output** from the above code looks like this:

```
Encryption public key: 0x0dc8e06c055b45ecf110258ed5c0261ce2019b1bd0f8f226dcd010dade448b8f304
a0915c68cdf7ddded8e4021d28fb92e27d08df695f48a0d2c41ddee750fc7
Decryption private key: 0x487fd8b53c471e3c38484a0f8e4751ace67a9ed28e60ea6b0b44c445b881f99d
Plaintext: b'Some plaintext for encryption'
Encrypted: b'045699078bbd101e270572d0d68e87a8f7b6cc377ebeeffb60d2fcac5dc7bdd86a26d7f79d13b92
e923a0e2cdbc418a7856b27157ef150d5c72f4f8f312467d13221ebe7049b7ed2f0ed253bce13117129a7b01bb88
1b8dfbf004ff11f3ebed4c732744bc49ea03230c2d1b2ec80774e79c075431d2019464d3de97ceb96'
Decrypted: b'Some plaintext for encryption'
```

The Python `eciespy` library internally uses **ECC** cryptography over the **secp256k1** curve + **AES-256-GCM** authenticated encryption. Note that the above encrypted message holds together 4 values: `{cipherPubKey, AES-nonce, authTag, AES-ciphertext}`, packed in binary form and not directly visible from the above output.

Exercises: ECC-Based Asymmetric Encrypt / Decrypt (ECIES)

Write a program to **encrypt / decrypt** a **message** by public / private key using **ECIES** (Elliptic Curve Integrated Encryption Scheme). The **encryption** will require an EC **public key** and **decryption** will require the corresponding EC **private key**. Internally, use ECC cryptography based on a **256-bit elliptic curve** by choice (e.g. `brainpoolP256t1`) and **symmetric encryption** by choice (e.g. AES-256-CTR + MAC, AES-128-GCM or ChaCha20-Poly1305), along with **key-derivation function** by choice (e.g. PBKDF2).

You are free to choose between writing your **own ECIES implementation**, following the **SECG-SEC-1** standard or use a **standard ECIES library** for your language, e.g.

- Python: <https://pypi.org/project/eciespy>
- JavaScript: <https://github.com/bitchan/ecrypto>
- C#: <https://github.com/VirgilSecurity/virgil-sdk-crypto-net>
- Java: <https://github.com/Arrayboom/smartbox-ecies-java>
- C, C++, PHP, Perl: <https://github.com/jedisct1/libsodium>

ECIES Encryption

Write a program to **encrypt a message** using the **ECIES** hybrid encryption scheme and a **256-bit ECC public key** (2 * 256 bits).

- The **input** consists of the **public key in hex** (at the first line, uncompressed, 128 hex digits) + **plaintext message** for encryption (at the second line).
- The **output** is the **hex-encoded encrypted message**. It may hold the ECC ciphertext public key + the ciphertext + MAC code + the symmetric key algorithm parameters, but this depends very much on the underlying algorithms and implementation.

Sample input:

```
552e2b308514b38e4989d71ed263e0af6376f65ba81a94ebb74f6fadcd223ee80aa8fb710cfb445e0871cd1c1a0c1
f2adb2b6eedc2a0470b04244548c5be518c8
Sample text for ECIES encryption.
```

Sample output:

It will be different for each program execution due to the randomness in the encryption scheme:

```
0442e2fba3fddba1ba9207f3276e141809782dc72529523aa1fcf35b15c4c22a9333ddacd7d64de4abd0a36138d4
30c50be7a98d5512cb8c2fe36ca45a0bbd7927c150ae3637c45093207531ce75e3841d4808ced85e82305d8da891
708c20479388f6d4a7cde213bb36bf860c5df0077358a942eeb9a4c23e89bcc11f11
```

ECIES Decryption

Write a program to **decrypt an encrypted message** created by the program from the previous example, using the **ECIES** hybrid encryption scheme and a **256-bit ECC private key**.

- The **input** consists of the **private key in hex** (at the first line, 64 hex digits) + **encrypted message** for decryption (at the second line).
- The **output** is the **decrypted plaintext message**. In case of **decryption problem** (e.g. incorrect decryption key or broken encrypted message), display `Error: cannot decrypt the message`.

Sample input:

```
27f07d3251dee39ec2c5ff800641f4d839e6f8065033e9a710ea2e519473bdd7
0442e2fba3fddba1ba9207f3276e141809782dc72529523aa1fcf35b15c4c22a9333ddacd7d64de4abd0a36138d4
30c50be7a98d5512cb8c2fe36ca45a0bbd7927c150ae3637c45093207531ce75e3841d4808ced85e82305d8da891
708c20479388f6d4a7cde213bb36bf860c5df0077358a942eeb9a4c23e89bcc11f11
```

Sample output:

```
Sample text for ECIES encryption.
```

Sample input:

This example holds an incorrect decryption private key:

```
9ab686c269b2c58f0fca699dde09cf24e23353e56bd60095d681b23709cb0dc3
0442e2fba3fddba1ba9207f3276e141809782dc72529523aa1fcf35b15c4c22a9333ddacd7d64de4abd0a36138d4
30c50be7a98d5512cb8c2fe36ca45a0bbd7927c150ae3637c45093207531ce75e3841d4808ced85e82305d8da891
708c20479388f6d4a7cde213bb36bf860c5df0077358a942eeb9a4c23e89bcc11f11
```

Sample output:

```
Error: cannot decrypt the message
```

Digital Signatures, ECDSA and EdDSA

Digital signatures are a cryptographic tool to **sign messages** and **verify message signatures** in order to provide proof of **authenticity** for digital messages or electronic documents. Digital signatures provide:

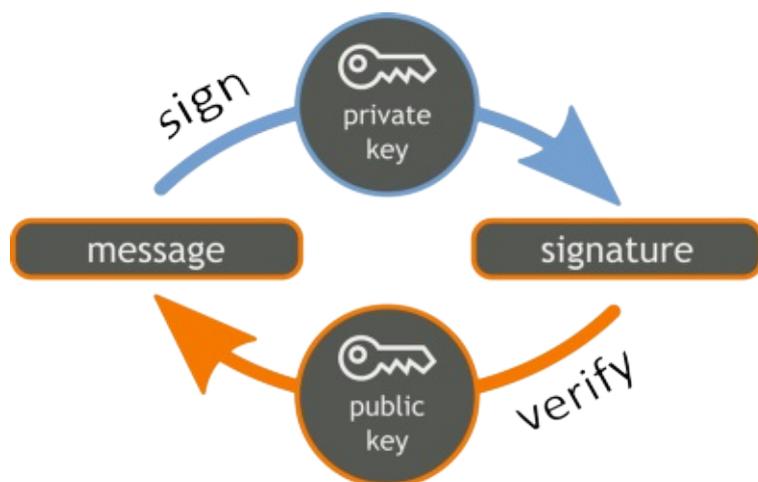
- Message **authentication** - a proof that certain known sender (secret key owner) have created and signed the message.
- Message **integrity** - a proof that the message was not altered after the signing.
- **Non-repudiation** - the signer cannot deny the signing of the document after the signature is once created.

Digital signatures are widely used today in the business and in the financial industry, e.g. for authorizing bank payments (money transfer), for exchange of signed electronic documents, for signing transactions in the public blockchain systems (e.g. transfer of coins, tokens or other digital assets), for signing digital contracts and in many other scenarios.

Digital signatures cannot identify who is the person, created a certain signature. This can be solved in combination with a **digital certificate**, which binds a public key owner with identity (person, organization, web site or other). By design digital signatures bind messages to public keys, not to digital identities.

Sign Messages and Verify Signatures: How It Works?

Digital signature schemes typically use a **public-key cryptosystem** (such as RSA or ECC) and use a **public / private key pairs**. A message is signed by a private key and the signature is verified by the corresponding public key:



Messages are **signed** by the sender using a **private key** (signing key). Typically the input message is **hashed** and then the **signature** is calculated by the signing algorithm. Most signature algorithms perform some calculation with the message hash + the signing key in a way that the result cannot be calculated without the signing key. The result from message signing is the **digital signature** (one or more integers):

```
signMsg(msg, privKey) signature
```

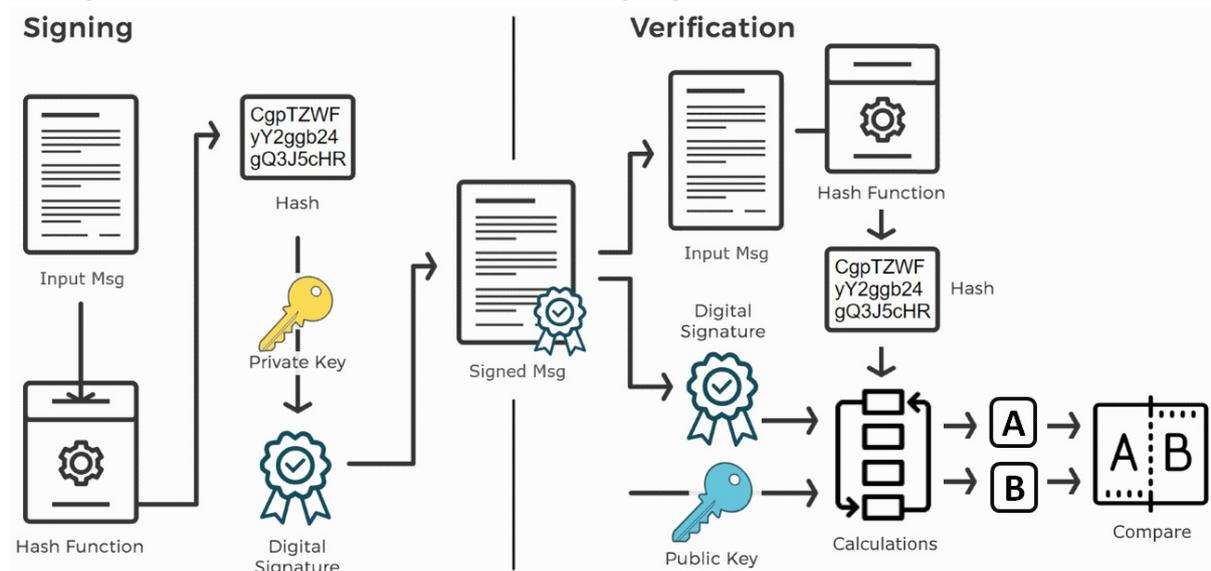
Message **signatures** are **verified** by the corresponding **public key** (verification key). Typically the signed message is **hashed** and some calculation is performed by the signature algorithm using the message hash + the public key. The result from signing is a boolean value (valid or invalid signature):

```
verifyMsgSignature(msg, signature, pubKey) valid / invalid
```

A **message signature** mathematically guarantees that certain message was signed by certain (secret) **private key**, which corresponds to certain (non-secret) **public key**. After a message is signed, the message and **the signature cannot be modified** and thus message **authentication** and **integrity** is guaranteed. Anyone, who knows the **public**

key of the message signer, can **verify the signature**. After signing the signature author cannot reject the act of signing (this is known as **non-repudiation**).

Most signature schemes work like it is shown at the following diagram:



At **signing**, the input message is **hashed** (either alone, or together with the public key and other input parameters), then some **computation** (based on elliptic curves, discrete logarithms or other cryptographic primitive) calculates the **digital signature**. The produced **signed message** consists of the original message + the calculated signature.

At **signature verification**, the message for verification is **hashed** (either alone or together with the public key) and some computations are performed between the message **hash**, the **digital signature** and the **public key**, and finally a **comparison** decides whether the signature is valid or not.

Digital signatures are different from **MAC** (message authentication codes), because MACs are created and verified by the same secret key using a **symmetric algorithm**, while digital signatures are created by a signing key and are verified by a different verification key, corresponding to the signing key using an **asymmetric algorithm**. Both signatures and MAC codes provide message authentication and integrity.

Digital Signature Schemes and Algorithms

Most public-key cryptosystems like **RSA** and **ECC** provide secure **digital signature schemes** (signature algorithms). Examples of well known digital signature schemes are: **DSA**, **ECDSA**, **EdDSA**, **RSA signatures**, **EIGamal signatures** and **Schnorr signatures**.

The above mentioned signature schemes are based on the difficulty of the **DLP** (discrete logarithm problem) and **ECDLP** (elliptic-curve discrete logarithm problem) and are **quantum-breakable** (powerful enough quantum computers may calculate the signing key from the message signature). **Quantum-safe** signatures (like **BLISS**, **XMSS** and **McEliece**) are not massively used, because of long key length, long signatures and slower performance, compared to ECDSA and EdDSA.

The most popular digital signature schemes (as of Nov 2018) are: **RSA signatures**, **ECDSA** and **EdDSA**. Let's give some details about them, along with some live code examples.

RSA Signatures

The **RSA** public-key cryptosystem provides a cryptographically secure **digital signature scheme** (sign + verify), based on the math of the **modular exponentiations** and discrete logarithms and the difficulty of the **integer factorization problem (IFP)**. The **RSA sign / verify** process works as follows:

- The **RSA sign** algorithm computes a message **hash**, then **encrypts** the hash with the private key exponent to

obtain the **signature**. The obtained signature is an **integer** number (the RSA encrypted message hash).

