Practical Blockchain Cryptography for Developers

Hashes, HMAC, Key Derivation, Scrypt, AES, Elliptic Curve Cryptography (ECC), ECDSA, Digital Signatures, Blockchain Cryptography, Ethereum, Blockchain Addresses, Keys, Wallets, BIP39, BIP44, Ethereum Signatures, Ethereum Wallets

Svetlin Nakov, PhD

SoftUni – https://softuni.org

Practical Blockchain Cryptography for Developers

Hashes, HMAC, Key Derivation, Scrypt, AES, Elliptic Curve Cryptography (ECC), ECDSA, Digital Signatures, Blockchain Cryptography, Ethereum, Blockchain Addresses, Keys, Wallets, BIP39, BIP44, Ethereum Signatures, Ethereum Wallets

Author: D-r Svetlin Nakov, PhD

Cryptography - Overview, What is Cryptography?, Hash Functions, Cryptographic Hash Functions, Crypto Hashes and Collisions, Hash Functions: Applications, Secure Hash Algorithms, Hash Functions - Examples, Proof-of-Work Hash Functions, HMAC and Key Derivation, Message Authentication Code (MAC), HMAC and Key Derivation, HMAC Calculation - Examples, Deriving Key from Password, PBKDF2, Modern Key Derivation Functions, Scrypt, Bcrypt, Argon2, Diffie-Hellman Key Exchange, Diffie–Hellman - Concepts, Key Exchange by Mixing Colors, The DHKE Protocol, Symmetric and Asymmetric Encryption, Symmetric Encryption, Public Key Cryptography, Asymmetric Encryption, Symmetric Key Ciphers, AES Cipher - Concepts, AES Encrypt / Decrypt - Examples, Ethereum Wallet Encryption, Asymmetric Key Ciphers, RSA Cryptosystem - Concepts, RSA Encrypt / Decrypt - Examples, Secure Random Number Generators, Random Numbers - Examples, Entropy and HD Wallets, Exercises: Cryptography, Calculate Hashes, Calculate HMAC, Scrypt Key Derivation, AES Encrypt / Decrypt, Blockchain Cryptography, Elliptic Curve Cryptography, Elliptic Curves - Concepts, Public Key Compression, ECC Parameters and secp256k1, Digital Signatures and ECDSA, Ethereum Cryptography, Ethereum Key to Address – Examples, Ethereum Signatures, Sign Message in Ethereum – Examples, Verify Message in Ethereum – Examples, EdDSA and Ed25519, Sign / Verify Message with Ed25519, Wallets, Keys, Signatures, From Wallet to Ethereum Address, From Wallet to Bitcoin Address, Generate Bitcoin Address - Examples, Merkle Trees, Quantum-Safe Cryptography, Exercises: Blockchain Cryptography, Create Ethereum Signature, Ethereum Signature to Address, Ethereum Signature Verifier, Bitcoin Address Generator in C#, Bitcoin Address Generator in JS, Implement a Merkle Tree, RSA Encrypt / Decrypt, ECIES Encrypt / Decrypt, Popular Crypto Libraries, JavaScript Crypto Libraries, Python Crypto Libraries, C# Crypto Libraries, Java Crypto Libraries, Exercises: Sign / Verify Ethereum Messages, Sign Messages in Ethereum Style, Verify Messages in Ethereum Style, Exercises: Private Key To Blockchain Address, Generate Private Key and Address, Existing Private Key to Address, Exercises: Sign / Verify Transactions, Sign / Verify Transactions in JavaScript, Sign / Verify Transactions in Python, Sign / Verify Transactions in C#, Sign / Verify Transactions in Java

Tags: blockchain, cryptography, ECDSA, secp256k1, elliptic curves, digital signatures

GitHub: https://github.com/nakov/blockchain-for-devs-book

Site: http://blockchain-dev-book.softuni.org

Practical Blockchain Cryptogtaphy for Developers

Publisher: XXXXXXXXXXXXX

ISBN: XXX-XXX-XX-XXXX-X

San Francisco, 2018

Table of Contents

Chapter 1. Blockchain Cryptography: Hashes, Elliptic Curves, Keys, Blockchain Addresses, Encryption, Key Derivation, Crypto Wallets

In this chapter we will introduce the **cryptography concepts** used in the blockchain networks, hashes, elliptic curves, public and private keys, encryption, the cryptography behind wallets and transactions.

Watch the video: https://www.youtube.com/watch?v=QbKJfQgQaU4.

We shall start by major cryptography concepts like hashing and cryptographic **hash functions**, **HMAC** and **key derivation** algorithms, symmetric and asymmetric **encryption algorithms**, **secure random generators** and pseudo-random generator functions (PRNG).

After introducing the first few crypto concepts, we shall write some code to play with them. We shall implement **calculation of hashes**, **HMAC calculations** by text and key, password to key derivation using **Scrypt** and **AES**-based **symmetric key encryption** and **decryption** with HMAC for message integrity.

The core of this lesson will be devoted to practical **elliptic curve cryptography** (ECC) and the crypto-systems used in the most popular blockchain networks like **Bitcoin** and **Ethereum**. We shall explore deeply the **ECDSA** and **EdDSA** digital signatures schemes using the `secp256k1` and `Curve25519` elliptic curves and shall explain how **blockchain addresses** are calculated and how **digital signatures** are created and verified in the Ethereum blockchain.

Next, we shall explain the cryptography behind the **crypto wallets**, the concept of **hierarchical (HD) wallets**, the BIP39 and BIP44 standards, the seed words and hierarchical key derivation.

A few words will be devoted to **quantum-safe cryptography** and which classical crypto algorithms are quantum-safe (like hashes and symmetric key ciphers) and which are **quantum-broken** (like most digital signature schemes and ECDSA).

After the next portion of concepts, we shall **write some code**: play with Ethereum message signing and signatures verification, generating private keys, deriving public keys and addresses for the Ethereum and Bitcoin networks, as well as implementing RSA-based asymmetric encryption.

Finally, we shall explore popular **crypto libraries** for implementing blockchain crypto algorithms for **JavaScript**, **Python**, **C#** and **Java**, along with practical exercises to sign Ethereum JSON messages, generate ECC private keys and derive blockchain addresses and signing / verifying transactions in JS, Python, C# and Java.

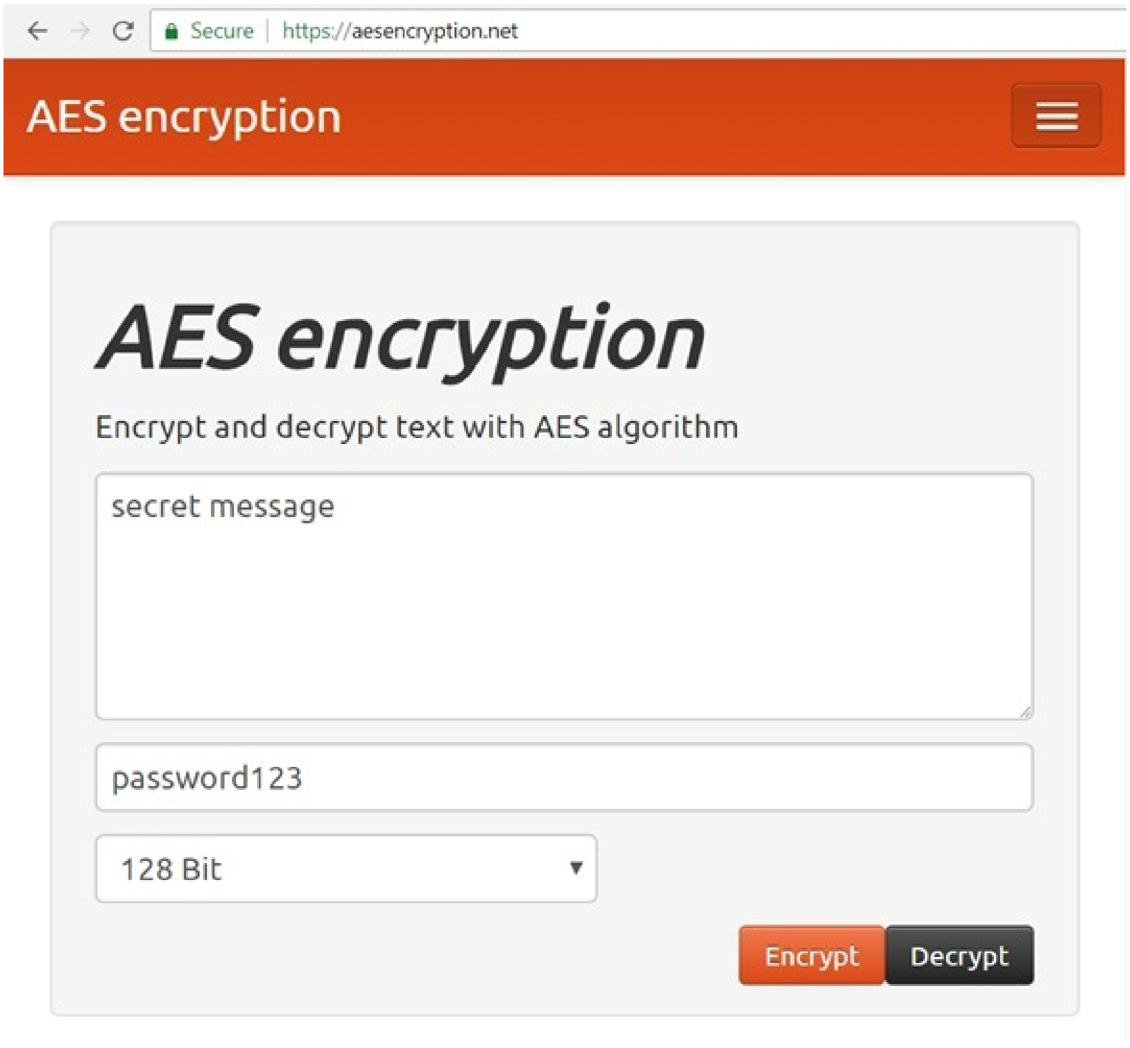
Cryptography - Overview

In this topic we shall introduce basic **cryptography concepts** like cryptographic **hash functions** (SHA-256, SHA3, RIPEMD and others), **HMAC** (hashed message authentication code), password to **key derivation**, the Diffie-Hellman key-exchange protocol, **symmetric key** encryption schemes (like the **AES** cipher) and **asymmetric key** encryption schemes (like the **RSA** cipher), as well as the concept of **entropy** and secure **random number** generation.