- The **RSA verify** algorithm first computes the message **hash**, then **decrypts** the message **signature** with the **public key** exponent and compares the obtained **decrypted hash** with the **hash** of the signed message to ensure the signature is valid.

RSA signatures are **deterministic** (the same message + same private key produce the same signature). A non-deterministic variant of RSA-signatures is easy to be designed by padding the input message with some random bytes before signing.

RSA signatures are widely used in modern cryptography, e.g. for signing digital certificates to protect Web sites. For example (as of Nov 2018) the Microsoft's official Web site uses `Sha256RSA` for its digital certificate. Nevertheless, the trend in the last decade is to move from RSA and DSA to **elliptic curve-based signatures** (like ECDSA and EdDSA). Modern cryptographers and developers **prefer ECC signatures** for their shorter key length, shorter signature, higher security (for the same key length) and better performance.

DSA (Digital Signature Algorithm)

The **DSA (Digital Signature Algorithm)** is a cryptographically secure standard for **digital signatures** (signing messages and signature verification), based on the math of the **modular exponentiations** and discrete logarithms and the difficulty of the discrete logarithm problem (**DLP**). It is alternative of RSA and is used instead of RSA, because of patents limitations with RSA (until Sept 2000). **DSA** is variant of the **EIGamal signature scheme**. The **DSA sign / verify** process works as follows:

- The **DSA signing** algorithm computes a message **hash**, then generates a random integer **k** and computes the **signature** (a pair of integers $\{r, s\}$), where **r** is computed from **k** and **s** is computed using the message **hash** + the **private key** exponent + the random number **k**. Due to randomness, the signature is **non-deterministic**.
- The **DSA signature verification** algorithm involves computations, based on the message **hash** + the **public key** exponent + the **signature** $\{r, s\}$.

The **random value k** (generated when the signature is computed) opens a potential vulnerability: if two different messages are signed using the same value of **k** and the same **private key**, then an attacker can compute the signer's private key directly (see <https://github.com/tintinweb/ecdsa-private-key-recovery>).

A **deterministic-DSA** variant is defined in **RFC 6979**, which calculates the random number **k** as **HMAC** from the private key, the message hash and few other parameters. The **deterministic DSA is considered more secure**.

In the modern cryptography, the **elliptic-curve-based signatures** (like ECDSA and EdDSA) are **preferred to DSA**, because of shorter key lengths, shorter signature lengths, higher security levels (for the same key length) and better performance.

ECDSA (Elliptic Curve Digital Signature Algorithm)

The **ECDSA** (Elliptic Curve Digital Signature Algorithm) is a cryptographically secure **digital signature** scheme, based on the elliptic-curve cryptography (**ECC**). **ECDSA** relies on the math of the **cyclic groups of elliptic curves over finite fields** and on the difficulty of the **ECDLP problem** (elliptic-curve discrete logarithm problem).

ECDSA is adaptation of the classical **DSA** algorithm, which is derived from the **EIGamal signature scheme**. More precisely, the **ECDSA** algorithm is a variant of the **EIGamal signature**, with some changes and optimizations to handle the representation of the group elements (the points of the elliptic curve). Like any other elliptic curve crypto algorithm, **ECDSA** uses an elliptic **curve** (like the `secp256k1`), **private key** (random integer within the curve key length - for signing messages) and **public key** (EC point, calculated from the private key by multiplying it to the curve generator point - for verifying signatures). The **ECDSA sign / verify** process works as follows:

- The **ECDSA signing** algorithm computes a message **hash**, then generates a random integer **k** and computes the **signature** (a pair of integers $\{r, s\}$), where **r** is computed from **k** and **s** is computed using the message **hash** + the **private key** + the random number **k**. Due to the randomness, the signature is **non-deterministic**.
- The **ECDSA signature verification** algorithm involves computations, based on the message **hash** + the **public**

key + the signature {r, s}.

The **random value k** (generated when the signature is computed) opens a potential vulnerability: if two different messages are signed using the same value of **k** and the same **private key**, then an attacker can compute the signer's private key directly (see <https://github.com/tintinweb/ecdsa-private-key-recovery>).

A **deterministic-ECDSA** variant is defined in [RFC 6979](#), which calculates the random number **k** as **HMAC** from the private key + the message hash + few other parameters. The **deterministic ECDSA is considered more secure**.

ECDSA signatures are the most widely used signing algorithm, used by millions every day (as of Nov 2018). For example, the digital certificates in Amazon Web sites are signed by the `Sha256ECDSA` signature scheme.

EdDSA (Edwards-curve Digital Signature Algorithm)

EdDSA (Edwards-curve Digital Signature Algorithm) is a fast **digital signature algorithm**, using **elliptic curves** in Edwards form (like [Ed25519](#) and [Ed448-Goldilocks](#)), a deterministic variant of the [Schnorr's signature](#) scheme, designed by a team of the well-known cryptographer [Daniel Bernstein](#).

EdDSA is more **simple** than **ECDSA**, more **secure** than **ECDSA** and is designed to be **faster** than **ECDSA** (for curves with comparables key length). Like **ECDSA**, the **EdDSA** signature scheme relies on the difficulty of the **ECDLP problem** (elliptic-curve discrete logarithm problem) for its security strength.

The **EdDSA** signature algorithm is works with Edwards elliptic curves like **Curve25519** and **Curve448**, which are highly optimized for **performance** and **security**. It is shown that **Ed25519 signatures** are typically **faster** than traditional **ECDSA signatures** over curves with comparable key length. Still, the performance competition is disputable. The **EdDSA sign / verify** process works as follows:

- The **EdDSA signing** algorithm generates a deterministic (not random) integer **r** (computed by **hashing** the **message** and the hash of the **private key**), then computes the **signature {Rs, s}**, where **Rs** is computed from **r** and **s** is computed from the **hash** of (the **message** + the **public key** derived from the private + the number **r**) + the **private key**. The signature is **deterministic** (the same message signed by the same key always gives the same signature).
- The **EdDSA signature verification** algorithm involves elliptic-curve computations, based on the **message** (hashed together with the public key and the EC point **Rs** from the signature) + the **public key** + the number **s** from the **signature {Rs, s}**.

By design **EdDSA** signatures are **deterministic** (which improves their security). A non-deterministic variant of EdDSA-signatures is easy to be designed by padding the input message with some random bytes before signing.

A short comparison between **Ed25519 EdDSA** signatures and **secp256k1 ECDSA** signatures is given below:

	EdDSA-Ed25519	ECDSA-secp256k1
Performance (source)	8% faster	8% slower
Private key length	32 bytes (256 bits = 251 variable bits + 5 predefined)	32 bytes (256 bits)
Public key length (compressed)	32 bytes (256 bits = 255-bit y-coordinate + 1-bit x coordinate)	33 bytes (257 bits = 256-bit x-coordinate + 1-bit y-coordinate)
Signature size	64 bytes (512 bits)	64 bytes (512 bits) or 65 bytes (513 bits) with the public key recovery bit
Public key recovery	not possible (signature verification involves hashing of the public key)	possible (with 1 recovery bit added in the signature)
Security level (source)	~128 bit (more precisely 125.8)	~128 bit (more precisely 127.8)
SafeCurves secu		

rity (source)	11 of 11 tests passed	7 of 11 tests passed
-------------------------------	-----------------------	----------------------

Modern developers often use **Ed25519 signatures** instead of **256-bit curve ECDSA** signatures, because EdDSA-Ed25519 signature scheme uses keys, which fit in 32 bytes (64 hex digits), signatures fit in 64 bytes (128 hex digits), signing and verification is faster and the security is considered better.

Public **blockchains** (like Bitcoin and Ethereum) often use **secp2561-based ECDSA** signatures, because the signer's public key (and its blockchain address) can be easily recovered from the signature (together with the signed message) by adding just 1 additional bit to the signature.

In the general case, it is considered that **EdDSA signatures are recommended** to ECDSA, but this is highly disputable and depends on the use case, on the curves involved and many other parameters.

Other Signature Schemes and Algorithms

Most signature algorithms are derived from generic signature schemes like [ElGamal signatures](#) and [Schnorr signatures](#).

- **RSA signature** is derived from the **RSA** encryption scheme.
- **DSA** and **ECDSA** are derived from **ElGamal** signature scheme.
- **EdDSA** is derived from the **Schnorr** signature scheme.

Other **signature schemes** include:

- **ECGDSA**: an elliptic-curve digital signature scheme (based on the difficulty of the ECDLP problem), a slightly simplified variant of ECDSA, known as **the German version of ECDSA**.
- **ECKDSA**: an elliptic-curve digital signature scheme (based on the difficulty of the ECDLP problem), a complicated variant of ECDSA, known as **the Korean version of ECDSA**. The ECKDSA signs given **message** by given EC **private key**, along with the signer's **digital certificate** hash. This add **identity** to the digital signature, in addition to message authentication, integrity and non-repudiation.
- **SM2 signature**: an elliptic-curve digital signature scheme (based on the difficulty of the ECDLP problem), known as the **Chinese digital signature algorithm**, developed by the Chinese Academy of Science.
- **GOST R 34.10-2001**: an elliptic-curve digital signature scheme (based on the difficulty of the ECDLP problem), known as the **Russian digital signature algorithm**, one of the Russian cryptographic standard algorithms (called GOST algorithms).

After the short review of the most popular digital signature algorithms, let's get into technical details about the **RSA sign**, **ECDSA** and **EdDSA** signature algorithms, with code examples.

RSA Signatures: Sign and Verify

The **RSA** public-key cryptosystem provides a **digital signature scheme** (sign + verify), based on the math of the **modular exponentiations** and discrete logarithms and the computational difficulty of **the RSA problem** (and its related integer factorization problem). The **RSA sign / verify** algorithm works as described below.

Key Generation

The RSA algorithm uses **keys** of size 1024, 2048, 4096, ..., 16384 bits. RSA supports also longer keys (e.g. 65536 bits), but the performance is too slow for practical use (some operations may take several minutes or even hours). For 128-bit security level, a 3072-bit key is required.

The **RSA key-pair** consists of:

- public key $\{n, e\}$
- private key $\{n, d\}$

The numbers n and d are typically big integers (e.g. 3072 bits), while e is small, typically 65537.

By definition, the RSA key-pairs has the following property:

$$(m^e)^d \equiv (m^d)^e \equiv m \pmod{n} \text{ for all } m \text{ in the range } [0 \dots n)$$

RSA Sign

Signing a message msg with the private key exponent d :

1. Calculate the message hash: $h = \text{hash}(msg)$
2. Encrypt h to calculate the signature: $s = h^d \pmod{n}$

The hash h should be in the range $[0 \dots n)$. The obtained **signature** s is an integer in the range $[0 \dots n)$.

RSA Verify Signature

Verifying a signature s for the message msg with the public key exponent e :

1. Calculate the message hash: $h = \text{hash}(msg)$
2. Decrypt the signature: $h' = s^e \pmod{n}$
3. Compare h with h' to find whether the signature is valid or not

If the signature is correct, then the following will be true:

$$h' = s^e \pmod{n} = (h^d)^e \pmod{n} = h$$

The **RSA sign / verify algorithm** is pretty simple. Let's implement it with some code.

RSA: Sign / Verify - Examples

Let's demonstrate in practice the **RSA sign / verify** algorithm. We shall use the `pycryptodome` package in Python to generate **RSA keys**. After the keys are generated, we shall compute RSA digital signatures and verify signatures by a simple modular exponentiation (by encrypting and decrypting the message hash).

```
pip install pycryptodome
```

Next, generate a 1024-bit **RSA key-pair**:

```
from Crypto.PublicKey import RSA

keyPair = RSA.generate(bits=1024)
print(f"Public key: (n={hex(keyPair.n)}, e={hex(keyPair.e)})")
print(f"Private key: (n={hex(keyPair.n)}, d={hex(keyPair.d)})")
```

The **output** from the above code might look like this (it will be different at each execution due to randomness):

```
Public key: (n=0xf51518d30754430e4b89f828fd4f1a8e8f44dd10e0635c0e93b7c01802729a37e1dfc8848d
7fbbdf2599830268d544c1ecab4f2b19b6164a4ac29c8b1a4ec6930047397d0bb93aa77ed0c2f5d5c90ff3d45875
5b2367b46cc5c0d83f8f8673ec85b0575b9d1cea2c35a0b881a6d007d95c1cc94892bec61c2e9ed1599c1e605f,
e=0x10001)
Private key: (n=0xf51518d30754430e4b89f828fd4f1a8e8f44dd10e0635c0e93b7c01802729a37e1dfc8848d
7fbbdf2599830268d544c1ecab4f2b19b6164a4ac29c8b1a4ec6930047397d0bb93aa77ed0c2f5d5c90ff3d45875
5b2367b46cc5c0d83f8f8673ec85b0575b9d1cea2c35a0b881a6d007d95c1cc94892bec61c2e9ed1599c1e605f,
d=0x165ecc9b4689fc6ceb9c3658977686f8083fc2e5ed75644bb8540766a9a2884d1d82edac9bb5d312353e63e4
ee68b913f264589f98833459a7a547e0b2900a33e71023c4dedb42875b2dfdf412881199a990dfb77c097ce71b9c
8b8811480f1637b85900137231ab47a7e0cbecc0b011c2c341b6de2b2e9c24d455ccd1fc0c21)
```

Now, let's **sign a message**, using the RSA private key $\{n, d\}$. Calculate its **hash** and raise the hash to the power d modulo n (encrypt the hash by the private key). We shall use **SHA-512 hash**. It will fit in the current RSA key size (1024). In Python we have **modular exponentiation** as built in function `pow(x, y, n)`:

```
# RSA sign the message
msg = b'A message for signing'
from hashlib import sha512
hash = int.from_bytes(sha512(msg).digest(), byteorder='big')
signature = pow(hash, keyPair.d, keyPair.n)
print("Signature:", hex(signature))
```

The obtained digital signature is an integer in the range of the RSA key length $[0 \dots n]$. For the above **private key** and the above **message**, the obtained **signature** looks like this:

```
Signature: 0x650c9f2e6701e3fe73d3054904a9a4bbdb96733f1c4c743ef573ad6ac14c5a3bf8a4731f6e6276f
aea5247303677fb8dbdf24fff78e53c25052cdca87eefee85476bcb8a05cb9a1efef7cb87dd68223e117ce800ac4
6177172544757a487be32f5ab8fe0879fa8add78be465ea8f8d5acf977e9f1ae36d4d47816ea6ed41372b
```

The **signature** is **1024-bit integer** (128 bytes, 256 hex digits). This signature size corresponds to the RSA key size.