We shall explain how the applied cryptography is related to **programming** and **blockchain development**.

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.


Secure | https://aesencryption.net

AES encryption

AES encryption

Encrypt and decrypt text with AES algorithm

secret message

password123

128 Bit

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

What is Cryptography and How It is Related to Blockchain?

Cryptography is the science of providing **security** and **protection** of information. It is widely used in blockchain systems to sign transactions, securely transfer blockchain assets, encrypt wallets and in many other scenarios.

Watch the video: https://www.youtube.com/watch?v=BLXRNfif6o4.

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 (like AES and RSA), where one or more **keys** are used to transform data from plain to encrypted form and back. **Symmetric encryption** uses the same key to encrypt and decrypt messages, while **asymmetric encryption** uses a key pair (encryption key and corresponding decryption key). In blockchain encryption is used in wallets to protect the private keys and user's assets on the chain from unauthorized access.

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). Wallets in the blockchain systems hold the user's keys, usually protected by a password or PIN code and sign transactions.

Cryptography defines **key-exchange algorithms** (like Diffie-Hellman key exchange), used to securely exchange data encryption **keys** between two parties that intend to transmit messages securely using **encryption**.

Cryptography uses **random numbers** and deals with **entropy** and secure generation of random numbers. Crypto wallets generate random keys to create a new blockchain account.

Cryptography provides **data hashing** functions (like SHA-256, SHA3-256 and RIPEMD-160), which transform messages to **message digest** (hash of fixed length), which cannot be reversed back to the original message and almost uniquely identifies it. In blockchain hashes are used for generating blockchain addresses, transaction identification and in many other algorithms and protocols.

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 blockchain systems **digital signatures** are used to sign transactions and allow users to transfer a blockchain asset from one address to another.

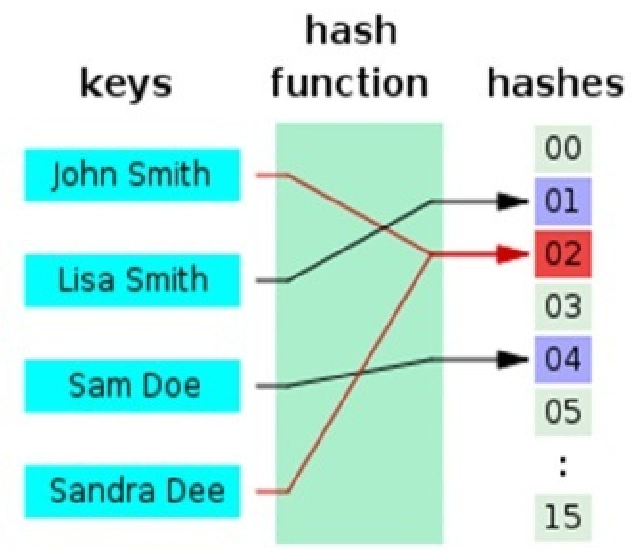
Cryptography in the blockchain networks deals also with **crypto wallets**, hierarchical key derivation, blockchain addresses, merkle trees, proof-of-work mining, and many other blockchain-specific crypto algorithms, standards and protocols. Blockchain developers need to understand cryptography very well in order to deal with wallets, keys, addresses, hashes, transactions, signatures, mining, etc. and to ensure strong level of security in the apps they build.

Hashing and 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).

Watch the video: https://www.youtube.com/watch?v=FIJVI16e2tI.

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**".

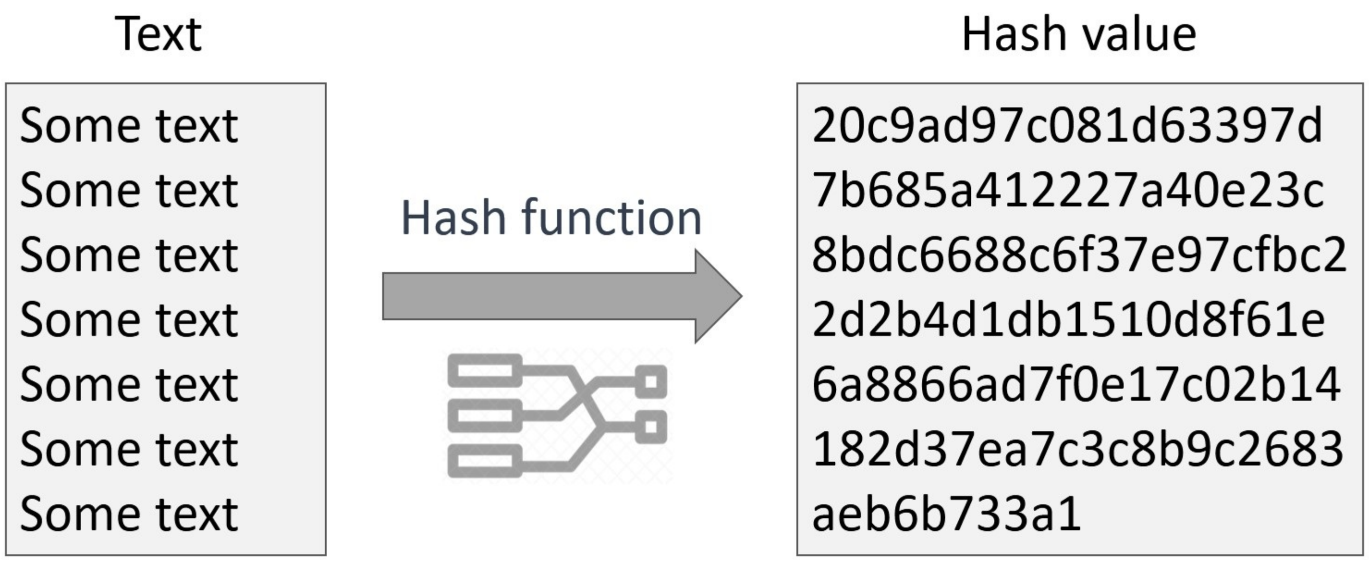
Hash functions are **irreversible by design**, which means that there is no fast algorithm to restore the input message from its hash value.

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 `ehllo` will have the same hash code. **Better hash functions** 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"
```

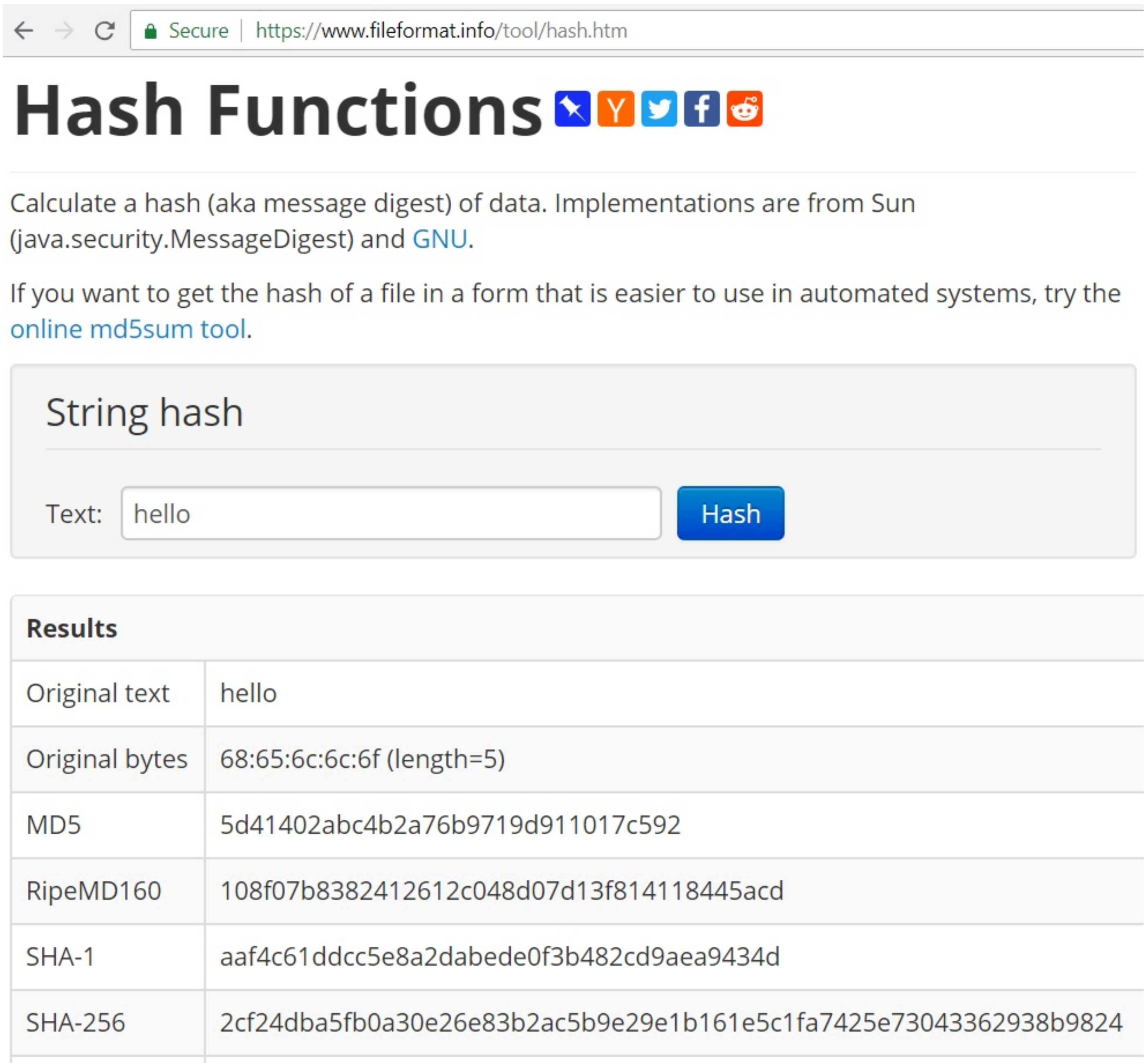
There is no efficient algorithm to find the input message (in the above example `hello`) from its hash value (in the above example `2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824`). It is well-known that cryptographic hash functions **cannot be reversed** back, so they are used widely 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") = "75d527c368f2efe848ecf6b073a36767800805e9eef2b1857d5f
984f036eb6df891d75f72d9b154518c1cd58835286d1da9a38deba3de98b5a53e5ed78a84
976"
```