Now, let's **verify the signature**, by decrypting the signature using the public key (raise the **signature** to power e modulo n) and comparing the obtained **hash from the signature** to the **hash** of the originally signed message:

```
# RSA verify signature
```

```
msg = b'A message for signing'
hash = int.from_bytes(sha512(msg).digest(), byteorder='big')
hashFromSignature = pow(signature, keyPair.e, keyPair.n)
print("Signature valid:", hash == hashFromSignature)
```

The output will show `True`, because the signature will be valid:

```
Signature valid: True
```

Now, let's try to **tamper the message** and verify the signature again:

```
# RSA verify signature (tampered msg)
msg = b'A message for signing (tampered)'
hash = int.from_bytes(sha512(msg).digest(), byteorder='big')
hashFromSignature = pow(signature, keyPair.e, keyPair.n)
print("Signature valid (tampered):", hash == hashFromSignature)
```

Now, the signature will be **invalid** and the output from the above code will be:

```
Signature valid (tampered): False
```

Enjoy **playing with the above RSA sign / verify examples**. Try to modify the code, e.g. use 4096-bit keys, try to tamper the public key at the signature verification step or the signature.

The RSA Signature Standard PKCS#1

The simple use of **RSA signatures** is demonstrated above, but the industry usually follows the **crypto standards**. For the RSA signatures, the most adopted standard is "**PKCS#1**", which has several versions (1.5, 2.0, 2.1, 2.2), the latest described in [RFC 8017](#). The PKCS#1 standard defines the RSA signing algorithm (**RSASP1**) and the RSA signature verification algorithm (**RSAPV1**), which are almost the same like the implemented in the previous section.

To demonstrate the **PKCS#1 RSA digital signatures**, we shall use the following code, based on the `pycryptodome` Python library, which implements RSA sign / verify, following the **PKCS#1 v1.5** specification:

```
from Crypto.PublicKey import RSA
from Crypto.Signature.pkcs1_15 import PKCS115_SigScheme
from Crypto.Hash import SHA256
import binascii

# Generate 1024-bit RSA key pair (private + public key)
keyPair = RSA.generate(bits=1024)

# Sign the message using the PKCS#1 v1.5 signature scheme (RSASP1)
msg = b'A message for signing'
hash = SHA256.new(msg)
signer = PKCS115_SigScheme(keyPair)
signature = signer.sign(hash)
print("Signature:", binascii.hexlify(signature))

# Verify valid PKCS#1 v1.5 signature (RSAPV1)
msg = b'A message for signing'
hash = SHA256.new(msg)
signer = PKCS115_SigScheme(keyPair)
try:
```

```
    signer.verify(hash, signature)
    print("Signature is valid.")
except:
    print("Signature is invalid.")

# Verify invalid PKCS#1 v1.5 signature (RSAVP1)
msg = b'A tampered message'
hash = SHA256.new(msg)
signer = PKCS115_SigScheme(keyPair)
try:
    signer.verify(hash, signature)
    print("Signature is valid.")
except:
    print("Signature is invalid.")
```

The output from the above code demonstrates that the **PKCS#1 RSA signing** with 1024-bit RSA private key produces **1024-bit digital signature** and that it is successfully validated afterwards with the corresponding public key. If the message or the signature or the public key is tampered, the signature fails to validate. The output from the above example looks like this:

```
Signature: b'243b9ed6561ab3bddead98508af0ac34b4567b1358011ace24db71ce2bc7f1a2e942b6231aa84cb
07bae85b668d7c7cd0bc40cdda6f8162de57f0ee842e589c58f94aa4f96d51355f8aa395d7db950ebb9d375fca31
24b6222699a645e93287bc6f5eb5b750fc0b470588f949a887dff75ed42cf01d9642a5d497f609b8cd043 '
Signature is valid.
Signature is invalid.
```

Note that in real-world applications the RSA key length should be **at least 3072 bits** to provide secure enough signatures.

Exercises: RSA Sign / Verify

In this exercise we shall **sign** messages and verify signatures using the **PKCS#1 v.1.5 RSA** signature algorithm with 4096-bit keys, following the technical specification from [RFC 8017](#), using **SHA3-512** for hashing the input message. The RSA-PKCS1 v1.5 digital signature algorithm can be found as library for the most programming languages.

The RSA **private key** will be given encoded in **PEM format** ([RFC 7468](#), see the example). The corresponding RSA **public key** will also be given encoded in **PEM format**. The RSA **signature** is 4096-bit integer (1024 hex digits).

Sign a Message with RSA

Write a program to sign given text **message** with given 4096-bit **private key**, using the **PKCS#1 v.1.5 RSA** signature algorithm with **SHA3-512** hashing for the input message. The input consists of 2 text lines. The **first line holds** the input **message** for signing. The **next few lines** holds the private key as **hex string**. Print the **output** as JSON document, holding the input **message** + the **public key** of the signer (as hex string) + the RSA **digital signature** (as hex string).

Sample **input** (text message + PEM-encoded RSA private key):

```
Message for RSA signing
-----BEGIN RSA PRIVATE KEY-----
MIIEJwIBAAKCAgEAoyjbzKmmss6BU/eCByhs3bFRBV1dM9D1aM7WhPZEgmY17C10
b0kmEPG20/GaNoB0nwtHykzvpYS42rr/D8I0LeJ/2dz0uStQv6kQDIDmNRTthW9
spcuBsQw3e1bRyf0JrL8ot1X8M4k0NGU7w508htPzRZXR52Zonv8G/N35ePDV5Xf
5mrcOP0ebconndXHRqfLlyXCxCS5x0q1JGX1d/oe2Xop7VfBP/UhrrmNmstZsBEv
L3f33+a482xuctVPIQcv62N31aR2wK0JN9CpF0pqjwR8cj9oLjbCRHd5T/+bgc6D
0/XVzr9DAPVaYLZvm8puMhhgckNIjgPOfwsVyI+/G1zuUH5caFMhMaS95M2a1XNI
wJ5sF25PFpsttIrFXC+b0J5325mk5qvzCgnu0DrWT2VPBNjw4rvmhBPnFb/JdXy5
e5ZlpxtGRpvKYcBxDk6yqgt9SC/e81+ tqCrFSQ9cG1QCV/BnZw4cd1JHe3/AHot1
YeWjb8YpIfiFj7Wx Fb0g9GIYJZTh3dC9hrNOFsIoOL+ZAYNfp43u0FJT7qtC7Ct
xPvJnt/h8ACE07ewgkq3M00zmT2RNI f0HhysMjwdIh1Y8QGG0V3+zDu9nTx63tvv
aBikPwtZuCz76ipWb5Kt6SA80fsN/Q0tDf1mBwj1T6UvoivgK8C7iIrwQGUCaWEA
AQKCAgAEuDBD3672jG3DiCpLzkMpYNvhTTzmXxvJsvtC1V6MqUHPPrq9DQvGL1
JP2zfIsxpEss43YeVL+wD6EIkfTmRyTpt6FLYXXjIIDhJyPdB8yudP/0wR9FK10C
W6yp8bf97IcEvdDfQPHpjcs5mTYqW3xBbM5xEdA7ihyLlj/hxI7VFNWuVrS6hvpM
93o6Xr3l2r2DZID05R/uJj/LztBEFVesiBezCx/TbXOL9NV0I7SkR1uT6WQpUiw
iIt0nDkaHDu/JlR8ppJ/6RYzJLS1BG3Hcw/fEwz7RgLMbe90h0085u1/PwNCRg5c
6DBbw3nX6Bo3gzThotIUsKkufgZsW+eGIEeH5SFGFcIsnVU8AP/FZEQ3RMNXrxdx
U67asN7iQICVJuj0419aZ/r+GiLnCp5+UIippYVBXf5RaHcZiurQ/SbuVppZskrA
8oDI4ykB2Y3DPxqfjygzSkxhWBncBv5CNvAwpFTu94NDYCEgz203x4VaDvc52RNF
I1X8Hc9kvvppWB024i8jv1Unnb5ojEh1JNfAmPn8GMI/d+j5oVBC0Qemiwdc1cp
Okik1iivvFPVSwZo1NC1EPg8K9TLMynE/PqV+0Ud05qeXn4Y08b9sqrF02JiTDH
aEdYtvcM80FPjqXwi4zIDkr9Im4fuX2anzFU3T0od6Ye150yPwKCAQEAYPQD5rQM
4yR6iUTTV0x00Se5noY2ztUL3JTioqkm+9kBDPhdGYRIB7sN1RacWgXdqPRC4M0h
+1IWYg6lJj8IsjJjEszpRmSwLUpt2xKuh+beaWPQhg/bx8jGUrTFqNo6FPJ8RT/3
UaeRUEmqj3vxw+1c7pZjk9GxVMD7uWbMU4GLE20h97oMr5cW4wvzzWoeofKMkyn
sDORAC8siwZSAYKJED1ziDujicFhVmU780igWiTODuAg4GXVqsBXC6FIMs6zGod
7D2+AUZGBj4H4o03miTMYsWCPDkPZMtvAbAKnNEp6UB4/j8ofApDUIYWF5fLBZgk
2Ri6x4XM/0TS9wKCAQEAZciWYXKjkQIVeTIXM1YGRE+irrsWL/wPmRBVFX13THqW
7U1/rYNHNS1JaIEe9KJqgLUD1kQ+x9vHiSgKTVjrL2c/VyZrJbI30UIAZmQ9AhpU
qJ709waXs2VnRdZspAp/1rnf5H8u/1zWHTebzvfJEq6Ys9vRam6Jcr1wI1/9u6pJ
lwS0ps67YVxAp2kAC6w2HuMlWQYTaphOHiyS0+Jx6nk15r4Vgbv5mJvjQetS65AD
gJPhtep+9Sa1HvzUPQ3Gu7sz0uz5vjRJTv4RRgNSEGLHjJf1gBs11epsB1H7Xrh
rotRox8AxVnbdXOVcONBB7GAAb7Z+bPCmXdSS2sUgwKCAQAy3Q/197tcgwDwXOu
```

```

rB9pPA0i1iYM0+0JY7uorLCJ+kCTWnDzqxbyKqMNF40J9ZOE1vIAxE/Ydwf9WiUV
eh7bfGL/JNc2xLSsjsdTiJyquNQLtFWC3ZWnoMaJn7tX+JNFFLc8SRoIQpD6l6oA
8JTHex1h++PrK+5kR7vjX4A1YrGwJWnmBZhkuQlKUfDqq1hQhLXE8xPr7To0SeMk
/OKNAkemWVhRAMg2nei7gos3xF76HK11Jy/k3ryGIrjb8S2x0qRTIhGngtWySFHT
28XrowfpDXr7ER7tuIIWghsrV28zgdIC05wwfRXWKFZHDY00HQoBu/7304ooAXBI
cqp9AoIBAHQHYYNkeAVS/z7VZm7jQjVSEZAJvKbhoInVQ6QSUim1FVRF1M/orVDQ
NIvTnYux6AsaBUfSwmM9YIXdHzHta04ZcQVajB99FNyB+M3CxnNgk/RPbB+8f9y0
2GPwOuFjaiFQPIVBkOY7Gh7vM9LGs4DtIPyIfNND7/HaDlhjz1T49vVHhIdZGR4U
PgAmm/f46asQuEDVhC0eIy2wQ+OwEjr6jQHF06siqeD6RHDu1pprYQ4mU6WwYm6/
nHAUbjbehadkLxHsak1IhCAAI1RYfwJ82bbUDnrk07iBrNcDcpA9u7LbwRifrTH
rY3T1fcIq6oC0wIo8g5w5NBTDvunLLEcggEASMBX4MfNUOYNqWoutQKNSAJjTdSy
MpSXD9MHI5uZnTEG2VEGbN74oQmMx1BtANI0hz17XrYy/fCGyRXiwiQtszMYoYDN
J3HXPQFCvUfTsb8f8f0CbfnRyxIA4jRlFYrobjTb0kBo16ISsT7+YQIean96Ppk
PHdAA+K0/yDvhG8wC1CUwzaBwfayKM2bhGQgVIqM+ZCV/Ji0g1KKiMVXUF0cvG6
FI+vQvLnxE7zbF0ewbZmqYqGen7J7g0pNjYHvKSIia3KpJpc8IzWsrOH+8xOIX06
fi/wuMFX07Zz9wyzEKuNXiFExo3FNrChSNonWCJj14Q8cuNW4S0J9hhvAg==
-----END RSA PRIVATE KEY-----