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: hello

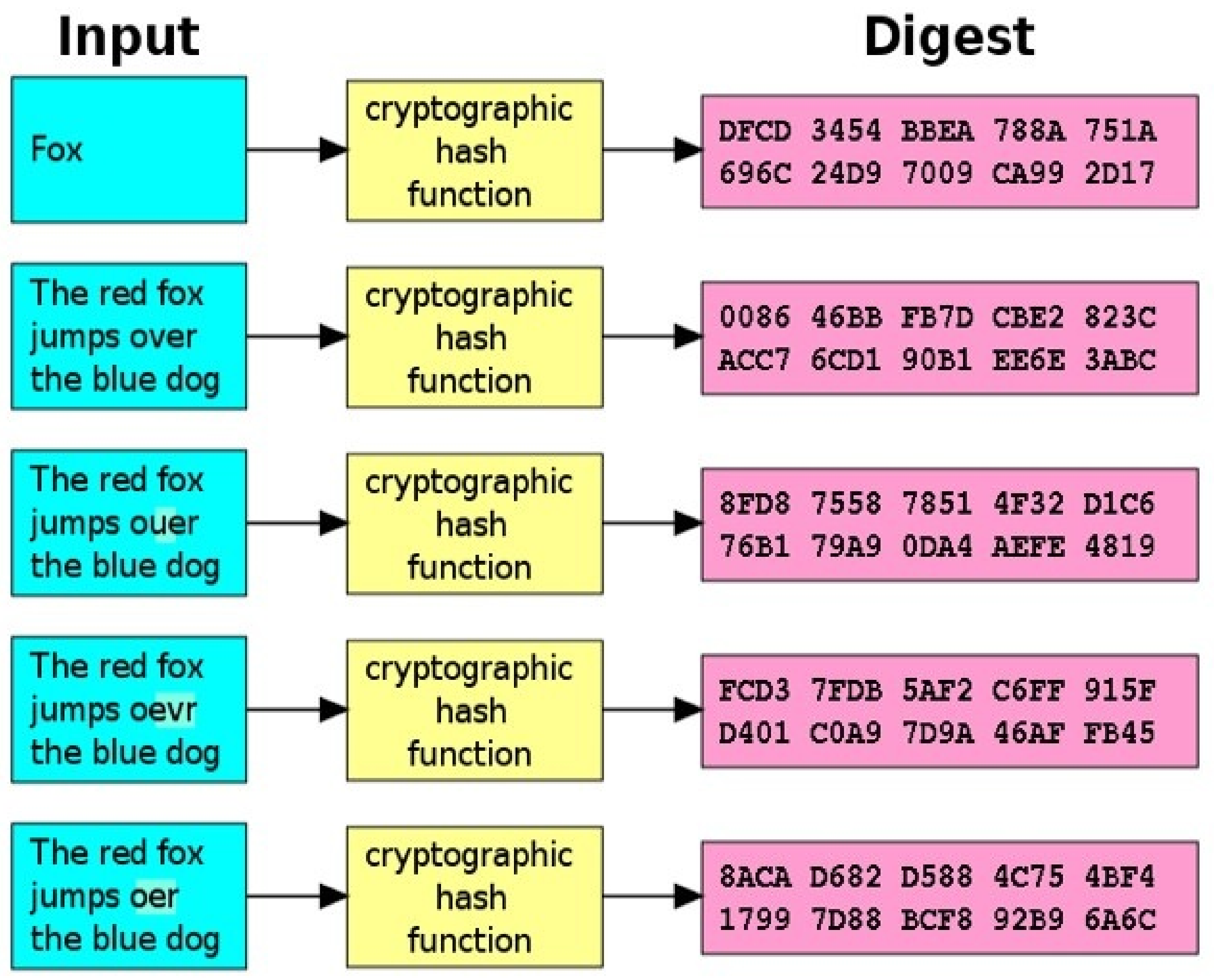
Hash

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).


Input
Digest
Fox
cryptographic hash function
DFCD 3454 BBEA 788A 751A 696C 24D9 7009 CA99 2D17
The red fox jumps over the blue dog
cryptographic hash function
0086 46BB FB7D CBE2 823C ACC7 6CD1 90B1 EE6E 3ABC
The red fox jumps ouer the blue dog
cryptographic hash function
8FD8 7558 7851 4F32 D1C6 76B1 79A9 0DA4 AEFE 4819
The red fox jumps oevr the blue dog
cryptographic hash function
FCD3 7FDB 5AF2 C6FF 915F D401 C0A9 7D9A 46AF FB45
The red fox jumps oer the blue dog
cryptographic hash function
8ACA D682 D588 4C75 4BF4 1799 7D88 BCF8 92B9 6A6C

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` = `hash(p)` for certain strong cryptographic hash function `hash`.
- It is expected to be **extremely hard** to find an input `p'`, such that `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.

Cryptographic Hash Functions: Applications

Cryptographic hash functions (like SHA-256 and SHA3-256) are used in many scenarios:

- 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).
- 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).
- Generate an **almost unique ID** of certain document / message. The document ID (hash value) can be used later to prove the existence of the message, or to retrieve the message from a storage system.
- **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.
- **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 `hash(x + p)` holds 10 zero bits at its beginning.

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 was broken, some are still considered secure. Let's review the most widely used cryptographic hash functions (algorithms).

Old hash algorithms like **MD5**, **SHA-0** and **SHA-1** were withdrawn due to **cryptographic weaknesses** (collisions found).

- **Don't use MD5**, **SHA0** and **SHA1**! 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.

Modern cryptographic hash algorithms are considered secure enough for most applications:

- **SHA-2** is a family of strong cryptographic hash functions: **SHA-256** (256 bits hash), **SHA-512** (512 bits hash), etc.
  - **SHA-2** is widely used 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.
- By design, **more bits** at the hash output are expected to achieve **stronger security**. 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** is considered **more secure than SHA-2** (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 also known as "**Keccak**" hashes. The hash function **Keccak-256** is used in **Ethereum**.
- **RIPEMD-160** is secure hash function, widely used in cryptography, e.g. in PGP and Bitcoin.
  - The 160-bit variant is considered cryptographically stronger than the other variations like RIPEMD-128, RIPEMD-256 and RIPEMD-320.
  - SHA-256 and SHA3-256 are more stronger than RIPEMD-160, due to higher bit length and less chance for collisions.
- **BLAKE** / **BLAKE2** / **BLAKE2s** / **BLAKE2b** is a family of fast, secure cryptographic hash functions.
  - The **BLAKE2** function is improved version of **BLAKE**.
  - **BLAKE** has **256-bit** variant (BLAKE-256, BLAKE2s) and **512-bit** variant (BLAKE-512, BLAKE2b).

  - The BLAKE2 hash function has similar security strength like SHA-256.

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

- **Brute forcing** to find collision 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 Mar 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.

Watch the video: https://www.youtube.com/watch?v=7U-hAvU01WI.

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:

```python
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))

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

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

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

```
SHA-256:    b'2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938
b9824'
SHA3-256:   b'3338be694f50c5f338814986cdf0686453a888b84f424d792af4b920239
8f392'
RIPEMD-160: b'108f07b8382412612c048d07d13f814118445acd'
BLAKE2s:    b'19213bacc58dee6dbde3ceb9a47cbb330b3d86f8cca8997eb00be456f14
0ca25'
```

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

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'1c8aff950685c2ed4bc3174f3472287b56d9517b9c948127319a09a7a36
deac8'
```

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**".

Watch the video: https://www.youtube.com/watch?v=m5wLcVsIXvs.

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

HMAC and Key Derivation

In this section, we shall explain **MAC** (message authentication codes), **HMAC** (hash-based message authentication code) and **KDF** (key derivation functions).

Watch the video: https://www.youtube.com/watch?v=Qev0lghHGpE.

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 **A**uthentication **C**ode is cryptographic code, calculated by given **key** and given **message**:

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

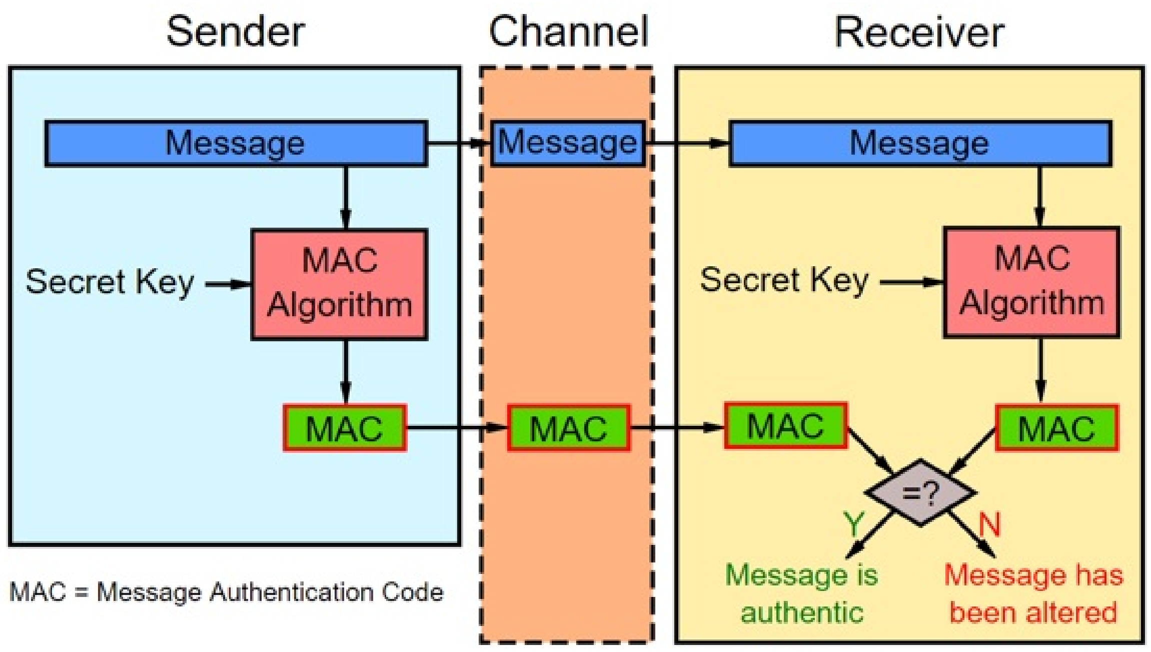
Typically, it behaves **like a hash function**: a minor change in the message or in the key results to totally different **MAC value.** Also, it should be practically infeasible to change the key or the message and get the same **MAC value**.