```

Sample **output** (message + hex-encoded signature + PEM-encoded RSA public key):

```

Message for RSA signing
28ce01ec0f8511e92407496b7aa777a8b38fbc27a3656c9862830427bf8a006d59d19ac3b3441f174aee2fe6560b
2b5f91aeb72ffa150dad4bd4e93a3b78eabfb48f8051b09cdfc5cbb63b014a7beb81cae4220fa01bedf5fa10aa3b
6dcdd7ad229fd7ce2e8883a4893bef532c7b7822993408577a20a198387e073e15e53ffab8d7f3c46fa50347cd45
d7a5d319b66d4ead658297a2e7127cc6a1dc5e104beb3ae2ba390d5ae90a66bd8a537e4df296c212186754206c5a
0b155c226a3a580e943db33bf9e9829ea3e38f00aafa0e4839bf4673d83b14c74cd9ac359afc3fa56ca7f3a8391e
6fb234561b9deca9ac3fa9147d70b0f1189db734b5ea96f287c604467dab0efbf5f1c78437920283dfe6fbc56f4
e859e24ee37552b3b8de5c4fa35e65cf9fe63ec7e2fd155d503fec2648303e495d86af97ac53c14b203bdc723336
5d481fda08cac905c48e31bd389bea0f983e035225c4fca269a66009332bbba83542dcd484832f4d40ab14eee360
9aa78c01539984758592de3355b528de3c8a669442cbd6f038082226ef0c13924b9553b7af7ff02a6916f106061a7
bcbe093f795d9455709ab2b030fff485e9993eb978ae57cabeade4747e981b47c36128c39df970d1f29c4e4a3c225
a8384d55995599c9ce3afe14068f4eb4edeb120beb534955b519a4424f82136d3511a0dafad46b010c3fa15bd233
bdc85a7fd42a
-----BEGIN PUBLIC KEY-----
MIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICGKCAgEAOYjzbkMms6BU/eCBYhs
3bFRBVldM9DlaM7WhPZEgmY17C10b0kmEPG20/GaNoB0nwpTHykzvpYS42rr/D8I
0LeJ/2dz0uStQv6kQDIDmNRTtHw9spcuBsqw3e1bRyf0JrL8ot1X8M4k0NGU7w50
8htPzRZXR52Zonv8G/N35ePDV5XF5mrcOPOebconndXHRqfLlyXCxCS5x0q1JGX1
d/oe2Xop7VfBFP/UhrrmNmstZsBEvL3f33+a482xuCTvPIQcv62N31ar2wK0JN9Cp
F0pqjwR8cj9oLjbcRHd5T/+bgc6D0/XVzr9DAPVaYLZvm8puMhhgckNIjgP0fwsV
yI+/G1zuUH5caFMhMaS95M2a1XNIwJ5sF25PFpsttIrFXC+b0JS325mk5qvzCgnu
0DrWT2VPBNjw4rvmhBPnFb/JdXy5e5ZIpxtGRpvKYcBxDk6yqgt9SC/e81+ttqCrF
SQ9cG1QCv/BnZw4cd1JHe3/AHot1Yewjb8YpIFifJ7WxMb0g9GIYJZTh3dC9hrNO
fSsIo0L+ZAYnFp43u0FJT7qtC7CtXpVJnt/h8ACE07ewgkq3M00zmT2RNIff0HhyS
MjwdIh1Y8QGG0V3+zDu9nTx63tvvaBikPwtZuCz76ipWb5Kt6SA80fsN/Q0tDF1m
Bwj1T6UvoivgK8C7iIrwQGUCAwEAAQ==
-----END PUBLIC KEY-----

```

Verify Message Signature with RSA

Write a program to **verify RSA signature** (calculated by using **PKCS#1 v.1.5 + SHA3-512**), created by the previous exercise. The **input** comes as signed **message** (first line) + RSA digital **signature** (second line) + 4096-bit RSA **public key** (all input lines to the end). Print as **output** a single word: **"valid"** or **"invalid"**.

Sample input (correctly signed message):

```

Message for RSA signing
28ce01ec0f8511e92407496b7aa777a8b38fbc27a3656c9862830427bf8a006d59d19ac3b3441f174aee2fe6560b
2b5f91aeb72ffa150dad4bd4e93a3b78eabfb48f8051b09cdfc5cbb63b014a7beb81cae4220fa01bedf5fa10aa3b
6dcdd7ad229fd7ce2e8883a4893bef532c7b7822993408577a20a198387e073e15e53ffab8d7f3c46fa50347cd45
d7a5d319b66d4ead658297a2e7127cc6a1dc5e104beb3ae2ba390d5ae90a66bd8a537e4df296c212186754206c5a
0b155c226a3a580e943db33bf9e9829ea3e38f00aafa0e4839bf4673d83b14c74cd9ac359afc3fa56ca7f3a8391e
6fb234561b9deca9ac3fa9147d70b0f1189db734b5ea96f287c604467dab0efbf5f1c78437920283dfe6fbc56f4
e859e24ee37552b3b8de5c4fa35e65cf9fe63ec7e2fd155d503fec2648303e495d86af97ac53c14b203bdc723336
5d481fda08cac905c48e31bd389bea0f983e035225c4fca269a66009332bbba83542dcd484832f4d40ab14eee360
9aa78c01539984758592de3355b528de3c8a669442cbd6f03808226ef0c13924b9553b7af7ff02a6916f106061a7
bcbe093f795d9455709ab2b030ff485e9993eb978ae57cabeade4747e981b47c36128c39df970d1f29c4e4a3c225
a8384d55995599c9ce3afe14068f4eb4edeb120beb534955b519a4424f82136d3511a0dafad46b010c3fa15bd233
bdc85a7fd42a
-----BEGIN PUBLIC KEY-----
MIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICGKCAgEAOyjbzkmMss6BU/eCBYhs
3bFRBVldM9DlaM7WhPZEgMY17C10b0kmEPG20/GaNoB0nwptHykzvpYS42rr/D8I
0LeJ/2dz0uStQv6kQDIDmNRTthW9spcuBsqW3e1bRyf0JrL8ot1X8M4k0NGU7w50
8htPzRZXR52Zonv8G/N35ePDV5XF5mrcOPOebconndXHRqfLlyXCxCS5x0q1JGX1
d/oe2Xop7VfBP/UhrrmNmstZsBEvL3f33+a482xuCTvPIQcv62N31aR2wK0JN9Cp
FOpqjwR8cj9oLjbCRhd5T/+bgc6D0/XVzr9DAPVaYLZvm8puMhhgckNIjgPOfwsV
yI+/G1zuUH5caFMhMaS95M2a1XNIwJ5sF25PFpsttIrFXC+b0JS325mk5qvzCgnu
0DrwT2VPBnjw4rvmhBPNFb/JdXy5e5ZIpxtGRpvKYcBxDk6yqgt9SC/e81+ tqCrF
SQ9cG1QCv/BnZw4cd1JHe3/AHot1Yewjb8YpIFifJ7WxFb0g9GIYJZTh3dC9hrN0
fSsIoOL+ZAYnFp43u0FJT7qtC7CtXpVJnt/h8ACE07ewgkq3M00zmT2RNIfoHhys
MjwdIhLY8QGGoV3+Zdu9nTx63tvvaBikPwtZuZc76ipWb5Kt6SA80fsN/Q0tDf1m
Bwj1T6UvoivgK8C7iIrwQGUCAWAAQ==
-----END PUBLIC KEY-----

```

Sample output:

```
valid
```

Sample input (tampered message):

```

Tampered message
28ce01ec0f8511e92407496b7aa777a8b38fbc27a3656c9862830427bf8a006d59d19ac3b3441f174aee2fe6560b
2b5f91aeb72ffa150dad4bd4e93a3b78eabfb48f8051b09cdfc5cbb63b014a7beb81cae4220fa01bedf5fa10aa3b
6dcdd7ad229fd7ce2e8883a4893bef532c7b7822993408577a20a198387e073e15e53ffab8d7f3c46fa50347cd45
d7a5d319b66d4ead658297a2e7127cc6a1dc5e104beb3ae2ba390d5ae90a66bd8a537e4df296c212186754206c5a
0b155c226a3a580e943db33bf9e9829ea3e38f00aafa0e4839bf4673d83b14c74cd9ac359afc3fa56ca7f3a8391e
6fb234561b9deca9ac3fa9147d70b0f1189db734b5ea96f287c604467dab0efbf5f1c78437920283dfe6fbc56f4
e859e24ee37552b3b8de5c4fa35e65cf9fe63ec7e2fd155d503fec2648303e495d86af97ac53c14b203bdc723336
5d481fda08cac905c48e31bd389bea0f983e035225c4fca269a66009332bbba83542dcd484832f4d40ab14eee360
9aa78c01539984758592de3355b528de3c8a669442cbd6f03808226ef0c13924b9553b7af7ff02a6916f106061a7
bcbe093f795d9455709ab2b030ff485e9993eb978ae57cabeade4747e981b47c36128c39df970d1f29c4e4a3c225
a8384d55995599c9ce3afe14068f4eb4edeb120beb534955b519a4424f82136d3511a0dafad46b010c3fa15bd233
bdc85a7fd42a
-----BEGIN PUBLIC KEY-----
MIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICGKCAgEAOyjbzkmMss6BU/eCBYhs
3bFRBVldM9DlaM7WhPZEgMY17C10b0kmEPG20/GaNoB0nwptHykzvpYS42rr/D8I
0LeJ/2dz0uStQv6kQDIDmNRTthW9spcuBsqW3e1bRyf0JrL8ot1X8M4k0NGU7w50
8htPzRZXR52Zonv8G/N35ePDV5XF5mrcOPOebconndXHRqfLlyXCxCS5x0q1JGX1

```

```
d/oe2Xop7VfBP/UhrrmNmstZsBEvL3f33+a482xuCTvPIQcv62N31aR2wK0JN9Cp
FOpqjwR8cj9oLjbCRhd5T/+bgc6D0/XVzr9DAPVaYLZvm8puMhhgckNIjgPOfWsV
yI+/G1zuUH5caFMhMaS95M2a1XNIwJ5sF25PFpsttIrFXC+b0JS325mk5qvzCgnu
0DrWT2VPBNjw4rvmhBPnFb/JdXy5e5ZIpXtGRpvKYcBxDk6yqgt9SC/e81+tqCrF
SQ9cG1QCV/BnZw4cd1JHe3/AHot1Yewjb8YpIFifJ7WxFb0g9GIYJZTh3dC9hrNO
fSsIoOL+ZAyNFp43u0FJT7qtC7CtxPvJnt/h8ACE07ewgkq3M0ZmT2RNIf0HhyS
MjwdIh1Y8QGGoV3+zDu9nTx63tvvaBikPwtZuCz76ipWb5Kt6SA80fsN/Q0tDf1m
Bwj1T6UvoivgK8C7iIrwQGUCAwEAAQ==
-----END PUBLIC KEY-----
```

Sample output:

```
invalid
```

ECDSA: Elliptic Curve Digital Signatures

The **ECDSA** (Elliptic Curve Digital Signature Algorithm) is a cryptographically secure **digital signature scheme**, based on the elliptic-curve cryptography (**ECC**). **ECDSA** relies on the math of the **cyclic groups of elliptic curves over finite fields** and on the difficulty of the **ECDLP problem** (elliptic-curve discrete logarithm problem). The **ECDSA sign / verify** algorithm relies on EC point multiplication and works as described below. ECDSA keys and signatures are shorter than in RSA for the same security level. A 256-bit ECDSA signature has the same security strength like 3072-bit RSA signature.

ECDSA uses **cryptographic elliptic curves (EC)** over finite fields in the classical Weierstrass form. These curves are described by their **EC domain parameters**, specified by various cryptographic standards such as **SECG: SEC 2** and **Brainpool (RFC 5639)**. Elliptic curves, used in cryptography, define:

- **Generator point G** , used for scalar multiplication on the curve (multiply integer by EC point)
- **Order n** of the subgroup of EC points, generated by G , which defines the length of the private keys (e.g. 256 bits)

For example, the 256-bit elliptic curve `secp256k1` has:

- Order $n = 115792089237316195423570985008687907852837564279074904382605163141518161494337$ (prime number)
- Generator point $G \{x = 55066263022277343669578718895168534326250603453777594175500187360389116729240, y = 32670510020758816978083085130507043184471273380659243275938904335757337482424\}$

Key Generation

The **ECDSA key-pair** consists of:

- **private key** (integer): $privKey$
- **public key** (EC point): $pubKey = privKey * G$

The **private key** is generated as a **random integer** in the range $[0..n-1]$. The public key $pubKey$ is a point on the elliptic curve, calculated by the EC point multiplication: $pubKey = privKey * G$ (the private key, multiplied by the generator point G).