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.

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.



Encrypt / Decrypt Messages using MAC

Another scenario to use **MAC codes** is when we **encrypt a message** and we want to be sure the **decryption password is correct**.

- 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**

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

Simply calculating `hash_func(key + msg)` is considered **insecure** (see the details).

It is recommended to use a **secure HMAC algorithm instead**, e.g. `HMAC-SHA256` or `HMAC-SHA3-512`.

What is HMAC?

HMAC = **H**ash-based **M**essage **A**uthentication **C**ode (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
```

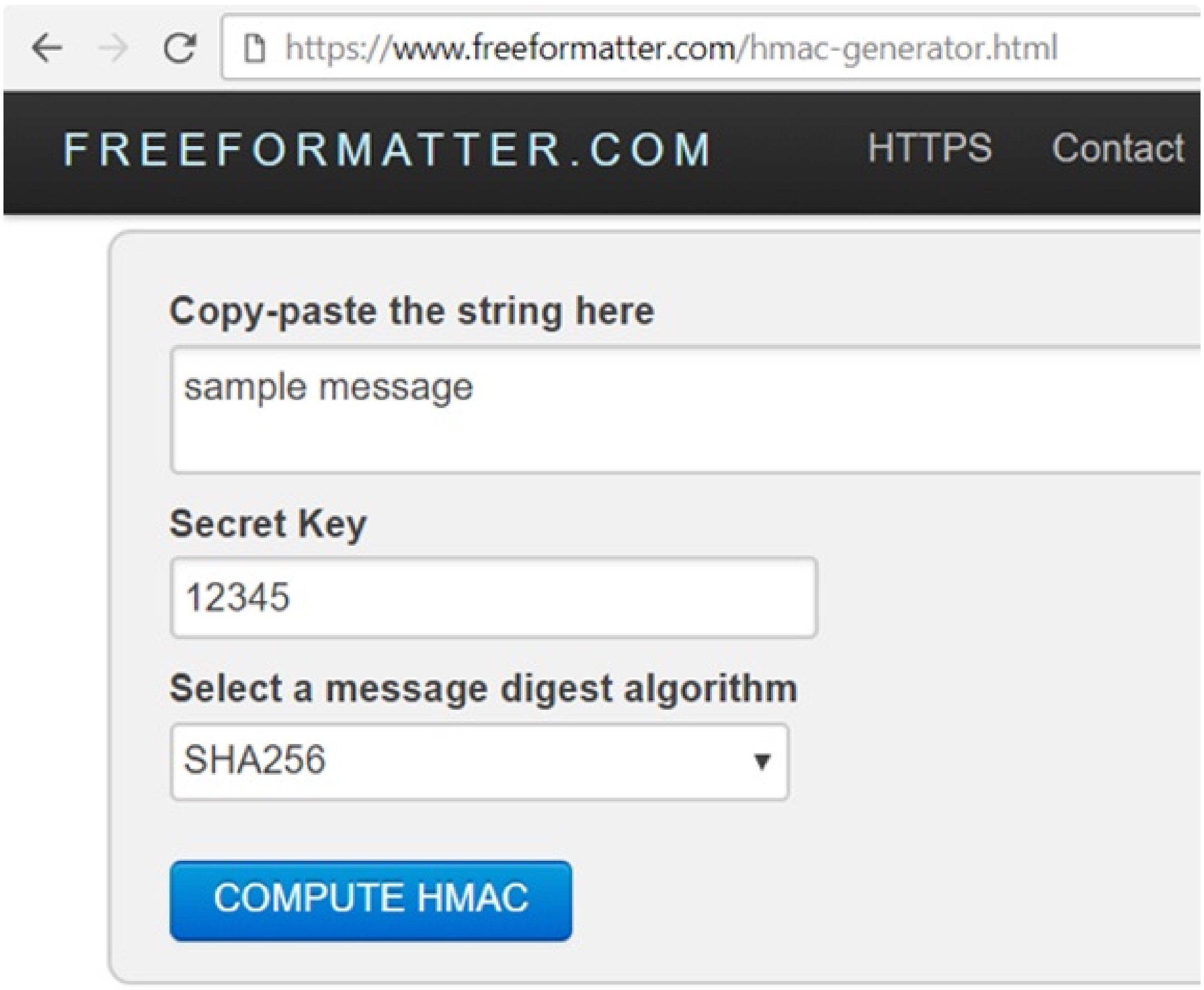
As **very simple KDF function**, we can use SHA256: just hash the password. Don't do this, because it is **insecure**. Simple hashes are vulnerable to **dictionary attacks**.

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

Copy-paste the string here

sample message

Secret Key

12345

Select a message digest algorithm

SHA256

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

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

Try the above 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.

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

In this section we shall 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.

Watch the video: https://www.youtube.com/watch?v=hUuld46yqOU.

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 algorithms 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.

Modern Key Derivation Functions

PBKDF2, Bcrypt, Scrypt and Argon2 are significantly stronger key derivation functions and are designed to survive password guessing 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).

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).

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. It is described in the Internet standard RFC 2898 (PKCS #5).

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

- `password` – array of bytes / string, e.g. "*p@$Sw0rD~3*"
- `salt` – securely-generated random bytes, e.g. "*df1f2d3f4d77ac66e9c5a6c3d8f921b6*"
- `iterations-count` , e.g. 1024 iterations
- hash-function for calculating **HMAC**, e.g. `SHA256`
- `derived-key-len` for the output, e.g. 256 bits

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

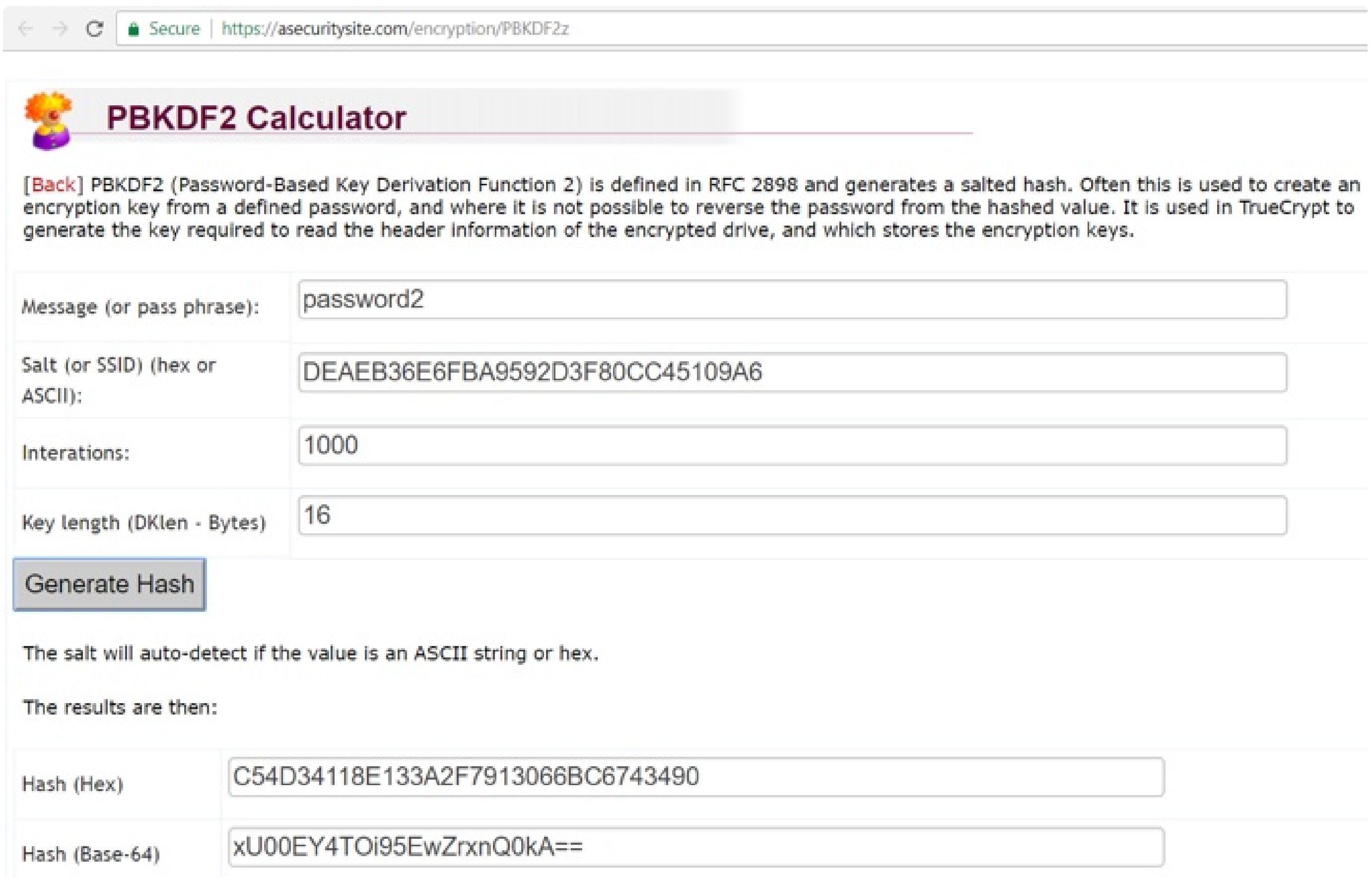
PBKDF2 and Number of Iterations

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.


Secure | https://asecuritysite.com/encryption/PBKDF2z

PBKDF2 Calculator

[Back] PBKDF2 (Password-Based Key Derivation Function 2) is defined in RFC 2898 and generates a salted hash. Often this is used to create an encryption key from a defined password, and where it is not possible to reverse the password from the hashed value. It is used in TrueCrypt to generate the key required to read the header information of the encrypted drive, and which stores the encryption keys.

Message (or pass phrase):	password2
Salt (or SSID) (hex or ASCII):	DEAEB36E6FBA9592D3F80CC45109A6
Interations:	1000
Key length (DKlen - Bytes)	16

The salt will auto-detect if the value is an ASCII string or hex.

The results are then:

Hash (Hex)	C54D34118E133A2F7913066BC6743490
Hash (Base-64)	xU00EY4TOi95EwZrxnQ0kA==

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('df1f2d3f4d77ac66e9c5a6c3d8f921b6')
passwd = "p@$Sw0rD~3".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'78856da569d1aa59c42c19f6b6c96ba773a4bb1f4d7a2787cd75095d6d
7fd145'
```

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

Modern KDFs: Bcrypt, Scrypt and Argon2

Modern key-derivation functions (KDF) like **Scrypt** and **Argon2** derive a key (of fixed length) from a password (text) and are **resistant** to **dictionary attacks** and **ASIC attacks**.

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

It takes **CPU time** to derive the key (e.g. 0.2 sec) + **memory** (RAM). The calculation process is memory-dependent, so **the memory access is the bottleneck**.

Thus, **cracking passwords will be slow** and inefficient (e.g. 5-10 attempts / second), even on very good password cracking hardware. The goal is to make practically infeasible to perform a brute-force attack to guess the password from its hash.

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

Scrypt: Derive Key from Password

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

The Scrypt algorithms takes several **input parameters**:

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

Scrypt Parameters

The **Scrypt 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`, `salt` and `derived-key-length` - work like in the others KDF algorithms

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 is calculated as follows:

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

Example: e.g. 128 * N * r * p = 128 * 16384 * 8 * 1 = 16 MB

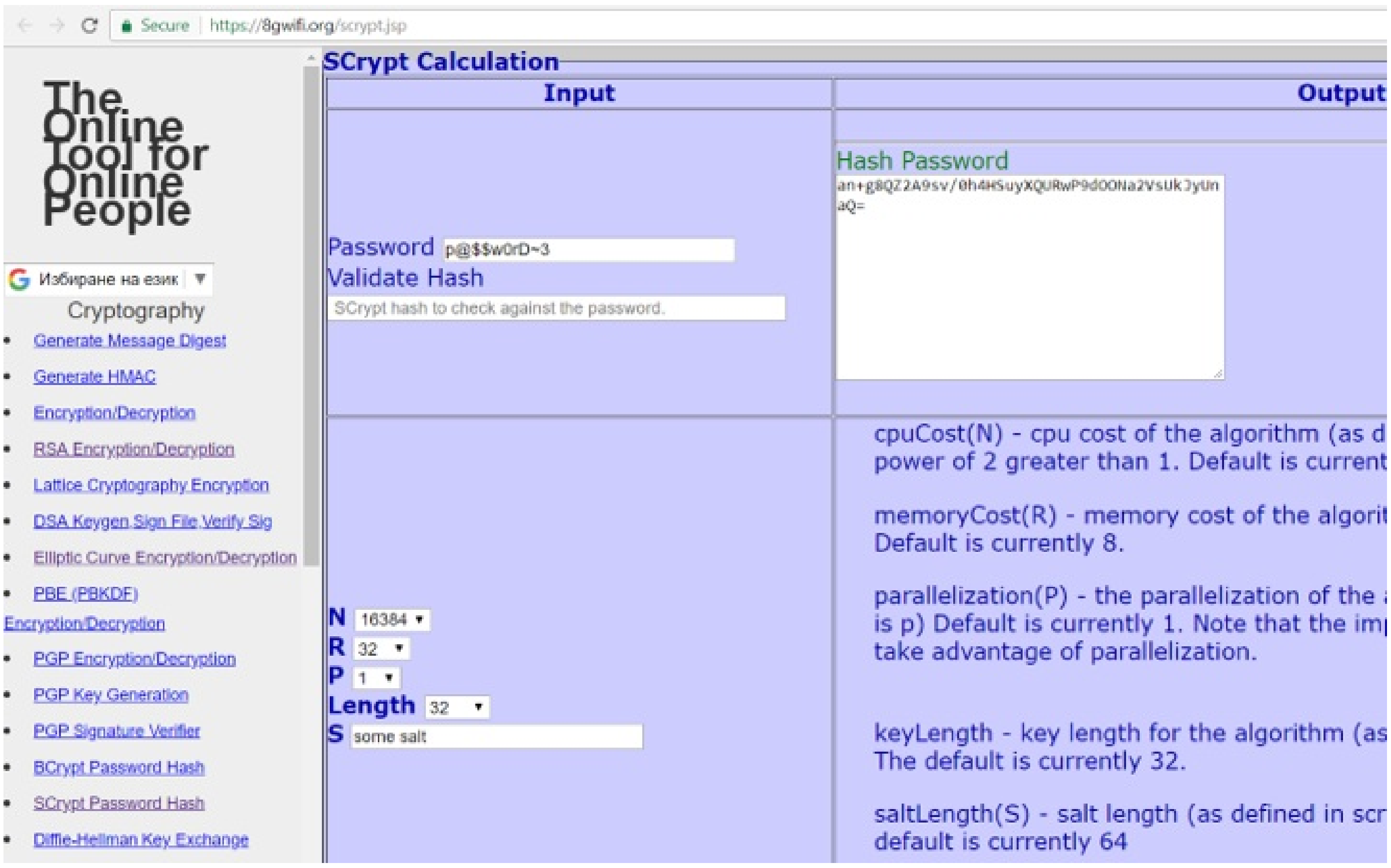
Choosing parameters depends on how much you want to wait and what level of security (password cracking resistance) do you need:

- Sample parameters for **interactive login**: N=16384, r=8, p=1 (RAM = 16MB)
- Sample parameters for **file encryption**: N=1048576, r=8, p=1 (RAM = 1GB)

You can perform tests and choose the Scrypt parameters yourself. In **MyEtherWallet**, the default parameters are N=8192, r=8, p=1 (which is not strong enough for crypto wallet, but this is how it works).

Scrypt - Example

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



https://8gwifi.org/scrypt.jsp

The Online Tool for Online People

Cryptography

- Generate Message Digest
- Generate HMAC
- Encryption/Decryption
- RSA Encryption/Decryption
- Lattice Cryptography Encryption
- DSA Keygen Sign File Verify Sig
- Elliptic Curve Encryption/Decryption
- PBE (PBKDF) Encryption/Decryption
- PGP Encryption/Decryption
- PGP Key Generation
- PGP Signature Verifier
- BCrypt Password Hash
- SCrypt Password Hash
- Diffie-Hellman Key Exchange

SCrypt Calculation

Input	Output
Password p@$$w0rD~3 Validate Hash SCrypt hash to check against the password.	Hash Password an+g8QZ2A9sv/0h4HSuyXQURwP9dOONa2VsUkJyUn aQ=
N 16384 R 32 P 1 Length 32 S some salt	cpuCost(N) - cpu cost of the algorithm (as d power of 2 greater than 1. Default is current memoryCost(R) - memory cost of the algorit Default is currently 8. parallelization(P) - the parallelization of the is p) Default is currently 1. Note that the im take advantage of parallelization. keyLength - key length for the algorithm (as The default is currently 32. saltLength(S) - salt length (as defined in scr default is currently 64

Scrypt Calculation in Python - Example

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

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

```
pip install scrypt
```

Now, write the Python code to calculate Scrypt:

```python
import scrypt, binascii

salt = binascii.unhexlify('df1f2d3f4d77ac66e9c5a6c3d8f921b6')
passwd = "p@$Sw0rD~3".encode("utf8")
key = scrypt.hash(passwd, salt, 16384, 8, 1, 32)
print("Derived key:", binascii.hexlify(key))
```