The public key EC point $\{x, y\}$ can be **compressed** to just one of the coordinates + 1 bit (parity). For the `secp256k1` curve, the private key is 256-bit integer (32 bytes) and the compressed public key is 257-bit integer (~ 33 bytes).

ECDSA Sign

The ECDSA signing algorithm (**RFC 6979**) takes as input a message msg + a private key $privKey$ and produces as output a **signature**, which consists of pair of integers $\{r, s\}$. The **ECDSA signing** algorithm is based on the **ElGamal signature scheme** and works as follows (with minor simplifications):

1. Calculate the message **hash**, using a cryptographic hash function like SHA-256: $h = \text{hash}(msg)$
2. Generate securely a **random** number k in the range $[1..n-1]$
 - In case of **deterministic-ECDSA**, the value k is HMAC-derived from $h + privKey$ (see **RFC 6979**)
3. Calculate the random point $R = k * G$ and take its x-coordinate: $r = R.x$
4. Calculate the signature proof: $s = k^{-1} * (h + r * privKey) \pmod{n}$
 - The modular inverse $k^{-1} \pmod{n}$ is an integer, such that $k * k^{-1} \equiv 1 \pmod{n}$
5. Return the **signature** $\{r, s\}$.

The calculated **signature** $\{r, s\}$ is a pair of integers, each in the range $[1 \dots n-1]$. It encodes the random point $R = k * G$, along with a proof s , confirming that the signer knows the message h and the private key $privKey$. The proof s is by idea verifiable using the corresponding $pubKey$.

ECDSA signatures are **2 times longer** than the signer's **private key** for the curve used during the signing process. For example, for 256-bit elliptic curves (like `secp256k1`) the ECDSA signature is 512 bits (64 bytes) and for 521-bit curves (like `secp521r1`) the signature is 1042 bits.

ECDSA Verify Signature

The algorithm to **verify a ECDSA signature** takes as input the signed message msg + the signature $\{r, s\}$ produced from the signing algorithm + the public key $pubKey$, corresponding to the signer's private key. The output is boolean value: **valid** or **invalid** signature. The **ECDSA signature verify** algorithm works as follows (with minor simplifications):

1. Calculate the message **hash**, with the same cryptographic hash function used during the signing: $h = \text{hash}(msg)$
2. Calculate the modular inverse of the signature proof: $s1 = s^{-1}(\text{mod } n)$
3. Recover the random point used during the signing: $R' = (h * s1) * G + (r * s1) * pubKey$
4. Take from R' its x-coordinate: $r' = R'.x$
5. Calculate the signature validation **result** by comparing whether $r' == r$

The general idea of the signature verification is to **recover the point R'** using the public key and check whether it is same point R , generated randomly during the signing process.

How Does it Work?

The **ECDSA signature** $\{r, s\}$ has the following simple explanation:

- The signing **signing** encodes a random point R (represented by its x-coordinate only) through elliptic-curve transformations using the private key $privKey$ and the message hash h into a number s , which is the **proof** that the message signer knows the private key $privKey$. The signature $\{r, s\}$ cannot reveal the private key due to the difficulty of the **ECDLP problem**.
- The **signature verification** decodes the proof number s from the signature back to its original point R , using the public key $pubKey$ and the message hash h and compares the x-coordinate of the recovered R with the r value from the signature.

The Math behind the ECDSA Sign / Verify

Read this section **only if you like math**. Most developer may skip it.

How does the above sign / verify scheme work? It is not obvious, but let's play a bit with the equations.

The equation behind the recovering of the point R' , calculated during the **signature verification**, can be transformed by replacing the $pubKey$ with $privKey * G$ as follows:

$$\begin{aligned} R' &= (h * s1) * G + (r * s1) * pubKey = \\ &= (h * s1) * G + (r * s1) * privKey * G = \\ &= (h + r * privKey) * s1 * G \end{aligned}$$

If we take the number $s = k^{-1} * (h + r * privKey)(\text{mod } n)$, calculated during the signing process, we can calculate $s1 = s^{-1}(\text{mod } n)$ like this:

$$\begin{aligned}
\mathbf{s1} &= s^{-1}(\text{mod } n) = \\
&= (k^{-1} * (h + r * \text{privKey}))^{-1}(\text{mod } n) = \\
&= k * (h + r * \text{privKey})^{-1}(\text{mod } n)
\end{aligned}$$

Now, replace $\mathbf{s1}$ in the point \mathbf{R}' .

$$\begin{aligned}
\mathbf{R}' &= (h + r * \text{privKey}) * \mathbf{s1} * \mathbf{G} = \\
&= (h + r * \text{privKey}) * k * (h + r * \text{privKey})^{-1}(\text{mod } n) * \mathbf{G} = \\
&= k * \mathbf{G}
\end{aligned}$$

The final step is to **compare** the **point \mathbf{R}'** (decoded by the **pubKey**) with the **point \mathbf{R}** (encoded by the **privKey**). The algorithm in fact compares only the x-coordinates of \mathbf{R}' and \mathbf{R} : the integers r' and r .

It is expected that $r' = r$ if the signature is **valid** and $r' \neq r$ if the signature or the message or the public key is incorrect.

ECDSA: Public Key Recovery from Signature

It is important to know that the **ECDSA signature scheme** allows the **public key to be recovered** from the signed **message** together with the **signature**. The recovery process is based on some **mathematical computations** (described in the **SECG: SEC 1** standard) and returns 0, 1 or 2 possible EC points that are valid **public keys**, corresponding to the signature. To avoid this ambiguity, some ECDSA implementations add one additional bit v to the signature during the signing process and it takes the form $\{r, s, v\}$. From this extended ECDSA signature $\{r, s, v\}$ + the signed **message**, the signer's public key can be restored with confidence.

The **public key recovery from the ECDSA signature** is very useful in bandwidth constrained or storage constrained environments (such as blockchain systems), when transmission or storage of the public keys cannot be afforded. For example, the Ethereum blockchain uses extended signatures $\{r, s, v\}$ for the signed transactions on the chain to save storage and bandwidth.

Public key recovery is possible for signatures, based on the **EIGamal signature scheme** (such as DSA and ECDSA).

Sign / Verify Messages using ECDSA - Examples in Python

After we explained in details how the **ECDSA signature** algorithm works, now let's demonstrate it in practice with **code examples**.

In this example, we shall use the `pycoin` Python package, which implements the **ECDSA signature algorithm** with the curve `secp256k1` (used in the Bitcoin cryptography), as well as many other functionalities related to the Bitcoin blockchain:

```
pip install pycoin
```

ECDSA Sign / Verify using the secp256k1 Curve and SHA3-256

First, define the functions for **hashing**, **ECDSA signing** and **ECDSA signature verification**:

```
from pycoin.ecdsa import generator_secp256k1, sign, verify
import hashlib, secrets

def sha3_256Hash(msg):
    hashBytes = hashlib.sha3_256(msg.encode("utf8")).digest()
    return int.from_bytes(hashBytes, byteorder="big")

def signECDSAsecp256k1(msg, privKey):
    msgHash = sha3_256Hash(msg)
    signature = sign(generator_secp256k1, privKey, msgHash)
    return signature

def verifyECDSAsecp256k1(msg, signature, pubKey):
    msgHash = sha3_256Hash(msg)
    valid = verify(generator_secp256k1, pubKey, msgHash, signature)
    return valid
```

The hashing function `sha3_256Hash(msg)` computes and returns a **SHA3-256** hash, represented as 256-bit integer number. It will be used in the sign / verify processes later.

The `signECDSAsecp256k1(msg, privKey)` function takes a text **message** and 256-bit `secp256k1` **private key** and calculates the ECDSA **signature** $\{r, s\}$ and returns it as pair of 256-bit integers. The ECDSA signature, generated by the `pycoin` library by default is **deterministic**, as described in [RFC 6979](#).

The `verifyECDSAsecp256k1(msg, signature, pubKey)` function takes a text **message**, a ECDSA **signature** $\{r, s\}$ and a 2*256-bit ECDSA **public key** (uncompressed) and returns whether the signature is **valid** or not.

Now let's demonstrate the above defined functions to **sign** a message and **verify** its signature:

```
# ECDSA sign message (using the curve secp256k1 + SHA3-256)
msg = "Message for ECDSA signing"
privKey = secrets.randbelow(generator_secp256k1.order())
signature = signECDSAsecp256k1(msg, privKey)
print("Message:", msg)
print("Private key:", hex(privKey))
print("Signature: r=" + hex(signature[0]) + ", s=" + hex(signature[1]))
```

```
# ECDSA verify signature (using the curve secp256k1 + SHA3-256)
pubKey = (generator_secp256k1 * privKey).pair()
valid = verifyECDSAsecp256k1(msg, signature, pubKey)
print("\nMessage:", msg)
print("Public key: (" + hex(pubKey[0]) + ", " + hex(pubKey[1]) + ")")
print("Signature valid?", valid)

# ECDSA verify tampered signature (using the curve secp256k1 + SHA3-256)
msg = "Tampered message"
valid = verifyECDSAsecp256k1(msg, signature, pubKey)
print("\nMessage:", msg)
print("Signature (tampered msg) valid?", valid)
```

The **output** from the above code is like this:

```
Message: Message for ECDSA signing
Private key: 0x79afbf7147841fca72b45a1978dd7669470ba67abbe5c220062924380c9c364b
Signature: r=0xb83380f6e1d09411ebf49afd1a95c738686bfb2b0fe2391134f4ae3d6d77b78a, s=0x6c305af
cac930a3ea1721c04d8a1a979016baae011319746323a756fbaee1811

Message: Message for ECDSA signing
Public key: (0x3804a19f2437f7bba4fcfbc194379e43e514aa98073db3528ccdbb642e240, 0x6b22d833b9a
502b0e10e58aac485aa357bccd1df6ec0fa4d398908c1ac1920bc)
Signature valid? True

Message: Tampered message
Signature (tampered msg) valid? False
```

As it is visible from the above output, the random generated **secp256k1 private key** is **64 hex digits** (256 bits). After signing, the obtained signature $\{r, s\}$ consists of $2 * 256$ -bit integers. The **public key**, obtained by multiplying the private key by the curve generator point, consists of $2 * 256$ bits (uncompressed). The produced ECDSA digital signature verifies correctly after signing. If the message is tampered, the signature fails to verify.

Public Key Recovery from the ECDSA Signature

As we already know, in ECDSA it is possible to **recover the public key from signature**. Let's demonstrate this by adding the following code at the end of the previous example:

```
from pycoin.ecdsa import possible_public_pairs_for_signature

def recoverPubKeyFromSignature(msg, signature):
    msgHash = sha3_256Hash(msg)
    recoveredPubKeys = possible_public_pairs_for_signature(
        generator_secp256k1, msgHash, signature)
    return recoveredPubKeys

msg = "Message for ECDSA signing"
recoveredPubKeys = recoverPubKeyFromSignature(msg, signature)
print("\nMessage:", msg)
print("Signature: r=" + hex(signature[0]) + ", s=" + hex(signature[1]))
for pk in recoveredPubKeys:
    print("Recovered public key from signature: (" +
        hex(pk[0]) + ", " + hex(pk[1]) + ")")
```

The above code recovers the all **possible EC public keys** from the ECDSA **signature** + the signed **message**, using the algorithm, described in <http://www.secg.org/sec1-v2.pdf>. Note that **multiple EC public keys** (0, 1 or 2) may match the message + signature. The expected output from the above code (together with the previous code) looks like this:

```

Message: Message for ECDSA signing
Private key: 0xc374556584db05001c2c9265b546e66d3dbbe8239d17427c176d834a19638dc
Signature: r=0xd034c98af3274ad93f3c8ce944bbc17b11b6aa170c5f097ed98687fa0d93347c, s=0xa2318cea2002caba38efbba3bf8ef8d43236a6edc33c040734d8eb2ed77f608

Message: Message for ECDSA signing
Public key: (0x10b5d9028ec828a0f9111e36f046afa5a0c677357351093426bcec10c663db7d, 0x271763c56fcd87b72d59ceaa5b9c3fd2122788fe344751a9bde373f903e5bb20)
Signature valid? True

Message: Tampered message
Signature (tampered msg) valid? False

Message: Message for ECDSA signing
Signature: r=0xd034c98af3274ad93f3c8ce944bbc17b11b6aa170c5f097ed98687fa0d93347c, s=0xa2318cea2002caba38efbba3bf8ef8d43236a6edc33c040734d8eb2ed77f608
Recovered public key from signature: (0x1353fd26a6cb6110980cfd2bb5eca3b3cc3e08c930ad5991395d826a250c79, 0xba6825142e230ee1fa2b406f3f9158a47ee49daca8ac47898c5fd92d805a101e)
Recovered public key from signature: (0x10b5d9028ec828a0f9111e36f046afa5a0c677357351093426bcec10c663db7d, 0x271763c56fcd87b72d59ceaa5b9c3fd2122788fe344751a9bde373f903e5bb20)

```

It is obvious that the **recovered possible public keys** are 2: one is equal to the public key, matching the signer's private key, and the other is not (it matches the math behind the public key recovery, but is not the correct one). To avoid this ambiguity, **the signature can be extended** to hold $\{r, s, v\}$, where v holds the parity of the y coordinate of the random point R from the ECDSA signing algorithm. This coordinate is lost, because the ECDSA signature takes just the x coordinate or R .