The **Scrypt** 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'f393c3c50ffdbd4a2f60d51b7a5bef3138c5b11f27e82304c465ea1614
42bb4c'
```

Try to change the number of **iterations** or the **block size** and see how they affect the **execution time**.

BCrypt: Derive Key from Password

Bcrypt is another cryptographic KDF function, **older than Scrypt**, 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.

Modern applications should **prefer Scrypt** instead of Bcrypt.

Argon2: Derive Key from Password

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

It is considered that when configured correctly, **Argon2 is more secure** than Scrypt, Bcrypt and PBKDF2.

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)

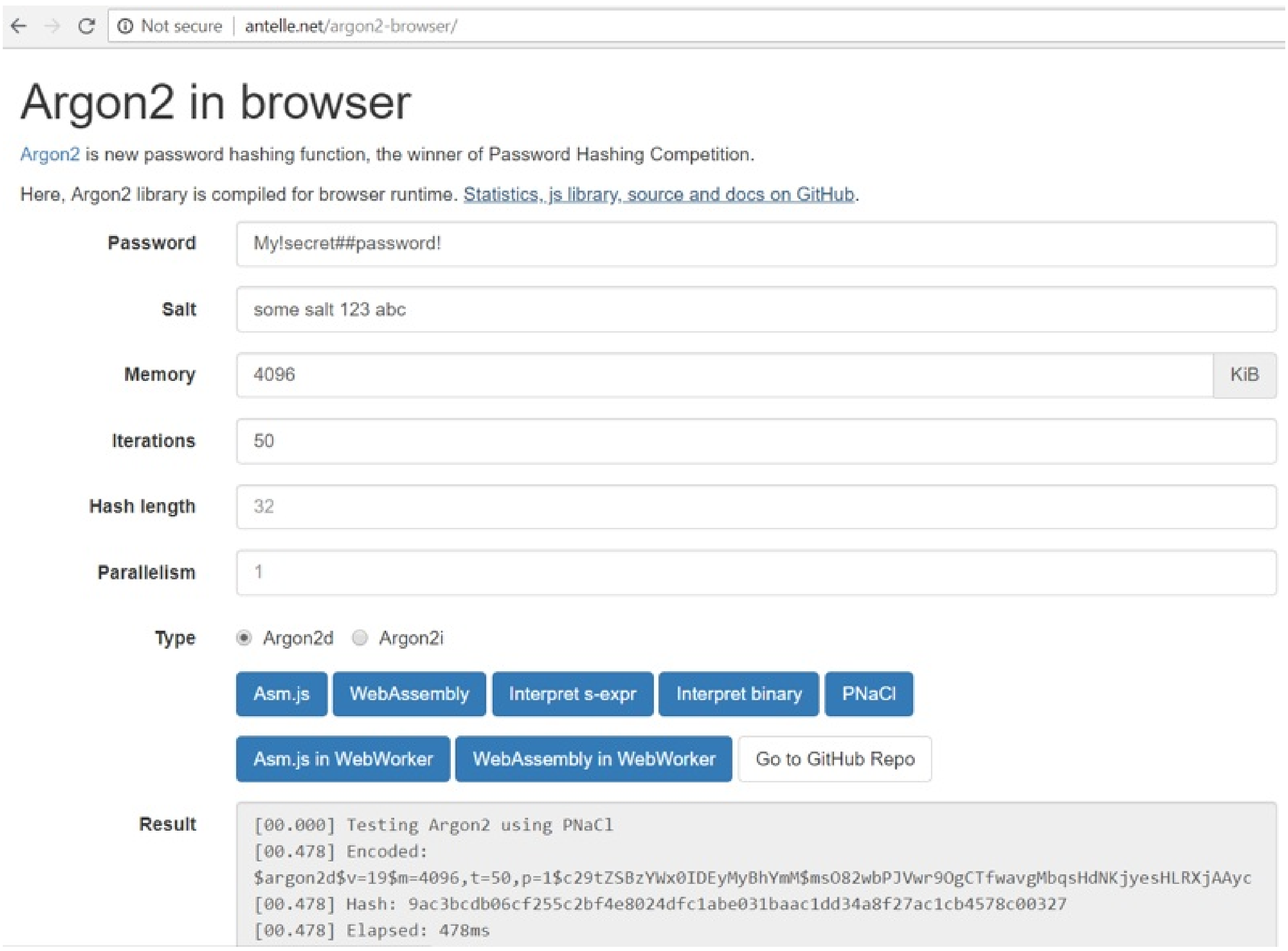
Config Parameters of Argon2

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

- **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.



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	My!secret##password!
Salt	some salt 123 abc
Memory	4096 KiB
Iterations	50
Hash length	32
Parallelism	1
Type	[✓ Argon2d] Argon2i

Result

```
[00.000] Testing Argon2 using PNaCl
[00.478] Encoded:
$argon2d$v=19$m=4096,t=50,p=1$c29tZSBzYWx0IDEyMyBhYmM$msO82wbPJVwr9OgCTfwavgMbqsHdNKjyesHLRXjAAyc
[00.478] Hash: 9ac3bcdb06cf255c2bf4e8024dfc1abe031baac1dd34a8f27ac1cb4578c00327
[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:

```python
import argon2

argon2Hasher = argon2.PasswordHasher(time_cost=50, memory_cost=102400, pa
rallelism=8, hash_len=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).

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

```
Derived key: $argon2id$v=19$m=102400,t=50,p=8$JPoIjwAPeCGiLFwdhcCMwQ$Mf9d
8TtMA7b21/8VTyW+zEYlzMo2TyPclkf4qnNUzCI
```

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.

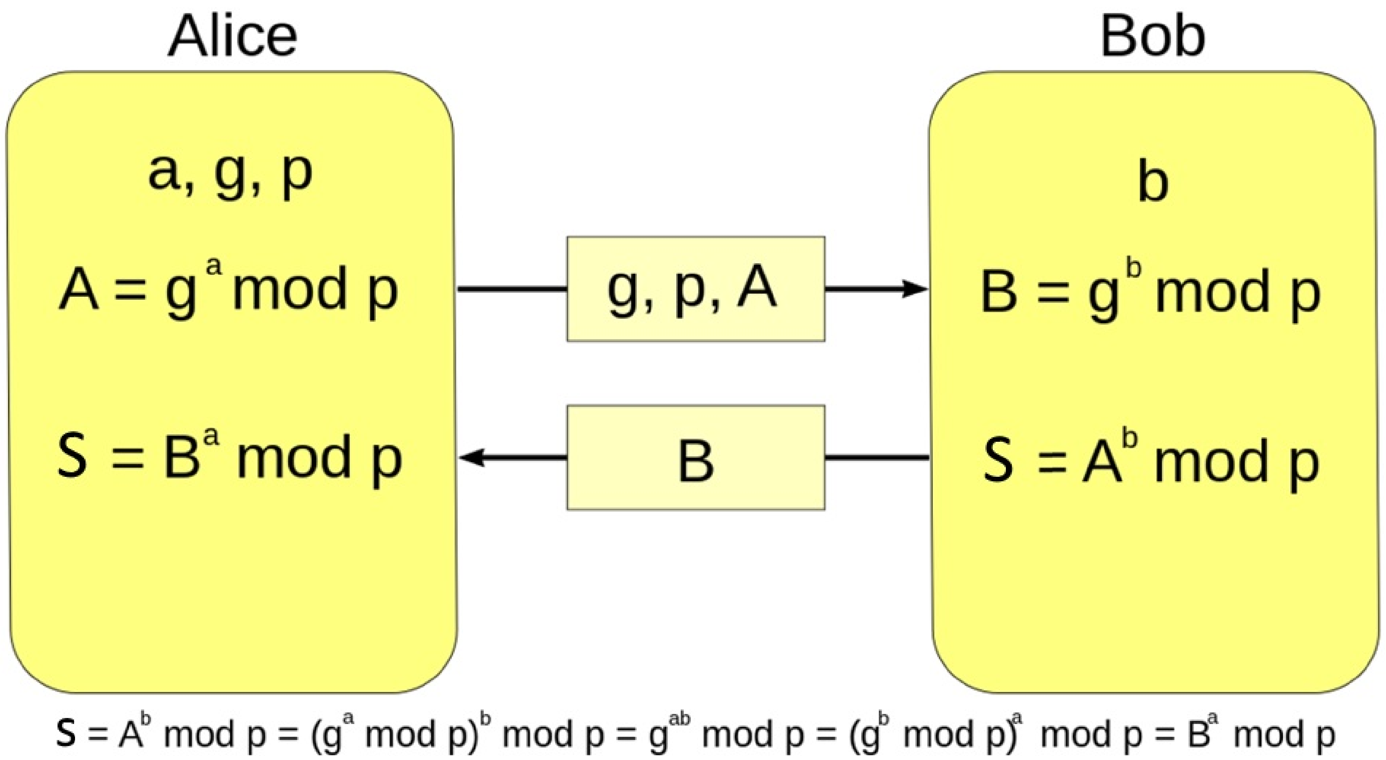
Try to change the **time_cost** or the **memory_cost** settings and see how they affect the **execution time** for the key derivation.

Diffie-Hellman Key Exchange

Diffie–Hellman Key Exchange (DHKE) is a method to securely **exchange cryptographic keys** in a way that overheard communication does not reveal the keys. The exchanged keys are further used for encrypted communication.

Watch the video: https://www.youtube.com/watch?v=NahYHUSyknE.

In the next few sections we shall explain **how DHKE works** and the math behind it. In short, the process is outlined on the below scheme:


Alice
Bob
a, g, p
$A = g^a \bmod p$
$S = B^a \bmod p$
g, p, A
B
b
$B = g^b \bmod p$
$S = A^b \bmod p$
$S = A^b \bmod p = (g^a \bmod p)^b \bmod p = g^{ab} \bmod p = (g^b \bmod p)^a \bmod p = B^a \bmod p$

Let's see the details under these calculations.

Diffie–Hellman Key Exchange (DHKE) - Concepts

Diffie–Hellman Key Exchange (DHKE) is a cryptographic method (protocol) to **securely exchanging cryptographic keys** over a public (insecure) channel. Once the keys are securely exchanges, an encrypted communication can start, 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 is someone sniffs the communication between the parties, the information exchanged can be revealed.

The Diffie–Hellman (DH) method 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).

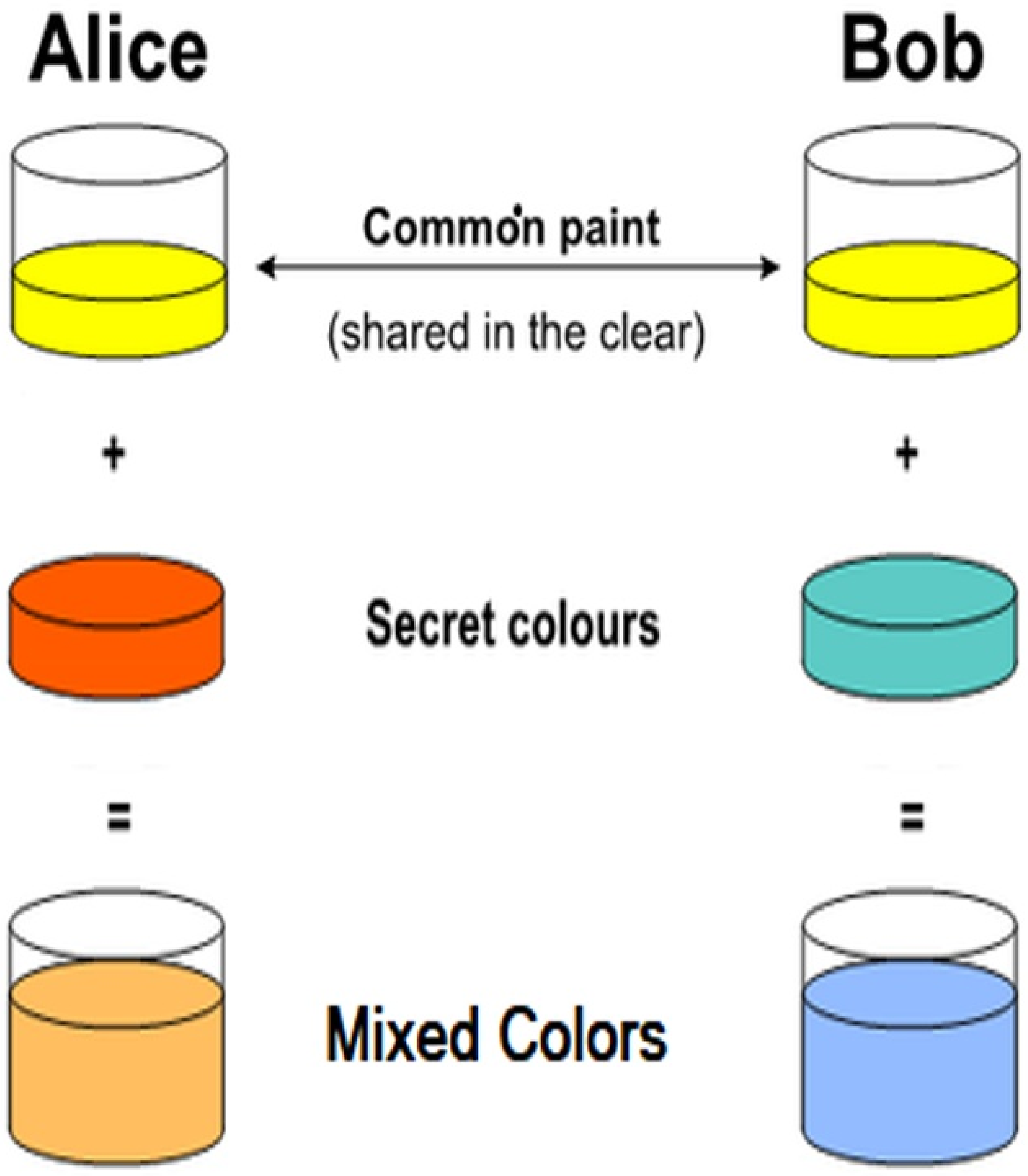
Key Exchange by Mixing Colors

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.

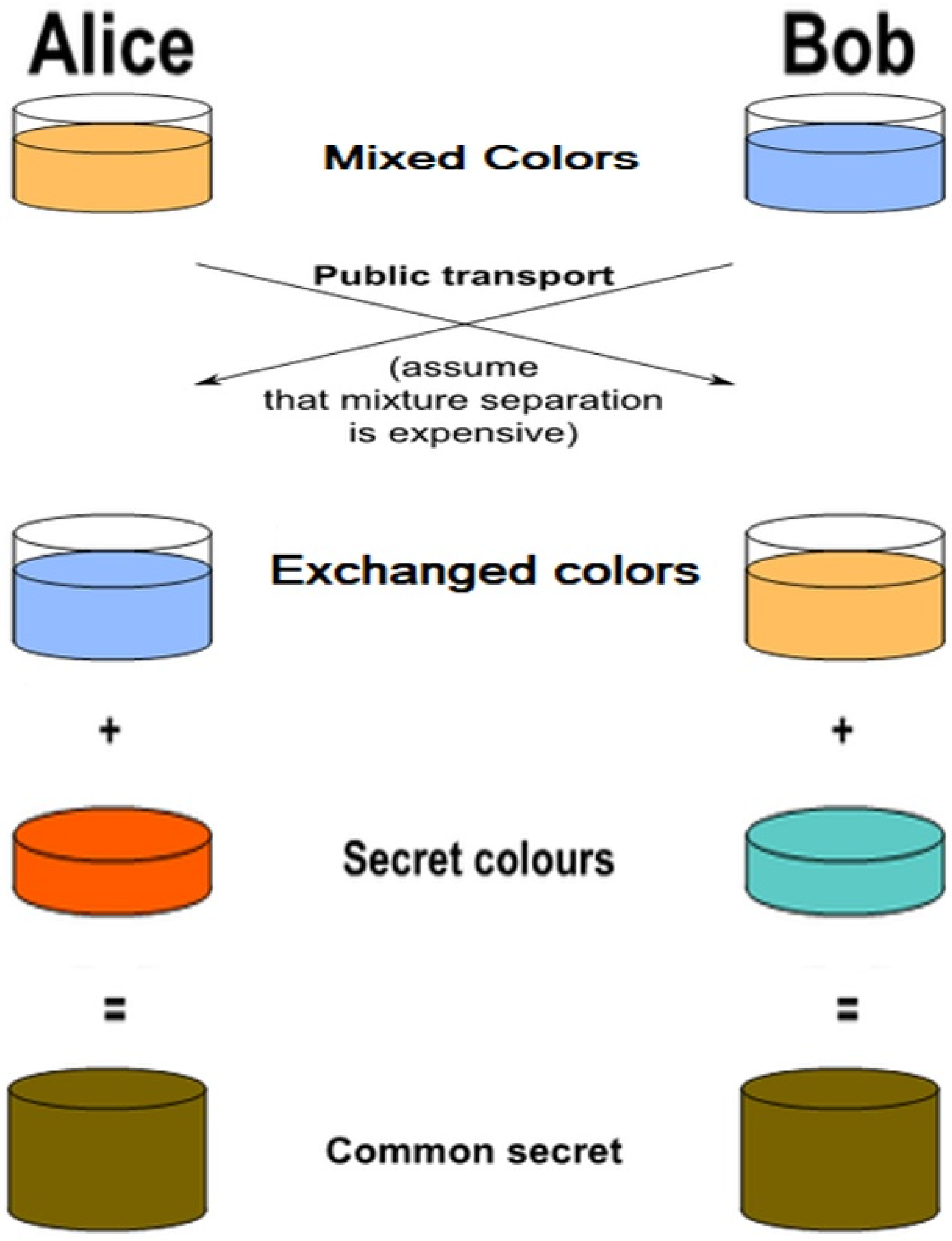
This is the color exchange **scenario**, step by step:

- **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 area 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

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

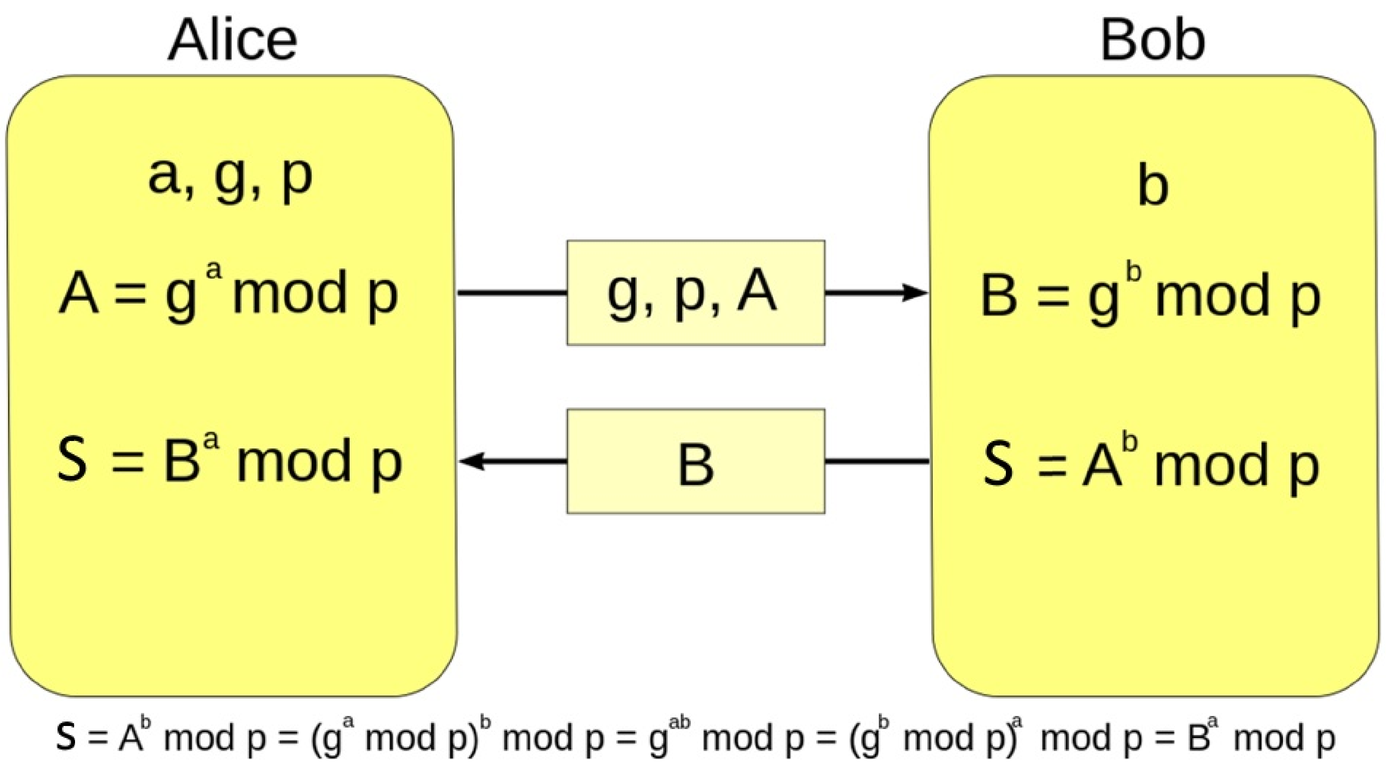
In computing theory, these is no efficient algorithm which can find a secret exponent. If we have **m**, **g** and **p** from the below equation:

$m = g^s \bmod p$

there is not efficient (fast) algorithm to find the secret exponent **s**.

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:


Alice
Bob
a, g, p
A = g^a mod p
S = B^a mod p
g, p, A
B
b
B = g^b mod p
S = A^b mod p
S = A^b mod p = (g^a mod p)^b mod p = g^ab mod p = (g^b mod p)^a mod p = B^a mod p

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

- Alice and Bob agree to use two public integers: **modulus p** and **base g**.
  - 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 $\mathbf{A = g^a \bmod p}$.
  - 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 mod 23 = 4.
- Bob chooses a **secret integer b** (e.g. **b** = 3), then calculates and sends to Alice the number $\mathbf{B = g^b \bmod p}$.
  - In our case we have: **B** = 5^3 mod 23 = 10
- Alice computes s = B^a mod p
  - In our example: **s** = 10^4 mod 23 = **18**
- Bob computes s = A^b mod p
  - In our example: **s** = 4^3 mod 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.

Security of the DHKE Protocol

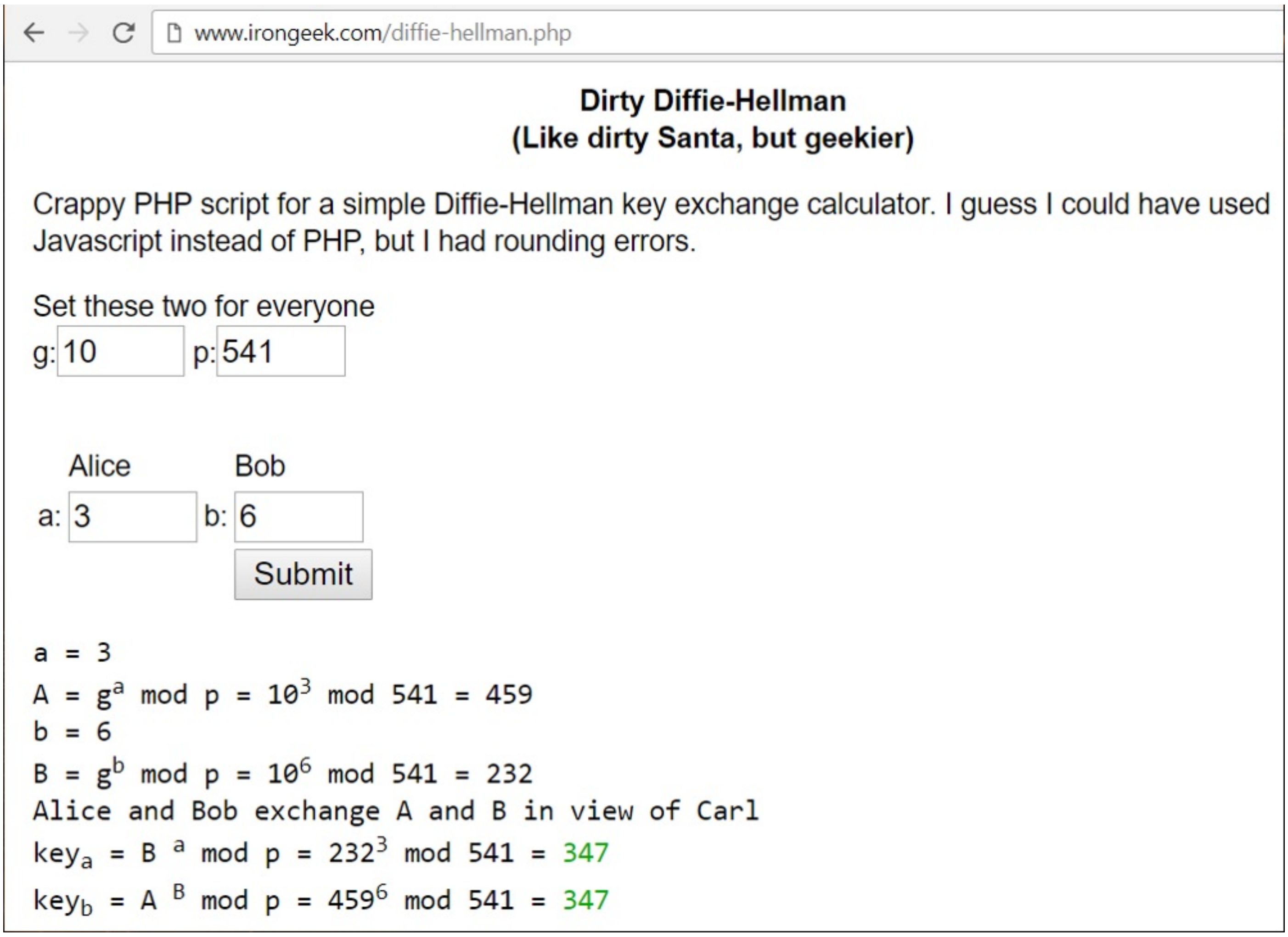
This is how DHKE works. It exchanges a **non-secret sequence of integer numbers** over insecure, public (sniffable) channel (such as signal going through a cable or propagated by waves in the air), but does not reveal the secretly-exchanged shared private key.

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

Finally, not that the integers **g**, **p**, **a** and **p** 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



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: 10 p: 541

Alice	Bob
a: 3	b: 6