Public Key Recovery from Extended ECDSA Signature

To **recover with confidence the public key** from ECDSA signature + message, we need a library that generates **extended ECDSA signatures** $\{r, s, v\}$ and supports internally the public key recovery. Let's play with the `eth_keys` Python library:

```
pip install eth_keys
```

The `eth_keys` is part of the Ethereum project and implements **secp256k1**-based ECC cryptography, private and public keys, ECDSA extended signatures $\{r, s, v\}$ and Ethereum blockchain addresses. The following example demonstrates private key generation, message signing, public key recovery from signature + message and signature verification:

```

import eth_keys, os

# Generate the private + public key pair (using the secp256k1 curve)
signerPrivKey = eth_keys.keys.PrivateKey(os.urandom(32))
signerPubKey = signerPrivKey.public_key
print('Private key (64 hex digits):', signerPrivKey)
print('Public key (uncompressed, 128 hex digits):', signerPubKey)

```

```
# ECDSA sign message (using the curve secp256k1 + Keccak-256)
msg = b'Message for signing'
signature = signerPrivKey.sign_msg(msg)
print('Message:', msg)
print('Signature: [r = {0}, s = {1}, v = {2}]'.format(
    hex(signature.r), hex(signature.s), hex(signature.v)))

# ECDSA public key recovery from signature + verify signature
# (using the curve secp256k1 + Keccak-256 hash)
msg = b'Message for signing'
recoveredPubKey = signature.recover_public_key_from_msg(msg)
print('Recovered public key (128 hex digits):', recoveredPubKey)
print('Public key correct?', recoveredPubKey == signerPubKey)
valid = signerPubKey.verify_msg(msg, signature)
print("Signature valid?", valid)
```

The output from the above code looks like this:

```
Private key (64 hex digits): 0x68abc765746a33272e47b0a96a0b4184048f70354221e04219fbc223bfe79
794
Public key (uncompressed, 128 hex digits): 0x30a6dc572da312587144e7ccda1e9abd901323adebe7091
bb4985e1202c2a10bc25f681b3d2e1a671438f0b125287b473c09ca345c5583cd627232b536b9ca0a
Message: b'Message for signing'
Signature: [r = 0x4cddf146c578d20a31fa6128e5d9afe6ac666e5ef5899f2767cacb56a42703cc, s = 0x38
47036857aa3f077a2e142eee707e5af2653baa99b9d10764a0be3d61595dbb, v = 0x0]
Recovered public key (128 hex digits): 0x30a6dc572da312587144e7ccda1e9abd901323adebe7091bb49
85e1202c2a10bc25f681b3d2e1a671438f0b125287b473c09ca345c5583cd627232b536b9ca0a
Public key correct? True
Signature valid? True
```

The **public key recovery** will always be successful, because there is no ambiguity with the **extended ECDSA signature**. The **signature verification** will be successful, unless the message, the public key or the signature is tampered. You are free to play with the above code, to change it, to tamper the signed message and to see what happens. Enjoy!

Exercises: Sign / Verify Messages using ECDSA and the NIST P-521 Curve

In this exercise we shall **sign** and **verify** messages using the **ECDSA** digital signature algorithm and the **NIST P-521** curve. The NIST P-521 elliptic curve, known also as `secp521r1` is 521-bit ECC curve, suitable for ECDSA digital signatures and ECDH key agreement. It uses **521-bit private keys** (encoded as 65-66 bytes, 130-132 hex digits) and **1042-bit public keys** (uncompressed, encoded as 130-131 bytes, 260-261 hex digits). The produced **signature** is 132 bytes (264 hex digits).

Sign a Message with ECDSA / P-521

Write a program to **sign a message** by given **private key**. The **input** consists of 2 text lines: message and private key. The message is given as **text** and the private key is given as **hex** string (130-132 hex digits). Use the **ECDSA deterministic signing** (following [RFC 6979](#)) and the curve **NIST P-521**, which also known as `secp521r1`. Print the **output** as JSON document, holding the input **message** + the **public key** of the signer (as hex string, uncompressed) + the ECDSA **digital signature** (as hex string).

Sample input:

```
Message for ECDSA-NIST-521p signing
00135799f9d1f033af26168780bf2503313acff854c44031321d7a29bba96edb3c1b93b9deea55229b1de058196a
d69a79c01463e3281d9fcc82afd73aac7fdfa4af
```

Sample output:

```
{
  "msg": "Message for ECDSA-NIST-521p signing",
  "pubKey": "0078a6bb6732cb3134d2ca3912b2876fe005b20027037512cf972605f58ce5908471a1f9817c8d24
290fcc943951f3113a7ee3716bd95f0b9c7326843a833ac6a0750021f08f88a6bd397525068300801521d2d97fea
32f2c8b0c74dc8e231a4dd73252c4a7398e25ab20dba0a9df3df0c256617e004a9623676b9f3f9a3aa21f57c90ce
",
  "signature": "00202029ab1a3326fe7d1e9ec36d7fab048e833c6c3cad37e1d5294695d28e9fd5583c23edaee
cb596782a4c85bac27780623c1a9216202f3828991cbeebbeb049d9008270ea623d8d26c5ab89b621bac12c7fa8e
9193e4057e16617f80cfc4279537f45169fb949deb3f9936400a130f6859aaa9c929e47c66610e59cc71a9f7ea79
e81"
}
```

Verify Message Signature with ECDSA / P-521

Write a program to **validate the ECDSA digital signature**, created by the previous exercise. The **input** comes as JSON document, holding the **message** + the **public key** (uncompressed, hex string) + the **signature**. Use the P-521 elliptic curve (`secp521r1`). Print as **output** a single word: **"valid"** or **"invalid"**.

Sample input (correctly signed message):

```
{
  "msg": "Message for ECDSA-NIST-521p signing",
  "pubKey": "0078a6bb6732cb3134d2ca3912b2876fe005b20027037512cf972605f58ce5908471a1f9817c8d24
290fcc943951f3113a7ee3716bd95f0b9c7326843a833ac6a0750021f08f88a6bd397525068300801521d2d97fea
32f2c8b0c74dc8e231a4dd73252c4a7398e25ab20dba0a9df3df0c256617e004a9623676b9f3f9a3aa21f57c90ce
",
  "signature": "00202029ab1a3326fe7d1e9ec36d7fab048e833c6c3cad37e1d5294695d28e9fd5583c23edaee
cb596782a4c85bac27780623c1a9216202f3828991cbeebbeb049d9008270ea623d8d26c5ab89b621bac12c7fa8e
"
```

```
9193e4057e16617f80cfc4279537f45169fb949deb3f9936400a130f6859aaa9c929e47c66610e59cc71a9f7ea79e81"
}
```

Sample output:

```
valid
```

Sample input (tampered message):

```
{
  "msg": "Tampered message",
  "pubKey": "0078a6bb6732cb3134d2ca3912b2876fe005b20027037512cf972605f58ce5908471a1f9817c8d24290fcc943951f3113a7ee3716bd95f0b9c7326843a833ac6a0750021f08f88a6bd397525068300801521d2d97fea32f2c8b0c74dc8e231a4dd73252c4a7398e25ab20dba0a9df3df0c256617e004a9623676b9f3f9a3aa21f57c90ce",
  "signature": "00202029ab1a3326fe7d1e9ec36d7fab048e833c6c3cad37e1d5294695d28e9fd5583c23edaeecb596782a4c85bac27780623c1a9216202f3828991cbeebbeb049d9008270ea623d8d26c5ab89b621bac12c7fa8e9193e4057e16617f80cfc4279537f45169fb949deb3f9936400a130f6859aaa9c929e47c66610e59cc71a9f7ea79e81"
}
```

Sample output:

```
invalid
```

EdDSA and Ed25519: Elliptic Curve Digital Signatures

EdDSA (Edwards-curve Digital Signature Algorithm) is a modern and secure digital signature algorithm based on performance-optimized elliptic curves, such as the 255-bit curve [Curve25519](#) and the 448-bit curve [Curve448-Goldilocks](#). The EdDSA signatures use the **Edwards form** of the elliptic curves (for performance reasons), respectively `edwards25519` and `edwards448`. The **EdDSA** algorithm is based on the [Schnorr signature algorithm](#) and relies on the difficulty of the **ECDLP problem**.

The **EdDSA** signature algorithm and its variants **Ed25519** and **Ed448** are technically described in the [RFC 8032](#).

EdDSA Key Generation

Ed25519 and **Ed448** use small **private keys** (32 or 57 bytes respectively), small **public keys** (32 or 57 bytes) and **small signatures** (64 or 114 bytes) with **high security level** at the same time (128-bit or 224-bit respectively).

Assume the elliptic curve for the EdDSA algorithm comes with a generator point **G** and a subgroup order **q** for the EC points, generated from **G**.

The **EdDSA key-pair** consists of:

- **private key** (integer): *privKey*
- **public key** (EC point): *pubKey* = *privKey* * **G**

The **private key** is generated from a **random integer**, known as **seed** (which should have similar bit length, like the curve order). The **seed** is first hashed, then the last few bits, corresponding to the curve **cofactor** (8 for Ed25519 and 4 for X448) are cleared, then the highest bit is cleared and the second highest bit is set. These transformations guarantee that the private key will always belong to the same subgroup of EC points on the curve and that the private keys will always have similar bit length (to protect from timing-based side-channel attacks). For **Ed25519** the private key is 32 bytes. For **Ed448** the private key is 57 bytes.

The public key *pubKey* is a point on the elliptic curve, calculated by the EC point multiplication: *pubKey* = *privKey* * **G** (the private key, multiplied by the generator point **G** for the curve). The public key is encoded as **compressed** EC point: the **y**-coordinate, combined with the lowest bit (the parity) of the **x**-coordinate. For **Ed25519** the public key is 32 bytes. For **Ed448** the public key is 57 bytes.

EdDSA Sign

The **EdDSA signing** algorithm ([RFC 8032](#)) takes as input a text message *msg* + the signer's EdDSA **private key** *privKey* and produces as output a pair of integers **{R, s}**. EdDSA signing works as follows (with minor simplifications):

```
EdDSA_sign(msg, privKey) --> { R, s }
```

1. Calculate *pubKey* = *privKey* * **G**
2. Deterministically generate a secret integer $r = \text{hash}(\text{hash}(\text{privKey}) + \text{msg}) \bmod q$ (this is a bit simplified)
3. Calculate the public key point behind *r* by multiplying it by the curve generator: $R = r * G$
4. Calculate $h = \text{hash}(R + \text{pubKey} + \text{msg}) \bmod q$
5. Calculate $s = (r + h * \text{privKey}) \bmod q$
6. Return the **signature** **{ R, s }**

The produced **digital signature** is 64 bytes (32 + 32 bytes) for **Ed25519** and 114 bytes (57 + 57 bytes) for **Ed448**. It holds a compressed point **R** + the integer **s** (confirming that the signer knows the *msg* and the *privKey*).

EdDSA Verify Signature

The **EdDSA signature verification** algorithm ([RFC 8032](#)) takes as input a text message **msg** + the signer's EdDSA **public key pubKey** + the EdDSA signature **{R, s}** and produces as output a boolean value (valid or invalid signature). EdDSA verification works as follows (with minor simplifications):

```
EdDSA_signature_verify(msg, pubKey, signature { R, s } ) --> valid / invalid
```

1. Calculate $h = \text{hash}(R + \text{pubKey} + \text{msg}) \bmod q$
2. Calculate $P1 = s * G$
3. Calculate $P2 = R + h * \text{pubKey}$
4. Return $P1 == P2$

How Does it Work?

During the verification the point **P1** is calculated as: $P1 = s * G$.

During the signing $s = (r + h * \text{privKey}) \bmod q$. Now replace **s** in the above equation:

- $P1 = s * G = (r + h * \text{privKey}) \bmod q * G = r * G + h * \text{privKey} * G = R + h * \text{pubKey}$

The above is exactly the other point **P2**. If these points **P1** and **P2** are the same EC point, this proves that the point **P1**, calculated by the private key matches the point **P2**, created by its corresponding public key.

ECDSA vs EdDSA

If we compare the signing and verification for EdDSA, we shall find that **EdDSA is simpler than ECDSA**, easier to understand and to implement. Both signature algorithms have **similar security strength** for curves with similar key lengths. For the most popular curves (liked `edwards25519` and `edwards448`) the **EdDSA algorithm is slightly faster than ECDSA**, but this highly depends on the curves used and on the certain implementation. Unlike ECDSA the EdDSA signatures do not provide a way to **recover the signer's public key** from the signature and the message. Generally, it is considered that EdDSA is recommended for most modern apps.

Sign / Verify Messages using EdDSA - Examples in Python

After we explained in the previous section how the **EdDSA signatures** work, now it is time to demonstrate them with code examples. First, we shall demonstrated how to use Ed25519 signatures.

Ed25519 Signatures - Example

We shall use the Python library `ed25519`, which is based on the Bernstein's original optimized highly optimized C implementation of the **Ed25519** signature algorithm (EdDSA over the Curve25519 in Edwards form):

```
pip install ed25519
```

Next, generate a **private + public key pair** for the Ed25519 cryptosystem, **sign** a sample message, and **verify** the signature:

```
import ed25519

privKey, pubKey = ed25519.create_keypair()
print("Private key (32 bytes):", privKey.to_ascii(encoding='hex'))
print("Public key (32 bytes): ", pubKey.to_ascii(encoding='hex'))

msg = b'Message for Ed25519 signing'
signature = privKey.sign(msg, encoding='hex')
print("Signature (64 bytes):", signature)

try:
    pubKey.verify(signature, msg, encoding='hex')
    print("The signature is valid.")
except:
    print("Invalid signature!")
```

The output from the above sample code looks like this:

```
Private key (32 bytes): b'1498b5467a63dfffa2dc9d9e069caf075d16fc33fdd4c3b01bfadae6433767d93'
Public key (32 bytes): b'b7a3c12dc0c8c748ab07525b701122b88bd78f600c76342d27f25e5f92444cde'
Signature (64 bytes): b'6dd355667fae4eb43c6e0ab92e870edb2de0a88cae12dbd8591507f584fe4912babf
f497f1b8edf9567d2483d54ddc6459bea7855281b7a246a609e3001a4e08'
The signature is valid.
```

The **Ed25519 key pair** is generated randomly: first a 32-byte random seed is generated, then the private key is derived from the seed, then the public key is derived from the private key. The hash function for key generation is SHA-512.

The **private key** is encoded as 64 hex digits (32 bytes). The **public key** is encoded also as 64 hex digits (32 bytes). The EdDSA-Ed25519 **signature** $\{R, s\}$ is 32 + 32 bytes (64 bytes, 128 hex digits).

If we try to verify a tampered message, the verification will fail:

```
try:
    pubKey.verify(signature, "Tampered msg", encoding='hex')
    print("The signature is valid.")
except:
```

```
print("Invalid signature!")
```

The output from the above sample code is as expected:

```
Invalid signature!
```

Ed448 Signatures - Example

Now, let's demonstrate how to use the **Ed448 signature** (EdDSA over the Curve448-Goldilocks curve in Edwards form).

We shall use the Python elliptic curve library `ecpy`, which implements ECC with Weierstrass curves (like `secp256k1` and `NIST P-256`), Montgomery curves (like `Curve25519` and `Curve448`) and twisted Edwards curves (like `Ed25519` and `Ed448`):

```
pip install ecpy
```

Next, generate a **private + public key pair** for the Ed448 cryptosystem:

```
from ecpy.curves import Curve
from ecpy.keys import ECPrivateKey
from ecpy.eddsa import EDDSA
import secrets, hashlib, binascii

curve = Curve.get_curve('Ed448')
signer = EDDSA(hashlib.shake_256, hash_len=114)
privKey = ECPrivateKey(secrets.randbits(57*8), curve)
pubKey = signer.get_public_key(privKey, hashlib.shake_256, hash_len=114)
print("Private key (57 bytes):", privKey)
print("Public key (compressed, 57 bytes): ",
      binascii.hexlify(curve.encode_point(pubKey.W)))
print("Public key (point): ", pubKey)
```

The **Ed448 key pair** is generated randomly. According to [RFC 8032](#) the Ed448 **private key** is generated from 57-byte random seed, which is transformed to 57-byte **public key** using the **SHAKE256**(x , $\text{hash_len}=114$) hash function, along with EC point multiplication and the special key encoding rules for Ed448.

The output from the above sample code may look like this:

```
Private key (57 bytes): ECPrivateKey:
  d: 625d3edeb5cd69b20b0b6387c3522a21d356ac40b408e34fb2f8442e2c91eee3f877afe583a2fd11770567d
  f69178019d6fbc6357c35eefa3e
Public key (compressed, 57 bytes): b'261d23911e194ed0cb7f9233568e906d6abc4d60f73451ca80763
  6d8fa6e4ea5ca12f51d240299a0b86a61ccb2174ce4ed2a8c4f7a8cced00'
Public key (point): ECPublicKey:
  x: cb5aec366d6b3293354418f8abf67bd5aaf46b49ff9c2154fbc14d9ca22fe93b680954f27c10fed3327ef51
  c8bce5d2522f41fd554731d88
  y: edcca8f7c4a8d24ece7421cb1ca6860b9a2940d2512fa15ceae4a68f6d6307a81c45730fd6f4bc6a6d908e5
  633927fcbd04e191e91231d26
```

The **private key** is encoded as 114 hex digits (57 bytes). The **public key** is encoded also as 114 hex digits (57 bytes), in compressed form. In the above example the public key EC point is printed also in uncompressed format (x and y coordinates). The EdDSA-Ed448 **signature** $\{R, s\}$ consists of 57 + 57 bytes (114 bytes, 228 hex digits).

Next, **sign** a sample message using the private key, and **verify** the signature using the public key after that:

```
msg = b'Message for Ed448 signing'
signature = signer.sign(msg, privKey)
print("Signature (114 bytes):", binascii.hexlify(signature))

valid = signer.verify(msg, signature, pubKey)
print("Valid signature?", valid)
```

The output from the above code example (for the above Ed448 key pair) is:

```
Signature (114 bytes): b'5114674f1ce8a2615f2b15138944e5c58511804d72a96260ce8c587e7220daa90b9
e65b450ff49563744d7633b43a78b8dc6ec3e3397b50080a15f06ce8005ad817a1681a4e96ee6b4831679ef448d7
c283b188ed64d399d6bac420fadf33964b2f2e0f2d1abd401e8eb09ab29e3ff280600'
Valid signature? True
```

The signature is **deterministic**: the same message with the same private key produces the same signature.

If we try to verify the same signature with a tampered message, the verification will fail:

```
valid = signer.verify(b'Tampered msg', signature, pubKey)
print("Valid signature?", valid)
```

The output from the above sample code is as expected:

```
Valid signature? False
```

Exercises: Sign / Verify Messages using Ed25519

In this exercise we shall **sign** and **verify** messages using the **EdDSA** digital signature algorithm and the `edwards25519` curve, following the technical specification from [RFC 8032](#). The **Ed25519** digital signature algorithm can be found as library for the most programming languages.

The Ed25519 **private key** is encoded as 64 hex digits (32 bytes). The corresponding Ed25519 **public key** is encoded also as 64 hex digits (32 bytes). The EdDSA-Ed25519 **signature** $\{R, s\}$ consists of 32 + 32 bytes (64 bytes, 128 hex digits).

EdDSA-Ed25519: Sign Message

Write a program to sign given text **message** with given **private key**. The input consists of 2 text lines. The **first line** holds the input **message** for signing. The **second line** holds the private key as **hex string**. Print the **output** as JSON document, holding the input **message** + the **public key** of the signer (as hex string, uncompressed) + the Ed25519 **digital signature** (as hex string).

Sample input:

```
Message for Ed25519 signing
de6d730f36a8607b8bfdaa79b3b1127291f1d50552c2fe05c5254a9719105c4a
```

Sample output:

```
{
  "msg": "Message for Ed25519 signing",
  "pubKey": "7721a5832cb70cce1a960cf236d50a0e862555ccad400b5fee0bcf777f7ab476",
  "signature": "6c4adbba332b5db520c0ec95433ea136f70fe2d50e8955a7049d216626a3491c0e5cbfefb8d779687cc9811311ccaf7cd07a0e96a570fb3a4b680a4ead60c602"
}
```

EdDSA-Ed25519: Verify Signature

Write a program to **validate the Ed25519 digital signature**, created by the previous exercise. The **input** comes as JSON document, holding the **message** + the **public key** (uncompressed, hex string) + the **signature**. Print as **output** a single word: **"valid"** or **"invalid"**.

Sample input (correctly signed message):

```
{
  "msg": "Message for Ed25519 signing",
  "pubKey": "7721a5832cb70cce1a960cf236d50a0e862555ccad400b5fee0bcf777f7ab476",
  "signature": "6c4adbba332b5db520c0ec95433ea136f70fe2d50e8955a7049d216626a3491c0e5cbfefb8d779687cc9811311ccaf7cd07a0e96a570fb3a4b680a4ead60c602"
}
```

Sample output:

```
valid
```

Sample input (tampered message):

```
{
  "msg": "Tampered msg",
```

```
"pubKey": "7721a5832cb70cce1a960cf236d50a0e862555ccad400b5fee0bcf777f7ab476",  
"signature": "6c4adbba332b5db520c0ec95433ea136f70fe2d50e8955a7049d216626a3491c0e5cbfefb8d77  
9687cc9811311ccaf7cd07a0e96a570fb3a4b680a4ead60c602"  
}
```

Sample output:

```
invalid
```

Quantum Computers and Quantum-Safe Cryptography

Quantum computers are ...

TODO

TODO

TODO

It is well known in computer science that **quantum computers will break some cryptographic algorithms**, especially the public-key cryptosystems like **RSA**, **ECC** and **ECDSA** that rely on the **IFP** (integer factorization problem), the **DLP** (discrete logarithms problem) and the **ECDLP** (elliptic-curve discrete logarithm problem). Quantum algorithms will not be the end of cryptography, because:

- Only some cryptosystems are **quantum-unsafe** (like RSA, DHKE, ECC, ECDSA and ECDH).
- Some cryptosystems are **quantum-safe** and will be only slightly affected (like cryptographic hashes, MAC algorithms and symmetric key ciphers).

Let's discuss this in details.

Quantum-Safe and Quantum-Broken Crypto Algorithms

Most cryptographic **hashes** (like SHA2, SHA3, BLAKE2), **MAC** algorithms (like HMAC and CMAK), **key-derivation functions** (bcrypt, Scrypt, Argon2) are basically **quantum-safe** (only slightly affected by quantum computing).

- Use 384-bits or more to be quantum-safe (256-bits should be enough for long time)

Symmetric ciphers (like AES-256, Twofish-256) are **quantum-safe**.

- Use 256-bits or more as key length (don't use 128-bit AES)

Most popular **public-key cryptosystems** (like RSA, DSA, ECDSA, EdDSA, DHKE, ECDH, ElGamal) are **quantum-broken!**

- Most **digital signature** algorithms (like RSA, ECDSA, EdDSA) are **quantum-broken!**
- **Quantum-safe signature** algorithms and public-key cryptosystems are already developed (e.g. lattice-based or hash-based signatures), but are not massively used, because of longer keys and longer signatures than ECC.

See https://en.wikipedia.org/wiki/Post-quantum_cryptography

...

Quantum-Resistant Crypto Algorithms

...

ECC Cryptography and Most Digital Signatures are Quantum-Broken!

...

A **k**-bit number can be factored in time of order **$O(k^3)$** using a quantum computer of **$5k+1$ qubits** (using Shor's algorithm).

- See <http://www.theory.caltech.edu/~preskill/pubs/preskill-1996-networks.pdf>

256-bit number (e.g. Bitcoin public key) can be factorized using 1281 qubits in $72 \cdot 256^3$ quantum operations.

- ~ 1.2 billion operations $\Rightarrow \sim$ less than 1 second using good machine

ECDSA, DSA, RSA, ElGamal cryptosystems are all quantum-broken

Conclusion: publishing the signed transactions (like Ethereum does) is not quantum safe -> avoid revealing the ECC public key

Hashes are Quantum Safe

Cryptographic **hashes** (like SHA2, SHA3, BLAKE2) are considered **quantum-safe**:

- On traditional computer, finding a collision for 256-bit hash takes $\sqrt{2^{256}}$ steps (using the **birthday attack**) -> SHA256 has 2^{128} crypto-strength
- Quantum computers might find hash collisions in $\sqrt[3]{2^{256}}$ operations (see [the BHT algorithm](#)), but this is disputed (see [Bernstein 2009] - <http://cr.yp.to/hash/collisioncost-20090823.pdf>)
- On theory it might take 2^{85} quantum operations to find SHA256 / SHA3-256 collision, but in practice it may cost significantly more.

Conclusion: SHA256 / SHA3-256 are most probably quantum-safe

- SHA384, SHA512 and SHA3-384, SHA3-512 are quantum-safe

...

Symmetric Ciphers are Quantum Safe

...

Most symmetric ciphers (like AES and ChaCha20) are quantum-safe:

- [Grover's algorithm](https://en.wikipedia.org/wiki/Grover's_algorithm) (https://en.wikipedia.org/wiki/Grover's_algorithm) finds AES secret key using $\sqrt{}$ quantum operations
- Quantum era will **double the key size** of the symmetric ciphers (see <http://cr.yp.to/codes/grovercode-20100303.pdf>)

AES-256 in the post-quantum era is like AES-128 before

- 128-bits or less symmetric ciphers are quantum-attackable

Conclusion: 256-bit symmetric ciphers are quantum safe

- AES-256, ChaCha20-256, Twofish-256, Camellia-256 are considered quantum-safe

Post-Quantum Cryptography

...