```
a = 3
A = g^a mod p = 10^3 mod 541 = 459
b = 6
B = g^b mod p = 10^6 mod 541 = 232
Alice and Bob exchange A and B in view of Carl
key_a = B ^a mod p = 232^3 mod 541 = 347
key_b = A ^B mod p = 459^6 mod 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)** later in this chapter.

Symmetric and Asymmetric Encryption - Concepts

...

Watch the video: https://www.youtube.com/watch?v=EMXGdGt6WYg.

Symmetric Encryption - Concepts and Algorithms

...

Public Key Cryptography - Concepts

...

Asymmetric Encryption - Concepts and Algorithms

...

Symmetric Key Ciphers - Overview

...

The AES Symmetric-Key Cipher - Concepts

...

AES Encryption / Decryption - Examples in Python

...

Ethereum UTC / JSON Wallet Encryption

AES in action: wallet encryption in Ethereum

...

Asymmetric Key Ciphers - Overview

...

The RSA Cryptosystem - Concepts

...

RSA Encryption / Decryption - Examples in Python

...

Secure Random Number Generators

...

Random Numbers - Examples

...

Entropy and Seed in HD Wallets (BIP-39)

...

Exercises: Blockchain Cryptography - Hashes, HMAC, SCrypt, Twofish

In this exercise, you shall write code to play with **popular cryptographic algorithms** using crypto libraries from your programming languages. You shall calculate **hashes**, calculate **HMAC** codes, **derive keys** by given password, **encrypt and decrypt** messages.

Exercise: Calculate Hashes

Exercise: Calculate HMAC

...

Exercise: Derive Key by Password using Scrypt

...

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

...

Blockchain Cryptography: Elliptic Curves, ECDSA, Keys, Addresses, Signatures, Merkle Trees

...

Elliptic Curve Cryptography (ECC)

...

Elliptic Curves - Concepts

...

Public Key Compression in Elliptic Key Cryptosystems

...

ECC Parameters and the "secp256k1" Curve

...

Digital Signatures and ECDSA

...

Ethereum Cryptography, Keys, Addresses and secp256k1

...

Ethereum Key to Address – Examples

...

Digital Signatures in Ethereum Blockchain

...

Sign Message in Ethereum – Examples

...

Verify Message in Ethereum – Examples

...

EdDSA Cryptosystem and the Curve "Ed25519"

...

Sign / Verify Message with Ed25519

...

Crypto Wallets, Public / Private Keys and Signatures in the Blockchain Networks

...

From Wallet to Ethereum Address

...

From Wallet to Bitcoin Address

...

Generate Bitcoin Address - Examples

...

Merkle Trees and Blockchain

...

Quantum-Safe Cryptography

Quantum-Resistant Crypto Algorithms

...

Exercises: Blockchain Cryptography

...

Exercise: Create Ethereum Signature

...

Exercise: Recover Ethereum Address from Signature

...

Exercise: Ethereum Signature Verifier

...

Exercise: Bitcoin Address Generator in C

...

Exercise: Bitcoin Address Generator in JavaScript

...

Exercise: Implement a Merkle Tree in Python

...

Exercise: RSA Encrypt / Decrypt

...

Exercise: ECC-Based Asymmetric Encrypt / Decrypt (ECIES)

...

Popular Cryptographic Libraries for JavaScript, Python, Java and C

...

JavaScript Crypto Libraries

...

Python Crypto Libraries

...

C# Crypto Libraries

...

Java Crypto Libraries

...

Exercises: Sign and Verify Ethereum Messages

...

Exercise: Sign Messages in Ethereum JSON Format

...

Exercise: Verify Signed Messages in Ethereum JSON Format

...

Exercises: Private Key To Blockchain Address

…

Exercise: Generate Random Private Key and Corresponding Blockchain Address

…

Exercise: Generate the Blockchain Address from Existing Private Key

…

Exercises: Sign / Verify Blockchain Transactions

...

Exercise: Sign / Verify Blockchain Transactions in JavaScript

...

Exercise: Sign / Verify Blockchain Transactions in Python

...

Exercise: Sign / Verify Blockchain Transactions in C# and .NET Core

...

Exercise: Sign / Verify Blockchain Transactions in Java

...

Conclusion

...

Practical Blockchain Cryptography for Developers

Hashes, HMAC, Key Derivation, Scrypt, AES, Elliptic Curve Cryptography (ECC), ECDSA, Digital Signatures, Blockchain Cryptography, Ethereum, Blockchain Addresses, Keys, Wallets, BIP39, BIP44, Ethereum Signatures, Ethereum Wallets

The "Practical Blockchain Cryptography for Developers" book introduces the most important concepts of the elliptic curve cryptography (ECC), key derivation and encryption (SCrypt, AES and HMAC), digital signatures (ECDSA, sign / verify) and crypto-wallets (HD wallets, mnemonics and BIP39, BIP44 and key derivation) with live examples in Python, JavaScript, C# and Java.

In book the author explains the key cryptographic concepts used in the blockchain systems and demonstrates them with code examples:

- Elliptic curve cryptography (ECC), ECC concepts, curves, secp256k1, ed25519
- From private key to public key to blockchain address
- Generating Ethereum addresses / signing / verifying signed messages
- Cryptographic hash functions like SHA256, SHA3, RIPEMD160, …
- HMAC and key derivation, key-derivation functions like HMAC-SHA256, PBKDF2, Scrypt
- Blockchain wallets: simple keystores and HD wallets (BIP39 and BIP44), wallet formats (like JSON / UTC), wallet encryption (AES + padding + CBC/CTR, Scrypt, HMAC)
- Ethereum signatures, Ethereum addresses, keys, wallets, APIs

Svetlin Nakov, PhD

Co-Founder @ SoftUni,

Blockchain Developer and Trainer

Svetlin Nakov

SoftUni – https://softuni.org

ISBN 978-619-00-0802-6

9 786190 008026 >