Quantum-Safe key agreement: <https://en.wikipedia.org/wiki/CECPQ1>

<https://ianix.com/pqcrypto/pqcrypto-deployment.html>

<https://pqcrypto.org/>

Post-quantum signature scheme XMSS:

- <https://tools.ietf.org/html/rfc8391>
- JS XMSS - <https://www.npmjs.com/package/xmss>
- Post-quantum key agreement schemes McEliece and NewHope

Post-quantum signatures and key agreements (XMSS, McEliece, NewHope):

<https://github.com/randombit/botan>

QC-MDPC and libPQC are quantum-broken: <https://eprint.iacr.org/2016/858.pdf>

Hash-Based Public-Key Cryptography

...

Code-Based Public-Key Cryptography

...

Lattice-Based Public-Key Cryptography

...

Multivariate-Quadratic-Equations Public-Key Cryptography

MQE

...

SPHINCS+ Signatures in Python

<https://github.com/sphincs/pyspx>

<https://pypi.org/project/PySPX/>

NewHope Key Exchange in Python

<https://github.com/anupsv/NewHope-Key-Exchange>

<https://github.com/scottwn/PyNewHope>

Quantum-Safe Signatures - Examples in Python

...

TODO

...

More Cryptographic Concepts for Developers

...

Digital Certificates, the X.509 Standard and PKI

...

<https://cryptography.io/en/latest/x509/>

Transport Layer Security (TLS) and SSL

...

https://en.wikipedia.org/wiki/Transport_Layer_Security

External Authorization and OAuth

...

Two-Factor Authentication and One-Time Passwords

<https://cryptography.io/en/latest/hazmat/primitives/twofactor/>

<https://tools.ietf.org/html/rfc4226.html>

Infected Cryptosystems and Crypto Backdoors

<https://en.wikipedia.org/wiki/Kleptography>

X.509 Digital certificates - Examples in Python

...

Generate X.509 self-signed certificate

Dump certificate data from the Python code

...

TLS (Transport Layer Security) - Examples in Python

...

Connect to HTTPS server and download resource.

Display the server certificate + the public key.

Display info about the TLS cipher suite.

...

One-Time Passwords (OTP) - Examples in Python

...

Cryptographic Libraries for JavaScript, Python, C# and Java

- Cryptography in **JavaScript**
 - ECDSA, elliptic.js, js-sha3.js
- Cryptography libraries in **Python**
 - ECDSA, eth_keys
- **C#** and **.NET** cryptography
 - Bouncy Castle .NET, Nethereum
- **Java** cryptography
 - JCA, Bouncy Castle, Web3j
- **C** and **C++** cryptography
 - Crypto++, OpenSSL bindings, Nettle, libgcrypt

TODO:

- **OpenSSL** - <https://en.wikipedia.org/wiki/OpenSSL>
- **LibSodium**
- **Crypto++**
- **Libgcrypt** - <https://en.wikipedia.org/wiki/Libgcrypt>
- **Bouncy Castle**
- **Nettle** - <https://git.lysator.liu.se/nettle/nettle>
- **Others...**

Summary

- **JavaScript** and **Python** provide simple cryptography libraries
 - Hashes, ECC, ECDSA, AES, and many more
- Cryptography in **C#** is heavy
 - Use **Bouncy Castle .NET** for general crypto
 - Or **Nethereum** for simplified secp256k1
- Cryptography in **Java** is heavy
 - **JCA** and **Bouncy Castle** are hard to use
 - **Web3j** is simplified library for secp256k1

...

JavaScript Crypto Libraries

...

Cryptography in JavaScript

- ECDSA with elliptic.js and js-sha3

ECDSA in JavaScript: Generate / Load Keys

```
npm install elliptic
npm install js-sha3
```

...

```
let elliptic = require('elliptic');
let sha3 = require('js-sha3');
let ec = new elliptic.ec('secp256k1');

// let keyPair = ec.genKeyPair(); // Generate random keys
let keyPair = ec.keyFromPrivate(
  "97ddae0f3a25b92268175400149d65d6887b9cefaf28ea2c078e05cdc15a3c0a");
let privKey = keyPair.getPrivate("hex");
let pubKey = keyPair.getPublic();
console.log(`Private key: ${privKey}`);
console.log("Public key :", pubKey.encode("hex").substr(2));
console.log("Public key (compressed):",
  pubKey.encodeCompressed("hex"));
```

ECDSA in JavaScript: Sign Message

```
let msg = 'Message for signing';
let msgHash = sha3.keccak256(msg);
let signature =
  ec.sign(msgHash, privKey, "hex", {canonical: true});

console.log(`Msg: ${msg}`);
console.log(`Msg hash: ${msgHash}`);
console.log("Signature:", signature);
```

Complete example:<https://gist.github.com/nakov/1dcbe26988e18f7a4d013b65d8803ffc>

ECDSA in JavaScript: Verify Signature

```
let hexToDecimal = (x) => ec.keyFromPrivate(x, "hex")
  .getPrivate().toString(10);
let pubKeyRecovered = ec.recoverPubKey(
  hexToDecimal(msgHash), signature,
  signature.recoveryParam, "hex");
console.log("Recovered pubKey:",
  pubKeyRecovered.encodeCompressed("hex"));
let validSig = ec.verify(
```

```
    msgHash, signature, pubKeyRecovered);  
    console.log("Signature valid?", validSig);
```

Complete example:<https://gist.github.com/nakov/1dcbe26988e18f7a4d013b65d8803ffc>

Python Crypto Libraries

...

Cryptography in Python

- Hashes, ECC and ECDSA, eth_keys Library

ECDSA in Python: Generate / Load Keys

```
import eth_keys, eth_utils, binascii, os

# privKey = eth_keys.keys.PrivateKey(os.urandom(32))
privKey = eth_keys.keys.PrivateKey(binascii.unhexlify(
    '97ddae0f3a25b92268175400149d65d6887b9cefaf28ea2c078e05cdc15a3c0a'))
pubKey = privKey.public_key
pubKeyCompressed = '0' + str(2 + int(pubKey) % 2) + str(pubKey)[2:66]
address = pubKey.to_checksum_address()
print('Private key (64 hex digits):', privKey)
print('Public key (plain, 128 hex digits):', pubKey)
print('Public key (compressed):', pubKeyCompressed)
print('Signer address:', address)
```

ECDSA in Python: Sign Message

```
msg = b'Message for signing'
msgHash = eth_utils.keccak(msg)
signature = privKey.sign_msg(msg)

print('Msg:', msg)
print('Msg hash:', binascii.hexlify(msgHash))
print('Signature: [v = {0}, r = {1}, s = {2}]'.format(
    hex(signature.v), hex(signature.r), hex(signature.s)))
print('Signature (130 hex digits):', signature)
```

ECDSA in Python: Verify Signature

```
msg = b'Message for signing'
msgSigner = '0xa44f70834a711f0df388ab016465f2eEb255dEd0'
signature = eth_keys.keys.Signature(binascii.unhexlify(
    '6f0156091cbe912f2d5d1215cc3cd81c0963c8839b93af60e0921b61a19c54300c71006dd93f3508c432dac
a21db0095f4b16542782b7986f48a5d0ae3c583d401'))
signerPubKey = signature.recover_public_key_from_msg(msg)
print('Signer public key (recovered):', signerPubKey)
signerAddress = signerPubKey.to_checksum_address()
print('Signer address:', signerAddress)
print('Signature valid?:', signerAddress == msgSigner)
```


C# Crypto Libraries

...

Cryptography in C# and .NET

- Bouncy Castle .NET and Nethereum:Hashes, ECC and ECDSA

.NET Cryptography and Bouncy Castle .NET

- Cryptography in C# and .NET is based on:
 - The build-in libraries: **System.Security.Cryptography**
 - The **Bouncy Castle .NET**– a powerful C# cryptography library
 - <http://www.bouncycastle.org/csharp>
- **Nethereum** – a simplified library for Ethereum and secp256k1
 - Nethereum – <https://github.com/Nethereum>
 - The cryptographic functionality is in Nethereum.Signer
 - Nethereum also includes the Bouncy Castle .NET library

ECDSA in C#: Initialize the Application

Install the "Nethereum.Signer" package from NuGet

```
dotnet add package Nethereum.Signer
```

Import the **Nethereum Signer** namespaces:

```
using Nethereum.Signer;
using Nethereum.Signer.Crypto;
using Nethereum.Util;
using Nethereum.Hex.HexConvertors.Extensions;
```

The **Bouncy Castle** namespaces will also be available, e.g.

```
Org.BouncyCastle.Math.EC.ECPoint p = ...;
```

ECDSA in C#: Generate / Load Keys

```
// var privKey = EthEKey.GenerateKey(); // Random private key
var privKey = new EthEKey( "97ddae0f3a25b92268175400149d65d6887b9cefaf28ea2c078e05cdc15a3c0a");
byte[] pubKeyCompressed = new EKey(
    privKey.GetPrivateKeyAsBytes(), true).GetPubKey(true);
Console.WriteLine("Private key: {0}",
    privKey.GetPrivateKey().Substring(4));
Console.WriteLine("Public key: {0}",
    privKey.GetPubKey().ToHex().Substring(2));
Console.WriteLine("Public key (compressed): {0}",
    pubKeyCompressed.ToHex());
```

Complete example:<https://gist.github.com/nakov/f2a579eb9893b29338b11e063d6f80c2>

ECDSA in C#: Sign Message

```
string msg = "Message for signing";
byte[] msgBytes = Encoding.UTF8.GetBytes(msg);
byte[] msgHash = new Sha3Keccak().CalculateHash(msgBytes);
var signature = privKey.SignAndCalculateV(msgHash);

Console.WriteLine("Msg: {0}", msg);
Console.WriteLine("Msg hash: {0}", msgHash.ToHex());
Console.WriteLine("Signature: [v = {0}, r = {1}, s = {2}]",
    signature.V[0] - 27,
    signature.R.ToHex(),
    signature.S.ToHex());
```

Complete example:<https://gist.github.com/nakov/f2a579eb9893b29338b11e063d6f80c2>

ECDSA in C#: Verify Message

```
var pubKeyRecovered =
    EthEckey.RecoverFromSignature(signature, msgHash);
Console.WriteLine("Recovered pubKey: {0}",
    pubKeyRecovered.GetPubKey().ToHex().Substring(2));

bool validSig = pubKeyRecovered.Verify(msgHash, signature);
Console.WriteLine("Signature valid? {0}", validSig);
```

Complete example:<https://gist.github.com/nakov/f2a579eb9893b29338b11e063d6f80c2>

Java Crypto Libraries

...

Cryptography in Java

- JCA, Bouncy Castle and Web3j:Hashes, ECC and ECDSA

JCA, Bouncy Castle and Web3j

- Cryptography in Java is based on the Java Cryptography Architecture (JCA)
 - Typical Java style: lot of boilerplate code
- **Bouncy Castle** is the leading Java cryptography library
 - Docs: <https://www.bouncycastle.org/documentation.html>
- **Web3j** – a simplified library for Ethereum and secp256k1
 - Web3j – <https://github.com/web3j>
 - The cryptographic functionality is in web3j/crypto

ECDSA in Java: Install the Crypto Libraries

- This **Maven** dependency will install the following libraries:
 - **org.web3j.crypto**– Ethereum style secp256k1 EC cryptography
 - **org.bouncycastle**– BouncyCastle crypto provider for Java

```
<dependency>
  <groupId>org.web3j</groupId>
  <artifactId>crypto</artifactId>
  <version>3.3.1</version>
</dependency>
```

ECDSA in Java: Initialize the Application

```
import org.bouncycastle.util.encoders.Hex;
import org.web3j.crypto.*;
import java.math.BigInteger;
```

ECDSA in Java: Generate / Load Keys

```
// Generate random private key
// BigInteger privKey = Keys.createEcKeyPair().getPrivateKey();

BigInteger privKey = new BigInteger(
    "97ddae0f3a25b92268175400149d65d6887b9cefaf28ea2c078e05cdc15a3c0a", 16);
BigInteger pubKey = Sign.publicKeyFromPrivate(privKey);
ECPublicKey keyPair = new ECPublicKey(privKey, pubKey);

System.out.println("Private key: " + privKey.toString(16));
System.out.println("Public key: " + pubKey.toString(16));
System.out.println("Public key (compressed): " +
    compressPubKey(pubKey));
```

ECDSA in Java: Sign Message

```
String msg = "Message for signing";
byte[] msgHash = Hash.sha3(msg.getBytes());
Sign.SignatureData signature =
    Sign.signMessage(msgHash, keyPair, false);

System.out.println("Msg: " + msg);
System.out.println("Msg hash: " + Hex.toHexString(msgHash));
System.out.printf(
    "Signature: [v = %d, r = %s, s = %s]\n",
    signature.getV() - 27,
    Hex.toHexString(signature.getR()),
    Hex.toHexString(signature.getS()));
```

ECDSA in Java: Verify Signature

```
BigInteger pubKeyRecovered =
    Sign.signedMessageToKey(msg.getBytes(), signature);
System.out.println("Recovered public key: " +
    pubKeyRecovered.toString(16));

boolean validSig = pubKey.equals(pubKeyRecovered);
System.out.println("Signature valid? " + validSig);
```

Conclusion

...